Extensión de Lógicas Temporales con Nociones Deónticas para la Especificación y Análisis de Sistemas Tolerantes a Fallas

por Cecilia Noelia Kilmurray

Presentado ante la Facultad de Matemática, Astronomía y Física
como parte de los requerimientos para la obtención del grado de
Doctor en Ciencias de la Computación de la
UNIVERSIDAD NACIONAL DE CÓRDOBA

Junio, 2020
@FaMAF - UNC 2020
Director: Dr. Pablo Francisco Castro

Resumen

Durante estas últimas décadas tanto las computadoras como los programas han ido cambiando su presencia en la vida de las personas, ya que a diferencia del pasado hoy las computadoras se encuentran cada vez más presentes en objetos de la vida cotidiana: teléfonos celulares, televisores, autos, aviones, válvulas que inyectan insulina, cajeros autómaticos y un sinfín de elementos que nos rodean contienen piezas de software para cumplir con su función. Una característica de muchos de los programas de computación que se ejecutan en estos objetos tecnológicos, es la necesidad que tienen de interactuar con otros sistemas, ya sea localmente o través de redes. Lo cual ocasiona que la complejidad de los mismos sea cada vez mayor, como así también la necesidad en muchos de ellos de garantizar disponibilidad y confiabilidad en todo momento. Esto es debido principalmente a que en ciertos casos, una falla de estos tendrían costos muy altos a nivel económico, o inclusive podrían poner en riesgo vidas humanas, o afectar el medio ambiente [87].

Toda esta automatización de tareas de la vida diaria ha provocado que el área de la computación tolerante a fallas haya ido tomando cada vez mayor relevancia, ya que una de las características deseable en este tipo de sistemas críticos, es la de poder garantizar cierto comportamiento *esperado*, a pesar de la ocurrencia ocasional de fallas.

En este trabajo presentamos algunos formalismos lógicos que resultan adecuados para la especificación, y luego la verificación, de propiedades de sistemas tolerantes a fallas. En particular, nos enfocamos en el uso de aquellos que si bien, tradicionalmente fueron utilizados para representar y analizar la estructura lógica de *normas* o *leyes* (conocidos con el nombre de *lógicas deónticas*), nos posibilitan, a diferencia de otros enfoques, distinguir entre el comportamiento *normal* y *anormal* de un sistema. Estos formalismos, además, incorporan nociones temporales los cuales han mostrado ser de extrema utilidad para especificar y verificar sistemas concurrentes y reactivos.

Hacia el final de esta tesis, además se presentan algunas incursiones hechas en el área de sistemas probabilistas, ya que cuando se piensa en sistemas tolerantes

a fallas surge naturalmente pensar en un *grado* de tolerancia/robustez deseado o esperado; y es justamente este tipo de noción cuantificable la que nos lleva a pensar en utilizar las probabilidades para capturar este concepto.

En particular la forma en que las lógicas probabilistas utilizan dicha noción hacen que resulten adecuadas para este objetivo. En este trabajo presentamos una variante de lógicas probabilistas, la cual a través de operadores que tienen una semántica de punto fijo capturan el concepto de recursión, permitiendo expresar propiedades acerca de la estabilidad de las ejecuciones del sistema. Dichas propiedades son comunes en aquellos escenarios donde uno está interesado en verificar cuando un sistema permanece, o revisita, un conjunto de estados *seguros*.

La principales características de todas las lógicas presentadas en este trabajo son evaluadas a través de diferentes casos de estudios tomados de bibliografía de ésta área y algunas variantes de los mismos definidas por nosotros para lograr exhibir la potencia de estos diferentes formalismos lógicos.

Palabras Claves: Métodos Formales, Tolerancia a Fallas, Lógicas Deónticas, Model Checking, Verificación de Software.

Abstract

During the last decades both computers and programs have been changing their presence in people's lives. Unlike years ago, today computers are increasingly present in daily life objects: cell phones, TV, cars , airplanes, insulin pumps, ATMs and a myriad of elements that surround us depend on software to fulfill their function. A characteristic of many of the computer programs that run on these technological objects is their need for interaction with other systems, either locally or through communication networks. This causes an increase in their complexity, as well as the need in many of them to guarantee availability and reliability at all times. This is mainly due to the fact that in some cases, a failure would have very high costs at an economic level, or could even put human lives at risk, or affect the environment [87].

All this automation of daily life tasks has caused the area of fault tolerant computing to have become increasingly important, since one of the desirable characteristics in this type of critical systems is to be able to guarantee certain *expected* behavior, despite the occasional occurrence of failures.

In this work we present some logical formalisms suitable for the specification, and later verification, of properties of fault tolerant systems. In particular, we focus on the use of those formalisms traditionally used to represent and analyze the logical structure of *norms* or *laws* (known by the name of *deontic logics*), that allows us to distinguish between *good*(normal) and *bad* (faulty) behavior of a system.

These formalisms also incorporate temporal notions which have proven to be extremely useful for specifying and verifying concurrent and reactive systems.

Towards the end of this thesis, some forays made into the area of probabilistic systems are also presented, due that when thinking about fault tolerant systems it naturally arises the notion of a desired or expected *degree* of tolerance / robustness; and it is precisely this type of quantifiable notion that leads us to think about using probabilities to capture this concept.

In particular, the way in which probabilistic logics use this notion makes them

suitable for this goal. In this work we present a variant of probabilistic logics, which through operators that have a fixed point semantics capture the concept of recursion, allowing us to express properties about the stability of the system executions. Such properties are common in those scenarios where one is interested in verifying if a system remains, or revisits, a set of *safe* states.

The main characteristics of all the presented logics in this work are evaluated through different case studies taken from the bibliography of this area and some others variants of them that we defined in order to show the power of those different logical formalisms.

Keywords: Formal Methods, Fault Tolerance, Deontic Logic, Model Checking, Software Verification.

Agradecimientos

Son muchas las personas a las que tengo que agradecer, ya que de diferentes maneras todas ayudaron a que llegara este hermoso momento en mi vida.

Sobre todo a Pablo gracias por toda tu paciencia, por tu generosidad y por todo lo que he aprendido a lo largo de estos años en los que trabajamos juntos. Gracias por siempre incentivarme para seguir adelante y por comprenderme.

Un agradecimiento especial para Naza, que fue el impulsor que comenzara a transitar este camino.

Gracias a todos mis compañeros del DC: Sonia, Vale, Marta, Pancho, Negro, Gastón, Germán, Marcelo A., Pablo, Ernesto, Marcelo U., Chino, Simón, Mariano, César, Facu y Luciano. Y no puedo dejar de nombrar a Nico y Zurdo con los también compartimos parte de este camino juntos.

Gracias Nir Piterman por permitirme trabajar contigo, compartir tus ideas y tu profesionalismo a la hora de investigar. Fue una gran experiencia que no olvidaré.

A Pedro por siempre compartir generosamente su conocimiento y darme consejos para mejorar mi trabajo.

A Daniel, Javier y Naza mi Comisión Asesora del doctorado, gracias por sus consejos y devoluciones.

A Laura, Raúl y Charly, quienes fueron jurados de este trabajo, es un gran honor para mi que hayan aceptado la revisión del mismo. Gracias por la lectura minuciosa y sus observaciones, que además de aportar a la calidad de este trabajo, fueron de gran enseñanza para mi carrera.

A la Universidad Nacional de Córdoba, en particular a la Facultad de Matemática, Astronomía, Física y Computación (FAMAF) y a todas las personas que allí trabajan, por su cordial e inmediata respuesta a todo lo que he necesitado.

Gracias Universidad Nacional de Río Cuarto, por haberme formado no sólo en mi carrera de grado, sino permitirme crecer como persona día a día haciendo lo que me gusta.

Sin duda, este trabajo no hubiera sido factible sin el financiamiento que me

otorgaron el Consejo Nacional de Investigaciones Científicas y Técnicas (CONICET), y el Fondo para la Investigación Científica y Tecnológica (FONCYT) perteneciente a la Agencia Nacional de Promoción de la Investigación, el Desarrollo Tecnológico y la Innovación. Mi más sincero agradecimiento a estas instituciones.

A mis amigos personales Ramiro, Yovi, Ale, Laura, Naty, Romi, Ana, Super y Chin, gracias por estar siempre a mi lado, por brindarme siempre su cariño y sus consejos justos.

A mis papás y mi hermano Gera, pilares fundamentales en mi vida. Por siempre brindarme todo su amor y los valores trasmitidos que hacen de mi lo que soy hoy.

Finalmente, gracias Juan por todo tu amor y por compartir juntos este camino que es la vida. Todo es mejor a tu lado!. Y sobre todo a Manu y Lucy que son los seres más importantes de mi vida.

Índice general

Índice de figuras

Índice de cuadros

Capítulo 1

Introducción

Como expresamos previamente, el cambio en la importancia del rol desempeñado por los sistemas de computación en las diversas actividades humanas ha traído como consecuencia que los niveles de confiabilidad y disponibilidad requeridos para los sistemas informáticos sean cada vez más altos. Debido a esto, el desarrollo de mecanismos que permitan razonar e implementar programas con estas características ha sido uno de los principales focos de atención en el área de *ingeniería* de *software*. Los formalismos lógicos han demostrado ser una pieza esencial a la hora de diseñar y verificar sistemas; en particular a la hora de especificar y verificar sistemas críticos, donde un comportamiento inesperado de los mismos podrían provocar grandes pérdidas financieras, afectar la integridad de personas o el medio ambiente.

En la última década, han surgido una gran cantidad de formalismos para asistir en la descripción de sistemas críticos; y en particular un grupo de estos se han abocado a caracterizar la capacidad de dichos sistemas de brindar el comportamiento *esperado*, aún luego de la ocurrencia de una *falla*. Sin embargo, algo sorprendente de esta área, es la falta de entornos de propósito general que permitan, tanto el análisis, como la verificación de la tolerancia a fallas. Ya que muchas de las soluciones propuestas para problemas de tolerancia a fallas son *ad hoc*, y se encuentran enfocadas a problemas específicos en contextos particulares. Por ejemplo en el caso de técnicas para tolerancia a fallas en software, las soluciones más importantes tienen que ver con redundancia o diversidad. La diversidad puede ser diversidad de diseño, como en *N-version programming* [21, 35] y bloques de recuperación, o diversidad de datos, como en los bloques de reintento y la *N-copy programming* [10, **?**]. 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. Por otro

lado, pese a que los métodos formales contribuyen a la construcción de sistemas libres de *bugs* en el código fuente (los cuales son una de las causas fundamentales de fallas en sistemas de computación); dichos enfoques son insuficientes, dado que no atacan, en general, el problema de lidiar con fallas provenientes de eventos externos al software, que en general no pueden ser previstos.

Más aún, el estudio de la tolerancia a fallas ha influenciado más recientemente áreas del diseño de software, como se puede observar con el origen de áreas como *self management*, la cual hace referencia a sistemas que pueden manejar sus ejecuciones adaptándose a los cambios externos o internos sin la intervención humana. A ésta área se la podría dividir en distintas sub-areas de acuerdo el objetivo que persiguen, por ejemplo: *self healing*, donde se busca diseñar sistemas que automáticamente detecten anomalías o fallas y logren revertir la situación de manera temporal o permanente; otro ejemplo son los llamados sistemas *self adaptative*, cuyo objetivo es lograr obtener sistemas que durante su ejecución monitoreen el ambiente y modifiquen sus acciones con el objetivo de mantener y garantizar un comportamiento normal o esperado.

El objetivo de esta tesis es contribuir en este sentido y lograr definir un *framework* lógico que se adecue a las características de este tipo de sistemas, permitiendo capturar sus propiedades de una manera natural, e intuitiva, para luego realizar el proceso de verificación de las mismas. Para ello nos enfocamos en el uso de lógicas *deónticas temporales*, las cuales combinan dos conceptos habituales en las teorías normativas: la descripción y prescripción de comportamiento. Por un lado los operadores *deónticos* permiten distinguir entre el comportamiento *normal* o *esperado* de un sistema, de aquellas situaciones en la que se comporta de una manera *inesperada* o *anormal*. Estos operadores representan nociones como *obligación*, *permiso* o *prohibición*; y si bien tradicionalmente han sido utilizados en el ámbito jurídico para analizar la integridad lógica de normas y leyes, se pueden trasladar facilmente a la ingeniería de software, donde por ejemplo la prohibión de una falla o la obligación de los requerimientos minimos de un sistema son propiedades que aparecen con mucha frecuencia. El otro componente que incluyen este tipo de lógicas es el que brindan los operadores temporales, con los cuales se puede predicar acerca de propiedades de las ejecuciones de los sistemas, permitiendo razonar sobre la ocurrencia de eventos a través del tiempo. Estos permiten expresar nociones de *invarianza* o *eventualidad*, por ejemplo: *siempre* en el futuro o en *algún* momento futuro se cumple cierta propiedad en las ejecuciones del sistema. Es por ello que el hecho de combinar estos dos tipos de operadores nos brindan cierta flexibilidad a la hora de expresar propiedades de los sistemas tolerantes a fallas, donde naturalmente surgen este tipo de nociones.

Finalmente otros formalismos que resultan interesantes son los llamados lógicas temporales probabilistas, las cuales permiten razonar sobre la probabilidad de las ejecuciones de los sistemas de computación, por lo cual resultan adecuadas para razonar sobre fallas. En este trabajo presentamos brevemente tres variantes de lógicas probabilistas, las cuales a través de operadores de punto fijo u operadores modales que capturan el concepto de recursión, permiten expresar propiedades acerca de la estabilidad de las ejecuciones del sistema. Dichas propiedades son muy comunes en aquellos escenarios donde uno está interesado en verificar cuando un sistema permanece, o revisita, un conjunto de estados *seguros*.

1.1. Tolerancia a Fallas

La primera intuición que se tiene cuando se piensa en *sistemas tolerantes a fallas*, es la de sistemas que siempre funcionan de la manera esperada, haciendo imperceptible la posible ocurrencia de una falla. Y si bien esto es correcto, no siempre es posible, o incluso necesario. Por lo que la tolerancia a fallas puede definirse de acuerdo a distintos niveles de tolerancia o robustez, aceptando por ejemplo cierto grado de *degradación* en la funcionalidad brindada ante la presencia de una falla en sistemas menos críticos. En esta sección presentaremos algunos de los conceptos más relevantes de ésta rama de la ciencia la cual, como hemos expresado previamente, ha ido ganando relevancia en el ámbito científico debido principalmente a la necesidad de garantizar la disponibilidad y confiabilidad de sistemas críticos, donde una falla tendrían costos muy altos a nivel económico, o inclusive podrían poner en riesgo vidas humanas, o bien afectar el medio ambiente.

Comenzaremos citando algunas definiciones que han sido ampliamente divulgadas:

- *Algirdas Avizienis* en [20], considera que un sistema es tolerante a fallas si los programas pueden ser correctamente ejecutados aún ante la ocurrencia de fallas.
- *Wilfredo Torres-Pomale* en [123], establece que la función del *software* tolerante a fallas es prevenir los accidentes o eventos indeseables y en lo posible, *enmascarar* las fallas.
- *Anish Arora* y *Mohamed Gouda* en [15], definen la tolerancia a fallas en función de dos conceptos, *Clausura* y *Convergencia*. Estos autores consideran que un

sistema es capaz de *"tolerar"* una serie de *fallas*, siempre y cuando se cumplan las siguientes condiciones:

1. **Clausura**: Si cuando ocurre una falla el sistema se encuentra en un estado del conjunto de estados *"legales"* o *"correctos"* del mismo, entonces el estado resultante será un estado de un conjunto de estados T cualesquiera, y si las fallas continuan ocurriendo, dicho sistema permanece en estados de T.
2. **Convergencia**: Si las fallas dejan de ocurrir, el sistema eventualmente retornará a un estado del conjunto de estados *"legales"* del sistema.

Como se puede apreciar estos autores admiten que ante la presencia de fallas el sistema permanezca *temporalmente* dentro de un subconjunto de estados, donde no necesariamente se brinde el comportamiento *ideal* esperado. Lo cual si es requerido en el caso de que las fallas cesen.

1.1.1. Conceptos básicos:

Los conceptos que se presentan a continuación forman parte del vocabulario común en el ámbito de tolerancia a fallas, los cuales permiten distinguir las distintas nociones relacionadas a la falla de un sistema:

- **Falla** (***Fault***): Intuitivamente una *falla* es lo que causa un *error*. Es decir, una *falla* es una *anomalía* presente en un componente de *hardware* (ej. *defecto*) o del *software* (ej. *bug*).
- **Error**: Se considera un *error* al estado en el que queda un sistema luego de la ocurrencia de una *falla*.
- ***Failure***: Se considera que un sistema ha fallado, cuando como consecuencia de una falla no puede brindar el comportamiento o la funcionalidad esperada. Es decir, cuando la falla se hace perceptible para el usuario (ya sea una persona u otro sistema que interactúa con el sistema en cuestión).

Notar que si bien las *fallas* pueden ser externas o internas al sistema, en general ocurre que las fallas externas vienen dadas por una falla interna previa, la cual habilita a la falla externa a provocar un error del sistema. Por otro lado, cabe aclarar que se considera que una falla está *activa* cuando provoca que el sistema entre en

un estado de error, caso contrario se dice que la misma está *inactiva* o que es una falla *latente*. Otro concepto que suele aparecer en la bibliografía es el de *propagación de error*, el cual se refiere a aquellos errores que son provocados por un error previo del sistema, por ejemplo un caso clásico es la utilización del valor incorrecto de una variable, la cual por una falla previa del sistema no fue correctamente actualizada.

Neeraj Suri en [122] presenta una clasificación de fallas de acuerdo a distintos criterios, la cual resulta útil para analizar el tipo de fallas que cada técnica de tolerancia a falla puede manejar. Por ejemplo considera que una falla es *simétrica* si la misma es percibida de igual manera por todos los subsistemas, si en cambio algunos componentes del sistema se ven más afectados que otros se considera que es una falla *asimétrica*. Por otro lado, en el caso de fallas en componentes de *hardware*, clasifica una falla como *ramdon* en aquellos casos donde la misma fue provocada por la degradación propia de los componentes (vida útil) o también cuando fue consecuencia del ambiente, por ejemplo: humedad, calor, vibraciones, etc. En el caso que las fallas fueron introducidas accidentalmente al momento de la construcción o diseño del componente se las denomina fallas *genéricas*.

Finalmente otra manera muy común de agrupar las fallas es por si son causadas por anomalías en el *hardware* o en el *software*; en la sección 1.1.3 presentaremos las principales tecnicas utilizadas para lidiar con las fallas en estos casos.

1.1.2. Desarrollo de Sistemas vs Fallas

Con el objetivo de desarrollar sistemas que cumplan con los niveles de seguridad y confiabilidad necesarios en los tiempos actuales, se han implementado a lo largo de los años diversos enfoques, los cuales podrían agruparse en las siguientes cuatro categorías de acuerdo a como intentan lidiar con las fallas presentes en el *software* y al momento en que lo hacen [22, 115, 123] :

- ***Fault prevention*** : Se enfoca en la construcción de sistemas libres de *fallas*, previniendo o reduciendo al máximo su introducción desde etapas tempranas del desarrollo. En lo que se refiere a *software*, la obtención de programas con altos niveles de calidad implica la utilización de herramientas, metodologías y técnicas rigurosas. Es por ello que los métodos formales resultan de particular interés, ya que a través de formalismos lógicos y teorías permiten probar matemáticamente que el *software* construído se encuentra libre de *fallas*.

- ***Fault tolerance***: Su aplicación busca la construcción de sistemas que sean capaces de sobrellevar las fallas que no fueron detectadas en instancias previas

del desarrollo. De manera que al momento en que una falla se produzca, el *software* tenga los mecanismos necesarios para evitar que el sistema llegue a un estado de *failure*, donde no pueda comportarse de la manera esperada. En las sección 1.1.3 conoceremos algunas de las principales técnicas utilizadas.

- ***Fault removal***: Tiene como objetivo la detección y eliminación de *fallas* que han sido introducidas accidentalmente durante el proceso de desarrollo de *software*. Para ello se apela al uso de técnicas como *testing*, revisiones de código, simulación e incluso *model checking*; este último se puede utilizar para determinar si la implementación se corresponde con la especificación del mismo. En particular, el testing es la técnica más difundida a la hora de detectar *bugs* en el *software*, su gran popularidad se debe principalmente a que gracias a las múltiples técnicas existentes se puede realizar durante las distintas fases del desarrollo, permitiendo descubrir fallas desde etapas tempranas del diseño y a lo largo de su implementación. Sin embargo, el *testing* no garantiza que se eliminen la totalidad de las *fallas*, por lo que en general se combina con otras técnicas para obtener *software* de mayor *confiabilidad*.

- ***Fault forecasting***: Este mecanismo intenta *predecir* la futura ocurrencia de fallas, de manera de prevenirlas o, en caso de no poder evitar su aparición, intentar minimizar sus consecuencias. Para poder realizar este tipo de predicciones se utilizan técnicas de estimación basadas en la ocurrencia de fallas del pasado o de resultados de *testing* realizados durante el proceso de desarrollo, y el uso de herramientas para análisis e interpretación de los resultados. Este tipo de técnicas sirve en algunos casos para estimar la efectividad de futuras tareas de *testing*. Conjuntamente con *fault removal* representan un buen indicador para determinar si el *software* se corresponde con los requerimientos de confiabilidad y seguridad especificados para el mismo.

Notar que los primeros dos métodos están orientados a ser aplicados durante el proceso de *construcción* del *software*, mientras que *Fault removal* y *Fault forecasting* están más bien involucrados en las tareas de *validación* llevadas a cabo una vez finalizado el desarrollo.

Sin embargo, ninguno de estos enfoques puede garantizar la obtención de sistemas 100 % libre de fallas, es por ello que en muchos casos se combinan las distintas técnicas, logrando así incrementar los niveles de confiabilidad. Ya que si bien, con técnicas como *testing* o revisiones de código, se logra encontrar y eliminar la presencia de una gran cantidad de defectos del software, no se puede demostrar la ausencia de

los mismos (como fuera remarcado por E. Dijkstra). Por otro lado, en enfoques como *Fault prevention* la implementación y puesta en práctica de metodologías rigurosas, como lo son los métodos formales, implican un alto costo en tiempo y esfuerzo para poder verificar que las aplicaciones satisfacen los requerimientos especificados; o inclusive en algunos casos resulta inviable su aplicación debido a sus limitaciones (por ej. *model checking*).

A raíz de esto, técnicas como *fault tolerance* representan una buena opción para el manejo de las fallas, teniendo ciertas ventajas sobre los demás enfoques. Ya que dado el caso improbable que se lograra eliminar la totalidad de las anomalías de una aplicación, el resto de los métodos no contemplan la aparición de fallas provenientes del entorno (por ejemplo fallas originadas desde el sistema operativo o eventos inesperados causados por otras aplicaciones). A través de esta visión más general de las fallas se busca incluir, ya sea en el diseño del *software* o bien en el código fuente del mismo, mecanismos que garanticen que el sistema se comporte adecuadamente ante la presencia de fallas. Para ello existen diversas técnicas las cuales son seleccionadas dependiendo principalmente de los requerimientos de la aplicación. Sistemas críticos, requieren un alto grado de confiabilidad y seguridad, en cambio en el caso de aplicaciones comerciales o personales, los requerimientos son muchos menores, ya que el uso de tolerancia a falla en este tipo de sistemas es más bien considerado un valor agregado que está mayormente relacionado a la disponibilidad del sistema desde la percepción de *usuario*. A continuación presentaremos las principales características de las técnicas existentes.

1.1.3. Técnicas de Tolerancia a Fallas

A grandes rasgos las técnicas de tolerancia a fallas pueden agruparse en dos grandes categorías, dependiendo si son aplicadas a nivel de *software* o a nivel de *hardware*.

En esta sección analizaremos las particularidades de estas técnicas y los recursos a los que apelan a la hora de intentar mitigar los efectos de las fallas presentes en los sistemas. En el caso del *software* al no existir físicamente no presenta fallas asociadas a la degradación propia de los componentes, como si ocurre con el *hardware*. Sino que en general se relacionan a la presencia de *bugs* en el diseño o en código fuente. Las fallas de *software* son consideradas la principal causa de la falla de sistemas [36]. Las fallas provocadas por desperfectos en el *hardware*, se caracterizan, en general, por el hecho de ser *ramdon* y de alguna manera más *independientes*, por lo que la *redundancia* de componentes, suele resultar de gran efectividad a la hora de

prevenirlas o minimizar sus consecuencias.

En relación a los efectos provocados por ambos tipos de fallas, el espectro puede ser muy variado, ya que puede ir desde inconvenientes menores, donde simplemente se requiera reiniciar la aplicación o el componente con problemas; hasta catástrofes que provoquen daños físicos en seres vivos o en el medio ambiente. De acuerdo a la magnitud de las fallas se ven reflejadas las consecuencias económicas, comerciales, sociales y/ó ambientales. A nivel económico existen algunos casos donde las fallas han provocado pérdidas millonarias por la rescinsión de contratos, retiro o reemplazo de productos de las líneas de venta, desembolsos de indemnizaciones a las personas afectadas o a sus familiares en aquellas situaciones donde se han perdido vidas, etc. En lo que se refiere a las consecuencias socio-ambientales, en muchos de los casos, los efectos provocados en el medio ambiente son irreversibles o bien lleva años reestablecer el equilibrio de las zonas afectadas[87].

Tolerancia a Fallas en *Software*

La tolerancia a fallas en *software* involucra la incorporación de programas, módulos o funciones adicionales a la implementación y diseño de un sistema, con el objetivo de que sea capaz de soportar las fallas.

Una característica interesante de este tipo de técnicas, es que no necesariamente tienen que ser implementadas sobre el sistema completo, sino que pueden aplicarse sólo a ciertos procedimientos o procesos críticos donde se requiera garantizar su correcto funcionamiento. Sin embargo existen técnicas con las que se pueden incluir mecanismos de tolerancia que abarquen la totalidad del sistema e inclusive contemplan fallas de aplicaciones con las cuales interactúa, como el sistema operativo.

A la hora de proveer al *software* de mecanismos que le permitan lidiar con fallas, existen dos tipos de técnicas, las que se enfocan en el diseño de una única pieza de *software* independiente capaz de soportar fallas, o las que atacan el problema de una perspectiva más general, a través del diseño de múltiples versiones de un módulo de *software* dado, las cuales se organizan de manera secuencial o en paralelo y mediante las cuales se implementan los mecanismos de detección, contención y recuperación de fallas.

- **Tolerancia a Fallas Simple** Existen diversas técnicas basadas en este enfoque, muchas de las cuales basan sus implementaciones en el uso de *redundancia*, *detección de errores*, *manejo de excepciones*, *diversidad de datos*, etc. Sin embargo algo en común de todas ellas es que atacan el problema mediante la descomposición del *software* en varias piezas o módulos lo más independiente

posible entre sí. Este tipo de diseño modular se complementa en muchos casos, con técnicas como *system closure*, [46] donde ninguna acción puede ejecutarse al menos que la misma esté explícitamente *permitida*. Bajo este enfoque se puede controlar y limitar la propagación de errores, habilitando o deshabilitando acciones de un componente en conflicto, al momento que se detecta una falla y así evitar que provoque más daño del inicialmente generado.

Otra manera de controlar la forma en que interactúan los componentes de un sistema, es a través del concepto de *acción atómica* la cual es una actividad en la que intervienen un conjunto particular de componentes, sin tener ningún intercambio de información con el resto de módulos del sistema, durante la ejecución de la misma. Para el resto de los componentes del sistema dicha actividad es considerada como una acción indivisible con dos posibles salidas: termina correctamente o es abortada por la detección de una falla. De este modo, en el caso de una falla, se logra *contener* la misma acotando su alcance únicamente a los componentes involucrados en la acción atómica. Las acciones de recuperación también son disparadas solamente para los componentes afectados [11].

En lo que respecta a *detección de errores*, pueden ser implementada dentro los módulos o bien a la salida de los mismos dependiendo el enfoque. Para realizar la detección, es necesario realizar chequeos que permitan establecer la existencia o no de un error. Una manera muy simple es a través de la duplicación de piezas de *software*, y comparando luego sus resultados. En el caso de sistemas que tienen requerimientos o límites de tiempo específicos, una forma eficaz de detectar un error es a través de la implementación de *timers* que monitorean los componentes del sistema y ante la falta de respuesta, de uno o varios de ellos, más alla de los límites de tiempo permitido generan la señal o código de error correspondiente. Existen muchas otras técnicas para determinar la presencia de un error de acuerdo a la caracterśtica de la aplicación, una buena fuente para ahondar en su estudio es [123].

Muchos de estos enfoques se complementan entre sí, como es el caso de los *manejadores de excepciones*, para los cuales es necesario la implementación previa de un mecanismo de *detección de errores*, ya que una vez detectado un error se inician las acciones de recuperación con el objetivo de volver al programa a un estado seguro. Una *excepción* es justamente una interrupción del comportamiento *normal* del sistema para poder manejar y sobreponerse de una situación *anormal*. Una vez generada una excepción, el manejador de

excepciones es ejecutado con el objetivo de intentar de restablecer el comportamiento adecuado del programa. Este tipo de mecanismos está ampliamente difundido en el ámbito de la programación, ya que muchos lenguajes lo incluyen y promueven su uso, entre los más conocidos podemos mencionar : *C++*, *JAVA* y *Eiffel*.

Finalmente, la *diversidad de datos* está basada en la idea de que las fallas se manifiestan dependiendo de los valores de entrada del sistema. Por lo que ésta técnica busca, mediante el ingreso de datos lógicamente equivalentes, la obtención de resultados iguales (equivalentes). Para obtener los distintos valores de entrada en general utilizan *data re-expression algorithms* (*DRAs*)[115]. En general la *diversidad de datos* es utilizada de manera combinada con *puntos de control* (*checkpoints*) y *reinicio*, ya que en cada reintento o reinicio del sistema se utilizan datos de entrada equivalentes con el objetivo de aumentar la efectividad de estos métodos y lograr así sobreponerse de las fallas. Los *puntos de control* pueden ser implementados de forma estática o dinámica, en el primer caso se realiza un *snapshot* al inicio de la ejecución del programa o módulo y en caso de detectarse un error se reestablece el estado del sistema a partir del mismo. En cambio, cuando se usan puntos de chequeo dinámicos se toman múltiples snapshots en distintos momentos de la ejecución y ante la presencia de una falla se retoma la ejecución desde el último estado correcto del sistema que tiene guardado, de esta manera se evita reiniciar completamente su ejecución.

De acuerdo a *Russel Abbot*, en [4], para que cualquier técnica de *tolerancia a fallas simple* sea efectiva, es necesario que los módulos que componen el sistema cumplan con las siguientes dos propiedades: *self-checking* y *self-protection*. La primera se refiere a la capacidad de un componente para detectar sus propios errores y tomar las acciones necesarias para así limitar la falla y evitar que el error se propague hacia otros módulos del sistema. En cambio la propiedad de *self-protection* significa que un componente debe ser capaz de protegerse a si mismo detectando errores en la información recibida de otros módulos con los cuales interactúa.

- **Tolerancia a Fallas Múltiple** La idea que inspira este enfoque fue presentada por *Algirdas Avizienis* en [19], donde expresa que si una misma aplicación es desarrollada utilizando distintos equipos, distintas metodologías o distintos lenguajes de programación, las versiones resultantes si bien satisfacen los mismos requerimientos, muy probablemente no tengan las mismas fallas. Es

decir se establece una relación directa de las fallas de *software* con el proceso, las herramientas y el factor humano utilizado para su desarrollo. Por lo que en ésta técnica se busca a través de la utilización de dos o más versiones de un componente de *software*, integradas de acuerdo a algún diseño, sobrellevar la posible ocurrencia de una falla y evitar así que el sistema total llegue a un estado de *failure*. La diferentes variantes pueden organizarse de manera secuencial o ejecutarse en paralelo, de acuerdo al método de tolerancia a falla a implementar. Entre los más conocidos podemos destacar:

- *Programación de N-Versiones* (*N-Version Programming*): En este enfoque cada variante es utilizada en paralelo y el resultado final se decide mediante la aplicación de algún mecanismo de decisión, como *voting*. Cabe destacar que las distintas versiones pueden ser ejecutadas en secuencia, pero en ese caso se necesitan implementar *puntos de control* y *reinicio* para poder reestablecer el estado del sistema luego de que finalice la ejecución de cada pieza de *software*.
- *Bloques de Recuperación*: Consiste en la utilización secuencial de cada módulo ya que al momento de detectarse una falla, se reintenta reanudar el sistema mediante la ejecución de una de las versiones con la esperanza de que en esta nueva ejecución no se repita la anomalía y así lograr recuperarse de la falla. Por lo que en general la implementación de este enfoque se utiliza combinado con *puntos de control* y *reinicio*.

En [123, 115], se pueden encontrar más detalles de éstas y otras técnicas de *tolerancia a falla múltiple*, con ejemplos de su aplicación en casos reales.

Notar que en general, la implementación de mecanismos de detección de errores o el uso de redundancia pueden afectar la performance del sistema, sin embargo a pesar de no agregar funcionalidad al sistema colaboran en la obtención de *software* de mayor calidad. Por lo que a la hora de diseñar un sistema tolerante a fallas, la elección de las técnicas a utilizar suele estar basada en la compensación de un posible detrimento de la performance o un incremento de los costos de implementación, por la confiabilidad, disponibilidad y seguridad ganada o deseable en el sistema final.

Tolerancia a Fallas en *Hardware*

De acuerdo *Wilfredo Torres-Pomale*, en [123], la redundancia en *hardware* puede ser implementada de 3 maneras: *estática, dinámica* o *híbrida*.

- **Redundancia Estática**: Se caracteriza por no incorporar ningún mecanismo de control para detectar la ocurrencia de fallas o para reconfigurar dinámicamente el sistema ante la presencia de una anomalía. Sino que consiste en la replicación de componentes de un sistema a través de los cuales, mediante diversas técnicas como por ejemplo *voting*, intentan enmascarar las posibles fallas y su consecuente propagación. El uso de este tipo de técnicas es muy común a la hora de implementar circuitos. Donde por ejemplo se suelen replicar circuitos idénticos, obteniendo un resultado de cada uno, y luego el valor de salida es definido en función del resultado que aparece el mayor número de veces (*voting*).

  La simplicidad y efectividad de este método hace que sea ampliamente utilizado, sin embargo en ciertos casos la aplicación del mismo resulta inviable, ya sea por los costo que implica tener módulos redundantes, o por ejemplo en sistemas como *satélites*, donde la limitación de peso y espacio hace que la replicación de componentes resulte inconveniente. Por otro lado, el número de fallas tolerado es dependiende de la cantidad de componentes replicados, es decir, si por ejemplo se triplica un módulo y 2 de ellos fallan arrojando resultados incorrectos, con la técnica de *voting* el error se propagará de todas maneras. Formalmente *voting enmascara* hasta $(n-1)/2$ fallas, donde n es el número de componentes redundantes [128].

- **Redundancia Dinámica**: A diferencia de la técnica anterior se implementan mecanismos para *detectar*, *localizar* y generar acciones de recuperación de las fallas, generalmente mediante la *reconfiguración* del sistema. Una manera muy simple de llevar a cabo la detección de una falla en un componente dado, es mediante la *duplicación* del mismo, luego se comparan las salidas de estos módulos que realizaron en paralelo cierta tarea y si las salidas no coinciden, se genera una condición de alarma o evento de falla, a partir del cual se generan acciones de *reparación* con el objetivo de restablecer el funcionamiento normal del sistema; dicha técnica se conoce en la bibliografía con el nombre de *duplication with comparison*.

  En el caso de la *localización* y *reconfiguración*, un claro ejemplo de este tipo de implementaciones es a través del uso de componentes de *repuesto* que pueden estar trabajando en paralelo con el módulo *original* o encontrarse en *stand-by*. Al momento de detectarse una falla, las acciones de reparación implican la habilitación de estos componentes los cuales comienzan a cumplir la funcionalidad del módulo donde se generó la misma (*reconfiguración* del sistema).

De acuerdo a la implementación dada, el módulo donde se localizó la anomalía, puede ser temporalmente inhabilitado o directamente puede requerir su reemplazo.

Respecto a las limitaciones de este enfoque, notar que con *duplicación*, no es posible localizar en cual de los módulos se generó la falla, ni reparar la misma, por lo que en general se combina con otros métodos. Mientras que en las técnicas de *reconfiguración* del sistema, la puesta en funcionamiento de los componentes de *repuesto* ante la presencia de una falla implican un costo extra de tiempo, tiempo que para algún tipo de sistemas puede resultar intolerable.

- **Redundancia Híbrida**: Este método combina elementos de las dos técnicas anteriores, ya que a través de mecanismos de *masking* se intenta contener la propagación de las fallas, y mediante recursos para *detección*, *localización* y *reconfiguración*, propios de la redundancia dinámica, encara el manejo los componentes con fallas. Una implementación muy divulgada es la de *N-Modular Redundancy with Spares*, en la cual un módulo dado es replicado N veces, de manera de enmascarar una posible falla como se hace en *redundancia estática*, pero en este caso se complementan con un grupo secundario de N componentes de *repuesto*, que funcionan en paralelo, los cuales entran en funcionamiento en el caso que se haya detectado una falla. Este enfoque tiene como ventaja que con los resultados arrojados por los módulos de repuesto, el mecanismo de *voting* puede ser restablecido luego de que los componentes fallidos fueron sacado de funcionamiento.

La tolerancia a fallas en *hardware* ha sido aplicada en distintos tipos de sistemas, desde sistemas críticos como controladores de vuelo, sistemas que necesitan funcionar a largo plazo o donde no es posible detener su ejecución, como es el caso de centros de investigación que manipulan material biológico como virus y bacterias, o incluso en artefactos donde donde cualquier acción de mantenimiento del mismo es extremadamente costosa o incluso inviable, ej: satélites. Una buena recopilación de los mismos puede encontrarse en [119].

1.2. Métodos Formales y Tolerancia a Fallas

Muchos métodos formales se enfocan principalmente en la especificación y el diseño de sistemas de software, aunque en la actualidad existen enfoques más generales que involucran todas las etapas del desarrollo. La principal motivación de

los métodos formales es la de garantizar la corrección del software (es decir, que el mismo satisface sus requisitos) y el mecanismo fundamental para lograr esto es la verificación formal. Como los métodos formales contribuyen a la construcción de sistemas libres de bugs, se podría decir que éstos contribuyen también a la construcción de sistemas tolerantes a fallas, en particular teniendo en cuenta que los bugs son la causa fundamental de fallas. Sin embargo, el enfoque tradicional de los métodos formales es insuficiente, dado que no atacan, en general, el problema de lidiar con fallas provenientes de eventos externos al software, no previstos.

A lo largo de las últimas décadas el interés por utilizar y aplicar métodos formales a la hora de especificar y verificar sistemas tolerantes a fallas, se ha visto incrementado. A continuación se presentan algunos de los enfoques más destacados.

- **Verificación de Programas**: Si se revisan los trabajos realizados en las décadas pasadas, se puede apreciar que muchos han sido los casos donde se han aplicado métodos formales para verificar sistemas tolerantes a fallas. Algunos de estos trabajos definieron y utilizaron *frameworks* lógicos los cuales contaban con ciertas reglas que permitian especificar y razonar acerca de fallas, como es el caso de *Flaviu Cristian* el cual presenta una extensión de la lógica de *Floyd/Hoare* con la cual permite razonar, especificar y verificar la correctitud de *software* tolerante a fallas. En este trabajo principalmente se enfocaron en fallas provocadas por componentes de *hardware* o por la *caída* de procesos de un sistema. Otra característica interesante de este *framework* es que las fallas son modeladas como operaciones, que ejecuta el ambiente *adversario*, de manera *random*. En el caso de *Arora* y *Gouda*, estos autores presentan una definición que unifica el concepto de *fault tolerance* a través del uso de predicados de *clausura* y *convergencia* como ya se detalló en la sección 1.1, y presentan una metodología para diseñar y verifificar sistemas tolerantes a fallas, independientemente de la arquitectura, tecnología o aplicación de los mismos. Un punto importante de su trabajo es el uso de invariantes, para lograr distinguir aquellos estados del sistema que se encuentran libre de fallas.

- **Model Checking**: En Tolerancia a Fallas, como en muchas otras áreas de la ciencias de la computación, la necesidad de comprender y analizar las propiedades que se encuentran presentes en este tipo de sistemas, ha llevado a fomentar el desarrollo de procedimientos automáticos que reciban como entrada modelos de los sistemas, y condiciones o propiedades que se desean analizar en los mismos, y que produzcan como resultado respuestas acerca de si se cumplen

o no las propiedades en cuestión, en dichos modelos. Este proceso suele denominarse verificación [67]. En particular *Model checking* constituye un método automático para la verificación de sistemas de estados *finitos*, es decir modelos con un número acotado de estados y determina mediante la exploración *exhaustiva* si el modelo dado cumple o no con cierta propiedad especificada. Respecto a los trabajos desarrollados con este método en el área de Tolerancia a Fallas se puede mencionar el trabajo de *Yokogawa* [127], donde los autores utilizan el *model checker SMV* [105, 104], para verificar sistemas tolerantes a fallas; para realizar la especificación de los sistemas a verificar utilizaron una variante del lenguaje que había presentado *Anish Arora* en su tesis de doctorado [14]. Por otro lado, otro trabajo para destacar es el de *Kouvaros* y *Lomuscio*, publicado en 2017 [78], donde los autores presentaron un método para insertar una serie de *comportamientos* "*anormales*" en ciertos modelos de *template*; los cuales luego son utilizados para generar una familia de sistemas multi-agentes de diferente tamaño, donde el tamaño constituye un parámetro de estos modelos. Cabe destacar que este equipo de investigadores cuentan con una serie de trabajos previos donde han presentado diferentes enfoques para verificar sistemas multi-agentes, este último trabajo busca generalizar muchas de las ideas desarrolladas en esos trabajos previos [96, 53]. Otros artículos interesantes para analizar del área son [25, 60, 86, 121], donde se presentan y aplican diversas técnicas para verificar sistemas tolerantes a fallas. Finalmente en el siguiente capítulo analizaremos algunas de las técnicas y enfoques más utilizados en *Model Checking* a la hora de verificar sistemas (ver sección 2.3).

- **Transformación de Programas** La idea detrás de este tipo de técnicas es aplicar ciertas operaciones al programa original para obtener otro programa semánticamente equivalente con algún objetivo particular. La transformación de programas tiene aplicaciones en muchas áreas de *Ingeniería de Software* como *Compiladores*, *Síntesis de Programas*, *Ingeniería Inversa*, *Optimización de Programas*, entre otras. En particular, en tolerancia a Fallas, este tipo de técnicas buscan introducir mecanismos para detectar, tolerar y/o corregir fallas en programas que originalmente no tenían estas características. Algunos trabajos interesantes para mencionar con este enfoque es el de *Arora* y *Kulkarni* quienes en un primer trabajo en 1998 [16], definieron una clase de componente que permitía la detección y corrección de fallas, y en dicho artículo plantean la transformación de programas mediante la incorporación de este tipo de componentes a sistemas que no tenían ningún mecanismo de tolerancia a fallas. En

este primer trabajo los autores demuestran que con ese tipo de componentes se pueden implementar los tipos de tolerancia usuales. Años después en [80], los autores presentan un método para incorporar la tolerancia a fallas, indicando que esta incorporación puede ser hecha en tres niveles de tolerancia de acuerdo al tipo de propiedad que se desee garantizar: *failsafe*, *nonmasking* ó *masking*. Otros enfoques interesantes para considerar son los presentados en [92], [58], [23] y [89] en los cuales se describen diferentes técnicas para transformar programas sin tolerancia a fallas en programas que posean diversos mecanismos de tolerancia a fallas.

- **Lógicas y Lenguajes de especificación**

  En lo que se refiere a Tolerancia a Fallas, uno de los grandes desafíos es lograr capturar a través de una especificación la ocurrencia de fallas, y así permitir representar, expresar y razonar acerca de las condiciones que se cumplen en un sistema antes y después de la ocurrencia de estos eventos anormales, inesperados o incorrectos. Es por ello que varios investigadores se han dedicado a lo largo de los años a intentar expresar formalmente los comportamientos de los sistemas tolerantes a fallas, para ello se han utilizado diversos formalismos y herramientas, algunos de ellos a los que podemos referirnos son [25, 127, 117, 45, 57, 94, 93, 84, 82]. En general, muchos de estos lenguajes o lógicas no tienen ningún constructor u operador especial para modelar sistemas tolerantes a fallas en términos de poder distinguir entre el comportamiento correcto o ideal y el comportamiento anormal. Sin embargo, muchos de estos enfoques han logrado codificar con éxito a la mayoría de estos comportamientos utilizando mecanismos *ad-hoc* como parte del diseño de los sistemas, por ejemplo este es el caso del trabajo realizado por *Lamport* y *Merz* [82], donde los autores especificaron utilizando una lógica temporal de acciones, denominada *TLA+* el clásico caso de estudio de los generales Bizantinos [83]. Algunos sistemas que han sido muy utilizados como casos de estudio son los relacionados con control de trenes de pasajeros, ya que los mismos contituyen sistemas críticos donde una falla puede provocar pérdidas millonarias y sobre todo puede afectar vidas humanas. En este aspecto, un trabajo destacable en el área es el realizado por *Abrial* y *Hallerstede* [6]. Estos investigadores utilizaron el lenguaje *Event-B* para modelar un componente de software que controla los movimientos de los trenes sobre las vías del tren. Pero además de la especificación del modelo, su principal objetivo fue analizar y verificar propiedades que permitieran prevenir situaciones donde el sistema de trenes entrara en riesgo

de provocar un accidente. Otros trabajos que han especificado y verificado sistemas de control de trenes son [62] y [5]. En el libro [7] se pueden encontrar otro tipo de sistemas modelados como protocolos para transferencia de archivos con reintentos, donde el autor plantea como lidiar con los problemas de tolerancia a falla y sobre todo como razonar formalmente sobre ellos.

Si nos referimos a lenguajes de especificación y modelado, no podemos dejar de mencionar la utilización de *Alloy* [69]. En particular en [75], los autores abordan un caso de estudio donde se incorporan nociones para manejar escenarios con fallas. En este artículo modelan un sistema de archivos donde definen un mecanismo de recuperación para aquellas situaciones donde las operaciones del sistema puedan fallar debido a un corte de energía eléctrica. Por último vale recalcar que, desde un punto de vista lógico, la distinción entre el comportamiento normal y anormal de un sistema puede lograrse incorporando algunas nociones de lógicas deónticas. Por ejemplo, en [56] realizan una extensión de lógicas temporales con operadores de obligación de manera de poder expresar conceptos como *Robustez*. Otra manera de incorporar estas nociones se ve reflejada en el trabajo realizado por *Coenen* [43, 42], donde el autor define una extensión de la lógica de *Hoare* para poder expresar propiedades ante la presencia de excepciones, por mencionar algunos. Finalmente, no podemos dejar de nombrar trabajos como [98, 27], donde los investigadores presentan diferentes formalismos lógicos con operadores deónticos con el objetivo de modelar y verificar sistemas con tolerancia a fallas, cabe aclarar que en las siguientes secciones continuaremos analizando con mayor detalle algunos de estos y otros trabajos notables de este enfoque.

1.3. Motivación y Objetivos de esta Tesis

La falta de entornos para el análisis y verificación de sistemas tolerantes a fallas, es uno de los puntapiés iniciales que motivaron este trabajo. Ya que muchas de las soluciones dadas a problemas de tolerancia a fallas son como mencionamos antes, implementadas *ad hoc*, y atacan problemas específicos en contextos particulares. El objetivo que se persigue en esta tesis es por un lado el estudio de la teoría de diferentes formalismos lógicos para la especificación de sistemas tolerantes a fallas y por el otro, su puesta en práctica en el proceso de verificación de los mismos. En particular, nuestro trabajo consiste en el uso de aquellos formalismos que si bien, tradicionalmente fueron utilizados por juristas y filósofos para representar y analizar

la estructura lógica de *normas* o *leyes* conocidos con el nombre de *lógicas deónticas*, nos posibilitan, a diferencia de otros enfoques, distinguir entre el comportamiento *normal* y *anormal* de un sistema. Permitiendo capturar algunas propiedades de tolerancia a falla de una manera más natural e intuitiva, gracias a la analogía que puede establecerse entre la ocurrencia de fallas en los sistemas, que podrían considerarse como *violaciones* de los requerimientos y donde se espera que exista algún mecanismo que permita reestablecer o corregir dicha situación, y lo que ocurre a nivel jurídico cuando se incumple con una ley o norma, donde se requiere la aplicación de medidas correctivas.

1.4. Estado del Arte

Existen algunos enfoques formales a la tolerancia a fallas. Entre ellos podemos citar el trabajo de *Arora* y *Gouda* [15], que formalizan la noción de sistema tolerante a fallas. Otro enfoque interesante es el de *Magee* y *Maibaum* [98], quienes proponen adoptar una semántica de sistemas basada en máquinas de estados, con diferentes tipos de estados para los sistemas, tales como estados "buenos" (normales) y "malos" (anormales), y un lenguaje para la especificación de requisitos que puede hacer uso de los distintos tipos de estados.

En el caso de la utilización de lógicas deónticas para razonar sobre tolerancia a fallas, uno de lo primeros trabajos al que podemos remontarnos es el de *Tom Maibaum* [99], publicado en el año 1987, donde el autor propuso una lógica modal con operadores deónticos para verificar propiedades sobre sistemas de computación. Estas ideas fueron tomadas y desarrolladas luego por *Samit Khosla* en su tesis de doctorado [77], defendida al año siguiente. Por su parte, *José Luiz Fiadeiro* en 1991 [55], definió operadores temporales de manera que le permitan razonar sobre especificaciones deónticas.

Pablo Castro, retomó algunas de estas ideas en su tesis de doctorado [28], la cual fue presentada en 2009 y donde propuso un framework matemático el cual utiliza teorías axiomáticas para especificar sistemas tolerantes a fallas. Para ello el autor definió dos sistemas deductivos; uno de ellos es un sistema deductivo *standard* que sigue el estilo de la lógica de *Hilbert*, en cambio el segundo es un sistema de *Tableaux*, que puede ser aplicado automáticamente para probar propiedades de las especificaciones. Algo para remarcar con respecto al uso de estas lógicas para tolerancia a fallas, es que prácticamente no existen herramientas de análisis disponibles.

Otra lógica que involucra operadores deónticos es la lógica epistémica denomina-

da *ATLK* [54, 112], la misma cuenta con un operador deóntico $O_i\phi$ cuya semántica es la siguiente: La propiedad ϕ es verdadera siempre y cuando el agente i este funcionando correctamente. Los estados *correctos* de los agentes son representados a través del color *verde*, en cambio los estados de color *rojo* representan los estados con fallas. MCMAS [96, 95] es un *model checker* que utiliza la lógica *ATLK* para verificar propiedades sobre sistemas interpretados [54], en particular se enfocan en sistemas multi-agentes.

En lo que respecta a análisis probabilístico de propiedades de sistemas, existe un gran desarrollo en el área. En los últimos años se han propuesto varias lógicas temporales que incorporan alguna noción de probabilidades con el objetivo de poder especificar y verificar propiedades sobre escenarios donde las probabilidades son necesarias, como es el caso de algoritmos *random* y protocolos distribuidos. Muchas de estas lógicas son básicamente extensiones de las lógicas temporales. Por ejemplo PCTL, es la contraparte probabilista de CTL, como PCTL* es la versión probabilista de CTL*, por nombrar alguna de ellas. En particular en lo que respecta a model checking de sistemas con probabilidades el trabajo de Kwiatkowska [81], resulta muy interesante para consultar. Herramientas como *PRISM* [63] y *LiQuor* [37] soportan *model checking* probabilístico, en particular estos model checkers permiten verificar fórmulas de PCTL y PCTL* sobre cadenas de *Márkov* como modelos.

Desde hace algunos años se viene realizando un mayor esfuerzo para intentar *standarizar* los lenguajes temporales de especificación de *hardware* ([48, 44]). Se intenta lograr un equilibrio entre la facilidad de uso del lenguaje a la hora de especificar, el poder expresivo y la complejidad del algoritmo de *model checking*. Llevando este mismo análisis al área probabilista, si analizamos las lógicas existentes PCTL es el lenguaje *standard* para razonar y verificar propiedades sobre sistemas probabilísticos, sin embargo el poder expresivo del misma es bastante limitado. Otros autores han intentado aumentar el poder expresivo de diversas lógicas, en [107, 31] y automátas [65], han definido diferentes extensiones de μ-cálculo probabilístico los cuales resultan lenguajes de especificación muy expresivos, sin embargo tienen problemas en lo que se refiere a la legibilidad de las fórmulas y facilidad de uso a la hora de especificar.

1.5. Organización

El resto de este trabajo estará organizado de la siguiente manera. En el capítulo 2 se introducen los conceptos y definiciones básicas necesarios para el desarrollo de

nuestro trabajo. En el capítulo 3 presentaremos nuestro formalismo lógico, denominado dCTL, el cual resulta adecuado para la especificación y verificación de sistemas tolerantes a fallas, combinando operadores deónticos y temporales. El algoritmo de *model checking* de dCTL es presentado en el capítulo 4. En el capítulo 5 se explora la capacidad expresiva de este nuevo formalismo lógico y su escalabilidad a través de algunos casos de estudio. Una breve introducción a sistemas probabilistas y como surgen naturalmente a la hora de especificar propiedades de sistemas tolerantes a fallas es presentada en el capítulo 6.

Por último en el capítulo 7 presentaremos trabajos relacionados del área y las conclusiones finales de nuestro trabajo, conjuntamente con algunas líneas de investigación para continuar desarrollando en un futuro.

Capítulo 2

Conceptos Preliminares

En este capítulo introduciremos algunos conceptos básicos los cuales son necesarios para el desarrollo de nuestro trabajo. Cabe aclarar que también se incluyen citas a trabajos relacionados, donde se pueden profundizar, en caso de ser necesario, muchos de los temas aquí presentados.

2.1. Estructuras de Kripke

Las estructuras de Kripke son una de las herramientas más utilizadas para la interpretación de fórmulas temporales o modales, como así también se las utiliza para caracterizar el comportamiento operacional de sistemas reactivos [39], por resultar muy intuitivas para representar dichos comportamientos . Básicamente son una variante de los sistemas de transición de estados, las mismas fueron introducidas por *Saul Kripke* en 1963 [79].

Definición 2.1 (Estructura de Kripke)**.** *Sea AP un conjunto de proposiciones atómicas. Una estructura de Kripke sobre el conjunto AP es una* 4-upla $\langle S, I, R, L\rangle$*, donde S es el conjunto de* estados*, $I \subseteq S$ el conjunto de* estados iniciales*, $R \subseteq S \times S$ es una relación de* transición *entre estados, o relación de* alcanzabilidad*, y $L : S \to 2^{AP}$ es una función de interpretación, la cual indica el conjunto de proposiciones atómicas que se satisfacen en cada estado.*

Dada una estructura de Kripke $M = \langle S, I, R, L\rangle$, la interpretación de conectivos lógicos y operadores modales puede definirse en función de L y de la relación R. En el caso de lógicas temporales, usualmente es necesario introducir la noción de *traza* para definir la semántica de algunos operadores.

Definición 2.2 (traza)**.** *Una* traza *es una secuencia maximal de estados adyacentes respecto a la relación de alcanzabilidad R. Cuando una traza comienza en un estado inicial es llamada una* ejecución *de M. Dada una traza $\sigma = s_0, s_1, s_2, s_3, \ldots$, el i-ésimo estado de σ es denotado por $\sigma[i]$, y el segmento final de σ comenzando en la posición i es denotado por $\sigma[i..]$. Finalmente, denotaremos por $\mathcal{U}_M$ el conjunto de todas las trazas, es decir, secuencias maximales de estados adyacentes, de M.*

2.1.1. Estructuras de Kripke coloreadas

A continuación definiremos una variante de las estructuras de Kripke presentadas, la cual es un punto clave en esta tesis, ya que nos ayudarán a definir la semántica de nuestra lógica.

Definición 2.3 (Estructura de Kripke coloreada)**.** *Definiremos a una* estructura de Kripke coloreada *como una* 5-upla $\langle S, I, R, L, \mathcal{N}\rangle$*, donde $\langle S, I, R, L\rangle$ es una estructura de kripke tal como la definimos en 2.1 y $\mathcal{N} \subseteq S$ es el conjunto de estados* normales *o* verdes*. Las transiciones en las cuales intervienen estados* anormales*, van a ser consideradas como* fallas *(enfoques similares a esta manera de modelar las fallas pueden encontrarse en la literatura, por ejemplo [70]). Las ejecuciones normales van a ser aquellas que sólo involucren estados* normales*. El conjunto de ejecuciones normales lo denotaremos como $\mathcal{NT}$.*

Algo importante para remarcar es que en este trabajo vamos a asumir que para cada estructura de kripke coloreada y para cada estado *normal*, existe al menos un sucesor que es también normal, y que además al menos un estado inicial también lo es. Esto garantiza que cada sistema tiene al menos una ejecución normal, es decir que $\mathcal{NT} \neq \emptyset$, lo cual es bastante razonable.

Un simple ejemplo: Protocolo para envío de mensajes

Con el objetivo de comenzar a adquirir el vocabulario que emplearemos a lo largo de esta tesis, consideremos la siguiente estructura de Kripke coloreada, la cual captura el comportamiento de un simple protocolo de comunicación, el cual funciona sobre un único canal, [24]. Inicialmente se intenta enviar un mensaje, y debido a que el canal por el cual se envían los mensajes es un medio inseguro, puede ocurrir que el mensaje enviado efectivamente llegue a destino o que se *pierda* durante el envío del mismo, lo cual representaría una falla en este simple sistema.

La estructura de Kripke coloreada, presentada en la figura 2.1, modela dicho protocolo. En el estado inicial q_0, el mensaje es creado, luego es enviado a través del

canal en el estado q_1, donde vale la proposición envío. Los estados q_2 y q_3 modelan justamente la situación en la cual el mensaje es *perdido* o *enviado* con éxito, respectivamente. En este último caso, al llegar el mensaje correctamente al destino, el sistema vuelve al estado inicial. Por otro lado, en el caso en que el mensaje se haya perdido, el protocolo reintentará infinitas veces , con el objetivo de que eventualmente el mensaje llegue a destino, situación que puede llegar a darse o no, dependiendo si las fallas dejan de ocurrir.

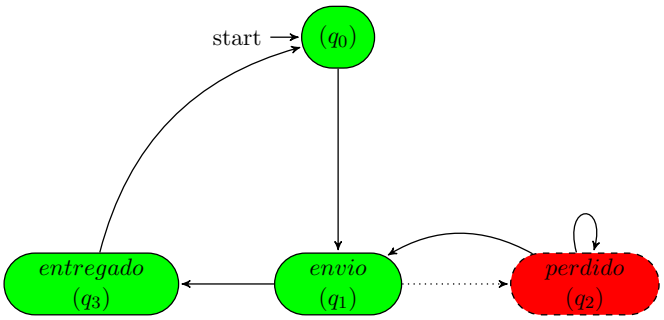


Figura 2.1: Un simple protocolo para envío de mensajes

En este pequeño ejemplo, el conjunto $\mathcal{N}$ de estados *normales* esta constituído por los estados $\{q_0, q_1, q_3\}$, en el caso del estado q_2, nos referiremos a el como estado *anormal* del sistema. Las trazas *normativas* de este protocolo son todas aquellas trazas en las cuales no intervenga el estado *anormal* q_2, en este caso la traza: $\sigma = q_0, q_1, q_3, q_0, q_1, q_3, q_0, q_1, q_3, \ldots$. En cambio ejecuciones como por ejemplo $\sigma = q_0, q_1, q_2, q_1, q_2, q_1, q_2, \ldots$ son trazas *anormales* o con *fallas*.

Cabe notar, que si bien los colores no forman parte de la definición formal de una estructura de Kripke coloreada, por una cuestión de claridad y con el objetivo de ayudar a identificar visualmente los estados normales de los estados fallidos, en algunas situaciones se utilizarán dos colores distintos para representar los mismos. En este trabajo en general utilizaremos el color *verde* para los estados *normales* y *rojo* para los estados *anormales*. Sin embargo, en aquellas situaciones donde consideremos que el uso de color no aporta a la claridad del modelo, otra convención gráfica que utilizaremos es la de dibujar con líneas punteadas los estados y arcos correspondientes a estados y transiciones anormales, respectivamente. El resto de los estados y transiciones graficados con líneas continuas representará el comportamiento *normal* del sistema.

2.2. Lógicas Temporales

Este tipo de lógicas permiten razonar sobre la ocurrencia de eventos a través del tiempo. Si bien existen muchas formas de representar el tiempo en lógica, una que ha dado buenos resultados, por la naturalidad con la cual se pueden expresar ciertos conceptos, es la representación *modal*. En esta sección introduciremos dos de las lógicas más utilizadas y que constituyen uno de los pilares base sobre el cual desarrollaremos nuestro trabajo. También incluiremos algunos ejemplos para exhibir la semántica de los principales operadores temporales, como así también mencionaremos algunas de las características más relevantes de las mismas.

2.2.1. *Linear Temporal Logic* (LTL)

Linear Temporal Logic (LTL), o en castellano *Lógica Temporal Lineal*, fue introducida en 1977 por *Amir Pnueli* en [113]; y como su nombre lo indica es una lógica en la cual el tiempo es modelado de manera *lineal*, es decir que para cada instante de tiempo existe un único posible instante futuro. Más formalmente, esto significa que las interpretaciones de las fórmulas LTL se definen en término de secuencias de estados (trazas) [24].

Respecto a la sintaxis se pueden encontrar en la bibliografía el uso de distinta simbología para denotar las modalidades temporales; en esta tesis vamos a introducir brevemente la versión más clásica de LTL, donde los operadores predican únicamente de los sucesos futuros y seguiremos la siguiente notación:

Definición 2.4 (Sintaxis)**.** *Sea* $AP = \{p_0, p_1, \dots\}$ *un conjunto de proposiciones atómicas; definiremos recursivamente al conjunto* Φ*, de fórmulas* LTL *bien formadas, a través de la siguiente gramática:*

$$\Phi ::= \top \mid p_i \mid \neg\Phi \mid \Phi \wedge \Phi \mid \mathsf{X}\Phi \mid \Phi \,\mathcal{U}\, \Phi$$

Notar que el resto de los operadores de la lógica proposicional pueden definirse en función de $\neg$ y $\wedge$; por ejemplo alguno de ellos podrían definirse de la siguiente manera:

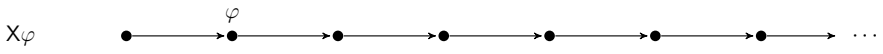
$$\begin{aligned} \Phi_1 \vee \Phi_2 &\equiv \neg(\neg\Phi_1 \wedge \neg\Phi_2) \\ \Phi_1 \rightarrow \Phi_2 &\equiv \neg\Phi_1 \vee \Phi_2 \\ \Phi_1 \leftrightarrow \Phi_2 &\equiv (\neg\Phi_1 \vee \Phi_2) \wedge (\neg\Phi_2 \vee \Phi_1) \end{aligned}$$

y lo mismo ocurre en el caso de los operadores temporales, donde a partir de X y $\mathcal{U}$ se pueden definir el resto de las modalidades temporales como F, G y $\mathcal{W}$.

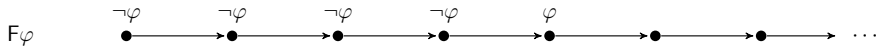
$$\begin{aligned} \mathsf{F}\Phi &\equiv \top\, \mathcal{U}\, \Phi \\ \mathsf{G}\Phi &\equiv \neg\mathsf{F}\neg\Phi \\ \Phi_1\, \mathcal{W}\, \Phi_2 &\equiv (\Phi_1\, \mathcal{U}\, \Phi_2) \vee \mathsf{G}\Phi_1 \end{aligned}$$

La semántica de $\top$, p_i, $\neg$, $\wedge$, $\vee$, $\rightarrow$ y $\leftrightarrow$ es la usual de lógica proposicional, ya que LTL no es más que una extensión de esta con las modalidades temporales. A la hora de entender la semántica de estos operadores, una buena manera es hacerlo a través de una representación gráfica; con la cual se puede capturar visualmente la intuición detrás de estas modalidades temporales:

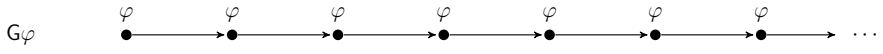
- $\mathsf{X}\varphi$, en el próximo instante vale φ.



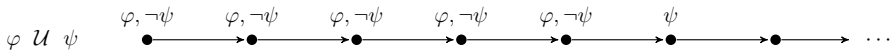
- $\mathsf{F}\varphi$, en algún instante futuro φ es verdadera.



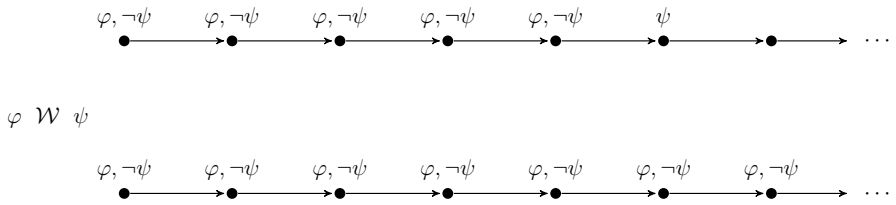
- $\mathsf{G}\varphi$, en todo instante futuro vale φ.



- $\varphi\, \mathcal{U}\, \psi$, en los instantes futuros φ es verdadera hasta que vale ψ.



- $\varphi\ \mathcal{W}\ \psi$, idem que $\varphi\ \mathcal{U}\ \psi$,con la diferencia que ψ puede que nunca se haga verdadera en los instantes futuros.



Como se puede apreciar las fórmulas LTL son evaluadas sobre *trazas* de una estructura de Kripke dada. Y se puede decir que un *estado* particular de un sistema satisface una fórmula LTL, si todas las *trazas* que comienzan desde ese estado satisfacen la misma. Es decir que LTL implícitamente cuantifica *universalmente* sobre trazas. Una definición formal y detallada de la semántica de cada operador de este formalismo lógico puede encontrarse en [24], o en bibliografía mas clásica como [50] ó [101].

Algunos patrones útiles

A la hora de especificar propiedades utilizando LTL, podemos encontrar una serie de combinaciones de operadores temporales que resultan útiles para capturar ciertos comportamientos, por ejemplo algunos patrones que aparecen frecuentemente son:

- GFφ: para todo instante futuro, eventualmente se cumple φ. Generalmente se la utiliza para expresar que una propiedad se cumple con infinita frecuencia; por ejemplo con la fórmula: GF *habilitado* podemos especificar fácilmente que cierto proceso se encuentra infinitamente habilitado a lo largo de las ejecuciones de un sistema.

- FGφ: en algún instante futuro se cumple φ y a partir de ese instante se cumple infinitamente. En este caso se expresa el hecho de que a partir de cierto momento j, se visitan únicamente estados donde se cumple la propiedad dada. Por ejemplo si queremos decir que pase lo que pase un proceso eventualmente se bloquea lo podríamos hacer con la siguiente fórmula: FG *bloqueado*.

- G($\varphi \rightarrow$ Fψ): para todo instante futuro siempre que se cumpla la propiedad φ, eventualmente se cumplirá ψ. Justamente este patrón sirve para capturar propiedades de *progreso*. Por ejemplo en el ejemplo del protocolo que vimos

anteriormente en 2.1 con la fórmula $\mathsf{G}(envio \rightarrow \mathsf{F}entregado)$, podríamos capturar el hecho de que "cada mensaje enviado, en algún momento futuro llega a ser entregado".

- $\mathsf{GF}\varphi \rightarrow \mathsf{GF}\psi$: Siempre que se cumpla la propiedad φ con infinita frecuencia, entonces también siempre eventualmente se cumplirá ψ. Justamente este patrón sirve para capturar propiedades de *strong fairness* (justicia). Continuando con el ejemplo presentado en 2.1 con la fórmula $\mathsf{GF}envio \rightarrow \mathsf{GF}entregado$, podríamos capturar el hecho de que "Si se envía un mensaje con infinita frecuencia, entonces también se cumple que será entregado infinitas veces".

Cabe destacar, que existen muchas otras variantes de LTL, por ejemplo algunos autores extienden la misma con operadores de *pasado*, los cuales permiten predicar sobre la ocurrencia de eventos en instantes de tiempo previos. Sin embargo algo curioso de estas extensiones lógicas es que si bien permiten especificar con mayor facilidad propiedades donde justamente se necesita hacer referencia a sucesos del pasado, haciendo más legibles y simples las mismas, es que en lo que se refiere al poder expresivo de la lógica, el mismo no se ve afectado; es decir han demostrado que para cualquier propiedad LTL que contenga uno o más operadores de *pasado*, con una noción de tiempo discreto, se puede encontrar una fórmula equivalente utilizando únicamente operadores LTL de *futuro*, para más detalles referirse a [88, 85].

Finalmente, respecto a la expresividad de LTL, al implícitamente cuantificar con un *universal* las trazas para cualquier propiedad dada, es lógico esperar que aquellas propiedades que involucren el cuantificador *existencial*, no puedan ser expresadas en esta lógica. Sin embargo, muchas de éstas propiedades que predican sobre la existencia de una traza en particular, pueden ser descriptas en LTL como la negación de la misma y luego interpretadas acorde esto, es decir: si desde un estado particular todas las trazas satisfacen la negación, entonces se puede asegurar que dicha propiedad no vale en dicho estado, y viceversa. Pero este enfoque no sirve para especificar aquellas propiedades que involucren ambos tipos de cuantificadores combinados y es ahí donde aparecen otro tipo de lógicas temporales, donde el tiempo se modela de manera ramificada, las cuales vienen a dar una solución a este tipo de problemas ya que justamente permiten predicar explícitamente sobre trazas.

2.2.2. *Computation Tree Logic* (CTL)

En particular aquí introduciremos, *Computation Tree Logic* (CTL) la cual es una lógica temporal donde justamente el tiempo es modelado de manera ramificada

o con forma de *árbol* (cada instante de tiempo puede tener uno o más instantes futuros), por lo que resulta adecuada para modelar las ejecuciones o *computaciones* de sistemas, de ahí su nombre. Una característica interesante de esta lógica es que permite la descripción de propiedades sobre estructuras de Kripke a través de la combinación, con ciertas restricciones, de operadores proposicionales, cuantificadores y modalidades temporales; lo cual le da un mayor poder de expresividad, ya que se pueden capturar propiedades que predican acerca del *árbol* que es construido a partir de la estructura de Kripke, comenzando desde un estado inicial y desplegando la estructura hasta formar un *árbol*, generalmente infinito.

Al igual que para LTL, de acuerdo a la bibliografía consultada varía la notación utilizada para identificar los operadores temporales, en nuestro caso continuaremos con la notación utilizada anteriormente. Veámos ahora la sintaxis formal de esta lógica:

Definición 2.5 (Sintaxis)**.** *Sea* $AP = \{p_0, p_1, \dots\}$ *un conjunto de proposiciones atómicas; definiremos recursivamente al conjunto* Φ*, de fórmulas* CTL *bien formadas, de la siguiente manera:*

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

Notar que el resto de los conectivos de la lógica proposicional pueden definirse en función de $\neg$ y $\to$; por ejemplo algunos de ellos podrían defnirse de la siguiente manera:

$$\begin{aligned} \Phi_1 \vee \Phi_2 &\equiv \neg\Phi_1 \to \Phi_2) \\ \Phi_1 \wedge \Phi_2 &\equiv \neg(\Phi_1 \vee \neg\Phi_2) \end{aligned}$$

y en el caso de los operadores temporales ocurre algo similar, ya que a partir de EX, AX, $\mathsf{E}\,\mathcal{U}$ y $\mathsf{A}\,\mathcal{U}$ pueden definirse el resto de las modalidades temporales como F, G y $\mathcal{W}$ (combinadas respectivamente con su cuantificador):

$$\begin{aligned} \mathsf{EF}\Phi &\equiv \mathsf{E}(\top\, \mathcal{U}\, \Phi) \\ \mathsf{AF}\Phi &\equiv \mathsf{A}(\top\, \mathcal{U}\, \Phi) \\ \mathsf{EG}\Phi &\equiv \neg\mathsf{AF}\neg\Phi \\ \mathsf{AG}\Phi &\equiv \neg\mathsf{EF}\neg\Phi \\ \mathsf{E}(\Phi_1\, \mathcal{W}\, \Phi_2) &\equiv \neg\mathsf{A}((\Phi_1 \wedge \neg\Phi_2)\, \mathcal{U}\, (\neg\Phi_1 \wedge \neg\Phi_2)) \\ \mathsf{A}(\Phi_1\, \mathcal{W}\, \Phi_2) &\equiv \neg\mathsf{E}((\Phi_1 \wedge \neg\Phi_2)\, \mathcal{U}\, (\neg\Phi_1 \wedge \neg\Phi_2)) \end{aligned}$$

A la hora de definir la semántica notar que existen dos tipos de fórmulas en esta lógica, aquellas que expresan una propiedad de acerca de un estado, conocidas justamente como fórmulas de *estado*,y otras que caracterizan propiedades de trazas, y a las cuales se las denomina fórmulas de *trazas* ó fórmulas de *camino*. Intuitivamente la semántica de los operadores temporales es la misma que para LTL, y para el caso de los cuantificadores, $\mathsf{A}\varphi$ se cumple en un estado, si todas las trazas, que comienzan a partir de dicho estado, satisfacen φ. En el caso de $\mathsf{EX}\varphi$ la misma es verdadera en un estado, si existe una traza a partir de dicho estado donde se cumple la propiedad.

Veamos de todas maneras una definición más formal de dicha semántica:

Definición 2.6 (Semántica)**.** *Dada una estructura de Kripke $M = \langle S, I, R, L\rangle$, y un estado $s \in S$, la semántica de una fórmula CTL está definida de la siguiente manera:*

- $M, s \models \top$
- $M, s \models p_i \Leftrightarrow p_i \in L(s)$*, donde* $p_i \in AP$*.*
- $M, s \models \neg\varphi \Leftrightarrow$ *not* $M, s \models \varphi$*.*
- $M, s \models \varphi \rightarrow \varphi' \Leftrightarrow (M, s \models \neg\varphi)$ *or* $(M, s \models \varphi')$*.*
- $M, s \models \mathsf{EX}\varphi \Leftrightarrow$ *para alguna traza* σ *tal que* $\sigma[0] = s$*,* $M, \sigma[1] \models \varphi$*.*
- $M, s \models \mathsf{AX}\varphi \Leftrightarrow$ *para toda traza* σ *tal que* $\sigma[0] = s$*,* $M, \sigma[1] \models \varphi$*.*
- $M, s \models \mathsf{E}(\varphi\, \mathcal{U}\, \varphi') \Leftrightarrow$ *para alguna traza* σ *tal que* $\sigma[0] = s$*, existe* $j \geq 0$ *el cual cumple* $M, \sigma[j] \models \varphi'$*, y para cada* $0 \leq k < j$*, vale* $M, \sigma[k] \models \varphi$*.*
- $M, s \models \mathsf{A}(\varphi\, \mathcal{U}\, \varphi') \Leftrightarrow$ *para toda traza* σ *tal que* $\sigma[0] = s$*, existe* $j \geq 0$ *el cual cumple* $M, \sigma[j] \models \varphi'$*, y para cada* $0 \leq k < j$*, vale* $M, \sigma[k] \models \varphi$*.*

Algunos patrones útiles

Algunas combinaciones que resultan útiles para caracterizar ciertos comportamientos utilizando CTL son:

- $\mathsf{AG}(\varphi \rightarrow \mathsf{AF}\psi)$: En todo instante futuro, eventualmente si se cumple φ, entonces en algún momento se cumplirá ψ. Para el caso del protocolo presentado en 2.1, $\mathsf{AG}(envio \rightarrow \mathsf{AF}\ entregado\ )$, caracterizaría el hecho de que todo mensaje enviado eventualmente llega a ser entregado.

- $\mathsf{AG}(\mathsf{AF}\varphi)$: Para toda traza futura, siempre ocurre que se cumple φ en algún momento. Este patrón sirve para expresar por ejemplo que un proceso dado esté habilitado en algún momento para cualquier ejecución del sistema.

- $\mathsf{AF}(\mathsf{AG}\varphi)$: Para toda traza futura, en algún momento se cumple φ infinitas veces. Al contrario que el caso anterior este patrón sirve para expresar por ejemplo el hecho de que ocurra lo que ocurra en algún momento dado un proceso se bloquea para cualquier ejecución del sistema, AF (AG *bloqueado*).

En [66], se pueden encontrar y profundizar sobre otros patrones que son frecuentemente utilizados en el área.

Otra aplicación que tienen muchas lógicas temporales, y en particular CTL, es su utilización en el área de *Model checking* [40, 38], lo cual la convierte en un formalismo muy atractivo para la especificación de propiedades. En la siguiente sección nos abocaremos a explicar en detalle sobre ésta técnica de verificación.

Finalmente, un aspecto interesante para analizar sobre este tipo de lógicas es su nivel expresivo y en base al mismo realizar comparaciones con otros formalismos lógicos. Existen muchos trabajos al respecto, donde se han demostrado las relaciones semánticas que existen entre cierto conjunto de lógicos temporales. En particular, se ha demostrado que la expresividad de ciertas lógicas temporales lineales y ramificadas son incomparables, esto significa que existen ciertas propiedades que pueden expresarse en lógica temporal lineal y que no pueden expresarse en lógicas temporales ramificadas; y viceversa. Esta es un área muy interesante ya que nos permite analizar y entender las características semánticas de las mismas, así como su poder expresivo. Si se desea ahondar sobre éste aspecto, un buen análisis se puede encontrar en [24].

2.3. Model Checking

Model Checking es una técnica de verificación automática que explora todos los posibles escenarios del modelo de un sistema de una manera sistemática, con el objetivo de comprobar si se cumple cierta propiedad dada. En la figura 2.2 se puede encontrar un esquema básico de este proceso de verificación, el cual dado un modelo que representa el sistema y una propiedad especificada en algún formalismo lógico, determina si la propiedad se cumple o no en el modelo dado. En el caso que la propiedad resulta falsa, se suele obtener un *contraejemplo*, el cual no es más que un escenario del modelo donde se viola dicha propiedad.

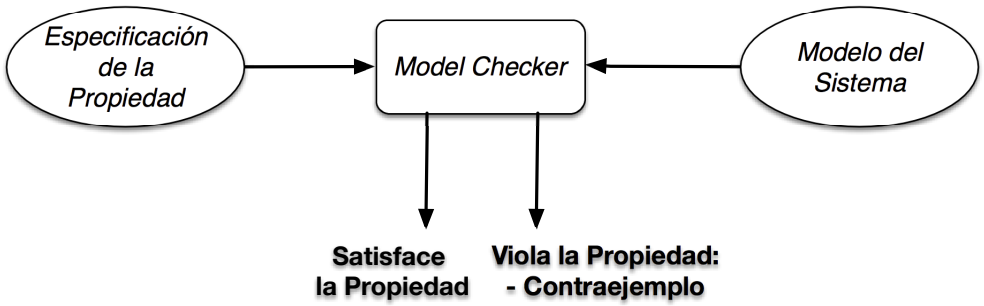


Figura 2.2: Esquema básico del proceso de *model checking*.

2.3.1. Un poco de Historia

Este enfoque formal de verificación surgió aproximadamente a comienzos de la década del ′80, motivado principalmente por la dificultad que representaba realizar las pruebas deductivas de forma manual en programas concurrentes, utilizando axiomas formales y reglas de inferencia, que originalmente estaban orientados a programas secuenciales [52]. Uno de los pioneros de ésta área fue *Amir Pnueli*, quien en 1977, remarcó las ventajas de la utilización de lógicas temporales para razonar sobre sistemas concurrentes, los cuales a menudo suelen ser representados como *sistemas reactivos*. Sistemas que "reaccionan" a eventos externos y que se caracterizan por la presencia, en muchos casos, de no determinismo, lo cual hace que la verificación de propiedades con los enfoques clásicos sea más compleja. Muchos autores consideran que los conceptos por él desarrollados fueron las cruciales para el éxito de *model checking* y sentaron las bases necesarias para los trabajos futuros del área.

Sin embargo, el término *"Model Checking"* apareció por primera vez en un artículo publicado por *Edmund M. Clarke* y *E. Allen Emerson* en 1981 [40], donde los autores introdujeron CTL y definieron un método para verificar propiedades de *fairness*, utilizando *strongly connected components*, el cual tenía complejidad cuadrática. Unos años más tarde, en 1986, dichos autores presentaron una versión mejorada del algoritmo de *model checking* para CTL, el cual resulta de complejidad lineal respecto al tamaño de la estructura de Kripke subyacente y de la fórmula que se va a verificar [38].

Existe mucha bibliografía que se puede consultar para ahondar en su estudio, sin embargo una excelente fuente para ello, es el libro *"Principles of Model Checking"*, publicado por *Christel Baier* y *Joost-Pieter Katoen* en 2008 [24], donde se puede

encontrar un estudio detallado de los diversos enfoques de ésta técnica. En particular, en dicho libro presentan la siguiente definición:

Definición 2.7 (*Model Checking*)**.** *Model Checking es una técnica automática de verificación, que dado un modelo de estados finitos de un sistema y una propiedad, chequea sistemáticamente si se cumple dicha propiedad en un estado particular del mismo.*

2.3.2. Enfoques de *Model checking*

Si bien existen diversos enfoques y técnicas que buscan optimizar ya sea el tamaño de los espacios de búsqueda como la manera en la cual se realiza la misma, con el objetivo de poder verificar sistemas reales que en en general tienen modelos de una gran tamaño, los mismos se podrían clasificar en dos grandes grupos:

Model Checking sobre modelos con estados explícitos:

Los primeros algoritmos de *model checking* estaban implementados de la manera más básica, es decir que se realizaba la exploración de todo el espacio de búsqueda para determinar si se cumplía o no la propiedad a verificar. Si bien estas técnicas de *fuerza bruta* permitieron dar los primeros pasos en el área, logrando verificar pequeños sistemas, enseguida vieron limitada su utilidad ya que a la hora de modelar programas que tenian una mayor complejidad el número de estados de los modelos crecía de manera exponencial, haciendo inviable la verificación.

Otros enfoques de *Model Checking*:

Con el objetivo de disminuir el problema de *explosión de estados*, se han desarrollado diversos métodos basados principalmente en *abstracción*, *representación simbólica* y razonamiento *composicional*. A continuación nombraremos algunos de los más divulgados:

Una característica común en todos estos métodos, es que trabajan sobre un modelo reducido del sistema original, el cual en algunos casos es obtenido mediante la aplicación de abstracciones que eliminan detalles del modelo que no son relevantes para la verificación. Es decir, que dado un modelo del sistema original M, estas técnicas permiten obtener un modelo M', equivalente al primero. Para garantizar la correctitud de los mismos, un concepto que aparece muy comunmente en la bibliografía es el de *Bisimulación*. El cual relaciona M y M', de manera tal que ambos

modelos no pueden ser distinguidos, en el sentido que cualquier comportamiento realizado en uno, puede ser imitado por el otro, y viceversa; en [24] se pueden encontrar una definición formal y las principales propiedades de este tipo de relaciones.

Otro enfoque que surgió principalmente de la aplicación de ésta técnica de verificación en componentes de *hardware*, es el denominado *model checking simbólico*, el cual trabaja sobre una representacón simbólica de los estados y las transiciones de un sistema. Principalmente utilizan *Diagramas de Decisión Binarios*, ordenados y reducidos, en inglés denominados *Reduced Ordered Binary Decision Diagrams (ROBDDs)* o simplemente *BDD*. Un *BDD* es una representación de una función lógica a través de un autómata finito determinístico acíclico con raíz. Luego mediante una cadena de bits $b_0, b_1, b_2, ...$ codifican aquellos estados o transiciones de un programa dado. Intuitivamente dicho autómata acepta sólo aquellas cadenas de bits de los estados o transiciones del programa en cuestión. Notar que un *BDD* con un número polinomial de estados, puede tener un número exponencial de *caminos*, por lo que puede representar un gran número de estados o transiciones. La aplicación de los algoritmos de *model checking* tradicionales conjuntamente con esta manera de representación simbólica ha permitido la verificación de sistemas con cientos de variables, de hecho se ha llegado a verificar diseños de *hardware* con hasta 10^{90} estados. Sin embargo, cabe aclarar que no todo sistema cuyo BDD tenga esa cantidad gigantesca de estados podrá ser siempre verificado, ya que en esos casos hay menos chances de poder encontrar una manera eficaz de representar tantos estados e innevitablemente la búsqueda se hace inmanejable. Para más información se puede consultar [52, 26, 24].

Por último, otra técnica que ha sido utilizada es la de *model checking parametrizado* realizando la verificación de sistemas de tamaño k o acotando el espacio de búsqueda hasta profundidad k, donde k es un entero positivo. Cualquiera de estas dos variantes tienen en común que permiten atacar el problema de verificar sistemas con un número infinito de estados, ya que con este tipo de técnicas de restricción se pueden encontrar soluciones a algunos problemas que si se los ataca globalmente resultarían indecidibles.

2.3.3. Ventajas y Desventajas

Algunos autores [24, 52], destacan los siguientes puntos a favor y en contra de esta técnica de verificación de sistemas:

- **Ventajas:**

1. Una de las principales ventajas de este enfoque es que sea *automático*, ya que el uso de la verificación del modelo no requiere una gran interacción del usuario ni un alto grado de experiencia en demostraciones lógicas tradicionales.
2. La obtención de contraejemplos, en el caso de que la propiedad a verificar no se cumpla, permite en muchos casos descubrir errores en etapas tempranas del desarrollo de sistemas, disminuyendo los costos de encontrar errores en etapas futuras.
3. Se la considera una de las técnicas más eficientes a la hora de encontrar futuros errores en el diseño, junto con *simulación* y *testing*.
4. Tiene una base matemática sólida que respalda las técnicas utilizadas en muchos de los algoritmos (teoría de grafos, lógica y algebraicas), lo cual garantiza su correctitud.
5. Uno de los grandes logros que tiene *model checking* es que su aplicación no ha quedado únicamente en el ámbito académico, sino que ha sido adoptado por grandes compañías, principalmente de *hardware*, las cuales han verificado muchos de sus productos en sus propios laboratorios utilizando diferentes *model checkers*.

- **Desventajas:**

1. La explosión del número de estados sigue siendo el *talón de aquiles* de esta técnica. Ya que si bien en la actualidad de pueden verificar sistemas con millones de estados, a través de diversas técnicas que atacan este problema, en muchos casos cuando se modelan sistemas reales, los modelos siguen siendo demasiado grandes para los algoritmos, que rápidamente agotan la memoria de la computadoras en la cual se los ejecuta.
2. La verificación de sistemas con un número infinito de estados o de propiedades que predican acerca de tipos abstractos de datos, en donde se requiere el uso de lógicas semi-decidibles o indecidibles, hace que la aplicación de *model checking* muchas veces no pueda ser efectivamente computado.
3. Debido a que la verificación es realizada sobre un *modelo* y no sobre el sistema *real*, muchos autores afirman que la correctitud de los resultados obtenidos depende mayoritariamente de la fidelidad con la que se haya modelado.

4. Si bien el nivel de *experticia* necesaria es relativamente menor al requerido en otros enfoques de verificación, la obtención de modelos precisos y la especificación de las propiedades que se desean verificar, depende en gran parte del conocimiento y habilidad que tenga el usuario para formalizar los mismos.

2.4. Lógicas Deónticas

La Lógica Deóntica es una rama de lógica que permite razonar acerca del comportamiento ideal o esperado, y distinguirlo de aquellos que no lo son. Dentro de ésta área, las lógicas modales juegan un rol importante, ya que muchas lógicas deónticas han sido definidas a través de operadores modales, interpretando cada modalidad con conceptos como *prohibición*, *obligación*, *permiso*, entre otros; ya que justamente surgieron para estudiar y caracterizar razonamientos en áreas jurídicas y filosóficas, donde estas nociones aparecen con mucha frecuencia.

Algunas nociones que aparecen comúnmente en este tipo de lógicas son:

- *Obligación*: Generalemente denotado con $\mathbf{O}(\varphi)$ o $\Box(\varphi)$ ya que algunos autores prefieren utilizar los operadores modales para representar dicha noción. En este caso $\mathbf{O}(\varphi)$ intuititivamente significa que es mandatorio que se cumpla φ.

- *Permiso*: $\mathbf{P}(\varphi)$ o $\Diamond(\varphi)$ es la notación más frecuente para denotar este concepto, en el cual se representa el hecho de que algo está permitido de ocurrir. En la mayoría de las lógicas deónticas este operador es definido como el dual de la obligación.

- *Prohibición*: Clásicamente representado como $\mathbf{F}(\varphi)$ (*Forbidden*), el cual captura el hecho de que φ se encuenta prohibido.

2.4.1. Un poco de historia

Aunque el concepto de *Obligación*, actualmente aparece en casi todas las lógicas deónticas, fue el filósofo austríaco *Ernst Mally* en 1926, el primero que definió formalmente esta noción, como así también un sistema axiomático para ésta lógica; y si bien dicha lógica tenía varios detalles técnicos, la misma motivó la aparición de muchas otras extensiones deónticas, por lo cual se lo considera como uno de los grandes pioneros de ésta área [100].

Otros autores que han contribuído al desarrollo de éstas lógicas en sus comienzos son los abogados *Carlos Alchourrón* (Argentino) y *Eugenio Bulygin* (Ucraniano), quienes se destacaron por su aporte en teoría del derecho a través del uso de este tipo de lógicas para la formalización del razonamiento jurídico. Algunos autores consideran que el trabajo de estos dos investigadores marcó la necesidad de distinguir entre una lógica de proposiciones normativas y una lógica de normas (Deóntica), dando el puntapié inicial para el estudio formal de las mismas [8].

Por otro lado, unos años antes, el filosósofo finlandés *Georg Henrik von Wright* abrió varias líneas de investigación que marcaron muchos de los avances obtenidos en ésta rama de la ciencia. Es por ello que muchos lo consideran el principal impulsor de la lógica deóntica como un área de investigación activa. Uno de sus trabajos más conocidos fue la definición formal, en 1951, de un sistema deóntico [124], el cual sirvió de base para el desarrollo de SDL (*Standard Deontic Logic*), lógica en la cual se le dió una interpretación a modal través de estructuras de *Kripke* [68, 76, 34]. SDL es una de las lógicas deónticas más estudiadas y conocidas dentro de la comunidad y se la considera el primer intento serio de capturar el razonamiento deóntico. A continuación presentaremos algunas de la principales características de la misma.

2.4.2. *Standard Deontic Logic* (SDL)

Si agregamos a la lógica *proposicional* clásica los siguientes axiomas, obtenemos un sistema axiomático básico para SDL:

- **A1**: $\mathbf{O}(\varphi \to \psi) \to (\mathbf{O}(\varphi) \to \mathbf{O}(\psi))$, el cual captura el hecho de que si es obligatorio que la verdad de φ implica a ψ, entonces si se cumple obligadamente φ implica que también está obligado ψ.
- **A2**: $\mathbf{O}(\varphi) \to \neg\mathbf{O}(\neg\varphi)$, el cual nos dice que φ es obligatorio siempre y cuando su negación no lo sea.

El concepto de *Prohibición*, $\mathbf{F}(\varphi)$ (se encuentra prohibido φ), puede expresarse a través del operador de obligación de la siguiente manera: $\mathbf{O}(\neg\varphi)$. Por otro lado, para representar *Permiso* lo definen como el dual de la obligación: $\mathbf{P}(\varphi) \leftrightarrow \neg(\mathbf{O}(\varphi))$, de lo cual se deduce, conjuntamente con el axioma **A2**, que $\mathbf{O}(\varphi) \to \mathbf{P}(\varphi)$ (Si está *obligado*, está *permitido*).

Por otro lado también se cumplen las siguientes reglas en SDL:

- *Modus Ponens*: Si $\vdash \varphi$ y $\vdash \varphi \rightarrow \psi$, entonces $\vdash \psi$. Si se cumple una implicación y su antecedente, entonces también es verdad su consecuente.
- *Ob-NEC*: Si $\vdash \varphi$, entonces $\vdash \mathbf{O}(\varphi)$, en el cual se establece que si tiene cierto teorema φ, entonces su obligación también es un teorema en dicha lógica.

Respecto a su semántica, la misma está dada a través de estructuras de *Kripke* definidas de la forma $K = \langle W, A \rangle$, donde W es el conjunto de estados ("mundos"), en los cuales las fórmulas son interpretadas. Y A es un subconjunto de $A \subseteq W \times W$, que cumple con lo siguiente: $\forall i \exists j A_{ij}$. A_{ij} denota el hecho que el estado j es sucesor de i, es decir que en estas estructuras cada estado tiene un sucesor. Por otro lado, se define $M = \langle K, V \rangle$ como un modelo para dicha estructura de Kripke, donde V es una función de valuación que asigna a cada variable proposicional el conjunto de estados donde la misma resulta verdadera. Finalmente la relación de *satisfactibilidad*, $M \vDash_i p$, que representa el hecho de que p es verdarero en el estado i del modelo M. Una definición un poco más formal que puede encontrarse en la bibliografía es la siguiente:

Definición 2.8 (Semántica)**.** *Dada una estructura de Kripke $K = \langle W, A \rangle$, y un modelo $M = \langle K, V \rangle$, la semántica de una fórmula* SDL *está definida :*

- *Los conectivos clásicos de lógica proposicional tienen la semántica estándar.*
- $M \vDash_i \mathbf{O}(\varphi) \Leftrightarrow (\forall j :$ *Si* A_{ij}*, entonces* $M \vDash_j \varphi)$.
- $M \vDash_i \mathbf{P}(\varphi) \Leftrightarrow (\exists j :$ *Si* A_{ij}*, entonces* $M \vDash_j \varphi)$.

Notar que esta definición de *obligación* va en contra de algunas propiedades intuitivas del concepto de obligación. Se puede encontrar más información acerca de estas discusiones en [34].

Como se puede apreciar SDL es un sistema relativamente simple, por ello es que algunos autores afirman que la misma presenta algunas carencias a la hora de capturar algunos de los conceptos básicos de la *obligación* [72], en particular apoyan dichas afirmaciones en el hecho que las siguientes sentencias, que son consideradas *paradojas*, son ciertas en SDL:

Ross's Paradox	:	$\mathbf{O}(\varphi) \to \mathbf{O}(\varphi \vee \psi)$
Good Samaritan Paradox	:	$\mathbf{O}(\varphi \wedge \psi) \to \mathbf{O}(\psi)$
Chisholm's Paradox	:	1. $\mathbf{O}(\varphi)$
		2. $\mathbf{O}(\varphi \to \psi)$
		3. $\neg\varphi \to \mathbf{O}(\neg\psi)$
		4. $\neg\varphi$
Gentle Killer Paradox	:	1. $\mathbf{O}(\neg\varphi)$
		2. $\varphi \to \mathbf{O}(\psi)$
		3. φ

Se denominan paradojas ya que es fácil encontrar enunciados para las mismas que resulten claramente contradictorios, por ejemplo algunos podrían ser los siguientes:

- *Ross's Paradox*:

  - Si una persona está obligada a comprar un auto, entonces está obligada a comprar el auto o robarlo.

  Es decir que la obligación de comprar un auto implica la obligación de que lo compra o lo roba, algo que se conoce que está socialmente prohibido.

- *Good Samaritan Paradox*:

  - Si un guardavidas está obligado a ayudar a un niño que se está ahogando, entonces es una obligación que el niño se esté ahogando.

  En este caso ya en lenguaje natural también suena inconsistente la afirmación hecha.

- *Chisholm's Paradox*:

  1. Es una obligación que A vote en las elecciones presidenciales.
  2. Es una obligación si A vota, que el estado mantenga el secreto de votación.
  3. Pero si A no vota, entonces no es obligación que el estado mantenga el secreto de votación.
  4. A no vota.

Analicemos de donde surge la contradicción en este caso, ya que intuitivamente en lenguaje natural no pareciera ser inconsistente. Por el axioma **A1** y 2, obtenemos que $\mathbf{O}(\varphi) \rightarrow \mathbf{O}(\psi)$, y por *Modus Ponens* con 1 tenemos que $\mathbf{O}(\psi)$. Por otro lado, aplicando también *Modus Ponens* entre 3 y 4, obtenemos que $\mathbf{O}(\neg\psi)$, es decir que el estado está obligado a mantener el secreto de votación y está obligado a no mantenerlo($\mathbf{O}(\psi) \wedge \mathbf{O}(\neg\psi)$). Lo cual, por el cálculo proposicional, es equivalente a $\neg(\mathbf{O}(\psi) \rightarrow \neg\mathbf{O}(\neg\psi))$, hecho que contradice el axioma **A2** de SDL. Este tipo de razomientos aparecen con bastante frecuencia en el mundo real, donde la violación de una obligación provoca la aparición de una obligación secundaria. Por lo que cualquier lógica que se desee utilizar para especificar y verificar propiedades de sistemas tolerantes a fallas en general, debería poder lidiar con este tipo de inconsistencias lógicas. Ya que, como hemos dicho antes, la ocurrencia de una falla en sistemas de software donde se requiere que se cumplan ciertas condiciones para restablecer o revertir la misma, puede relacionarse con la violación de una obligación y la aparición de nuevas obligaciones en el sistema.

- *Gentle Killer Paradox*:

  1. Es una obligación que *Juan* no mate a su madre.
  2. Si *Juan* mata a su madre, esta obligado a hacerlo *amablemente*.
  3. Juan mata a su madre.

  En este caso de las premisas 2 y 3, aplicando *Modus Ponens* se deduce:

  4. $\mathbf{O}(\psi)$, es decir *"Juan está obligado a matarla amablemente"*.

  Pero también sabemos que es necesario que:

  5. *"Si Juan mata a su madre amablemente, entonces de hecho es cierto que Juan mata a su madre"*, es decir que se cumple que $\psi \rightarrow \varphi$.

  Luego por *OB-RM* ($(\psi \rightarrow \varphi) \rightarrow (\mathbf{O}(\psi) \rightarrow \mathbf{O}(\varphi))$, la cual se puede derivar del conjunto básico de axiomas de SDL), tenemos que se cumple:

  6. $\mathbf{O}(\psi) \rightarrow \mathbf{O}(\varphi)$

  Nuevamente por *Modus Ponens* entre 6 y 4, obtenemos que se cumple:

7. $\mathbf{O}(\varphi)$, es decir *"Es una obligación que Juan mate a su madre"*

Por el axioma **A2** tenemos que 8. $\neg\mathbf{O}(\neg\varphi)$. Y finalmente entre 1 y 8 tenemos una contradiccón ($\mathbf{O}(\varphi) \wedge \neg\mathbf{O}(\neg\varphi)$). Lo cual, en lenguaje natural, ya podía apreciarse.

Notar que algunos autores, dependiendo del sistema de axiomas que utilizan, han sugerido que la paradoja *Gentle Killer Paradox* se trata de un problema derivado de dificultades de alcance [120], otros han argumentado que el problema es que la regla *OB-RM* es de hecho inválida, y rechazarla resuelve el problema [61].

Existen muchas otras paradojas que se han estudiado, se puede encontrar una detallada revisión de algunas de ellas en [106, 3, 110, 114]. Cabe aclarar que dichas paradojas no son exclusivas a SDL, sino que en muchas otras lógicas deónticas también están presentes este tipo de enunciados que van en contra de nuestra intuición (denominados en la comunidad *contrary-to-duty*). Por lo que han sido ampliamente estudiadas en el ámbito deóntico, motivando la aparición de nuevos formalismos lógicos que intentan solucionar este tipo de inconsistencias.

2.4.3. Lógicas Deónticas y sus aplicaciones en Cs. de la Computación

Respecto a las aplicaciones prácticas de estas lógicas, como dijimos, tradicionalmente fueron utilizadas para representar y analizar la estructura lógica de *normas* o *leyes*, y por ejemplo encontrar ambiguedades, contradicciones o dependencias entre ellas, pero sobre todo para poder caracterizar nociones como *violación* que con el enfoque de otras lógicas, como la proposicional [9], no podía hacerse [71]. De hecho, las primeras aplicaciones de lógicas deónticas en ciencias de la computación fueron justamente en la automatización de mecanismos que permitieran la verificación y análisis de normas jurídicas [47, 73].

Sin embargo, desde ya hace varios años se han comenzado a utilizar las mismas en otras áreas de ciencias de la computación, tal como inteligencia artificial, seguridad y tolerancia a fallas, se puede encontrar un completo resumen de trabajos relacionados a estas áreas en [125]. En particular en este trabajo nos centraremos en el uso de las lógicas deónticas para sistemas tolerantes a fallas, gracias a la capacidad de poder distinguir entre los comportamientos normales y anormales de los sistemas. De hecho en [125], consideran que las lógicas deónticas son adecuadas para la especificación

de propiedades de tolerancia a fallas, debido a la relación que se puede establecer entre la ocurrencia de fallas en sistemas de software y el concepto de *violación* a una norma en áreas legales o jurídicas.

Otra característica interesante de este tipo de lógicas es que pueden combinarse con otras lógicas temporales de una manera relativamente sencilla, logrando incorporar muchas de las ventajas de este tipo de lógicas para la especificación y verificación de sistemas. Chellas en [34], refiriendose a SDL y sus extensiones, remarcó la dependencia que tiene el operador de obligación respecto al tiempo. A su vez *Lennart Åqvist* también expresó la fuerte necesidad de incorporar a la semántica deóntica básica operadores temporales explícitos [12].

2.5. Tipos de Tolerancia a Fallas

En [59], *Felix C. Gärtner* presenta cuatro tipos de tolerancia a fallas, de acuerdo a las propiedades que se preservan en un sistema al momento de ocurrir una falla. El analisis está basado en dos tipos de propiedades, *Safety* y *Liveness*, las cuales son propiedades que se refieren al comportamiento de un sistema respecto al tiempo. En particular, las propiedades de *Safety* en general son caracterizadas como "Nada malo va a ocurrir en el futuro", es decir que podrían considerarse como una especie de invariante del sistema con el cual capturamos una situación que no queremos que ocurra, algo indeseable. Un claro ejemplo de una propiedad de *safety*, clásico en la bibliografía a la hora de explicar este concepto, es el de *exclusión mutua* donde se requiere que "dos procesos no se encuentren al mismo momento dentro de la región crítica", que es justamente un comportamiento que se desea evitar que ocurra.

Por otro lado, las propiedades de *liveness* capturan aquellas situaciones desables respecto al comportamiento futuro del sistema, se las suele caracterizar con la frase "Algo bueno eventualmente va a ocurrir". Siguiendo con el ejemplo de *exclusión mutua*, el hecho de que "cada proceso eventualmente ingresará a su región crítica" es una propiedad de *liveness*. Notar que los "buenos eventos" de una propiedad de *liveness* están relacionados con el comportamiento infinito del sistema, mientras que en el caso de las propiedades de *safety* la presencia de un "mal evento" se puede corroborar en un comportamiento finito del mismo. Esto es algo importante a tener en cuenta a la hora de verificar este tipo de propiedades de tolerancia a falla, en los siguientes capítulos ahondaremos en las técnicas que se utilizan para llevar a cabo esta tarea.

En la Tabla 2.1 se presenta gráficamente los distintos tipos o niveles de tolerancia

	Liveness	***¬Liveness***
Safety	*Masking*	*Fail-Safe*
¬Safety	*Non-Masking*	Sin Tolerancia a Fallas

Cuadro 2.1: Tipos de tolerancia de acuerdo a las propiedades que se preservan

a falla y su relación con las propiedades que se preservan ante la presencia de una falla. A continuación analizaremos las características de las mismas:

- ***Masking Fault Tolerant*** **(Safety + Liveness)**: Se lo podría considerar como el tipo de tolerancia a fallas "ideal", a pesar de que también suele ser el más costoso de implementar, ya que el sistema logra *enmascarar* la presencia de fallas, haciendo imperceptible la anomalía para el resto del sistema o los usuarios del mismo. Que se preserven las propiedades de *safety* y *liveness* se refiere al hecho de que el programa, al momento de manifestarse el defecto, nunca llega a estados *indeseables* o *incorrectos* (donde se viola la propiedad de *safety*) y en caso de quedar en un estado *anormal*, se garantiza que eventualmente retornará al conjunto de estados *normales* del sistema. En general muchos de los sistemas que son *Masking Fault Tolerant* lo logran mediante el uso de redundancia y la implementación de métodos de decisión, como *voting* o *consenso*.

- ***Non-Masking Fault tolerant*** **(Liveness)**: Los sistemas con este tipo de tolerancia garantizan que luego de la ocurrencia de una o más fallas y si las mismas dejan en algún momento de ocurrir, eventualmente el sistema retornará a un estado *normal* del mismo (*liveness*). Por ejemplo un sistema eventualmente terminará o un mensaje enviado mediante cierto protocolo de comunicación eventualmente llegará a su destinatario.

  Sin embargo, puede que en los momentos en que se manifiesta la falla, se vea afectado el comportamiento correcto del sistema, dejando en evidencia la ocurrencia de la misma para el resto de los componentes del sistema o los usuarios con los cuales interactuan. Es decir que no se garantiza *safety*. A pesar de ello el potencial de este tipo de tolerancia recae en el hecho de que en muchos casos la implementación de otras técnicas de tolerancia, por ejemplo *Masking*, resultarían muy costosas o inclusive imposibles de llevar a cabo por las limitaciones técnicas o características propias del sistema.

- ***Failsafe Fault Tolerant*** **(Safety)**: En este caso se preserva las propiedades

de *safety*, por lo cual los sistemas que implementen este tipo de tolerancia garantizan que al momento de producirse una falla, el sistema se mantiene *estable* dentro de un conjunto de estados donde se cumplen las condiciones mínimas y necesarias de seguridad. Los sistemas bancarios en general y en particular los cajeros atomáticos suelen estar diseñados con este tipo de tolerancia. Ya que si bien en ciertos momentos no funcionan como el usuario lo espera por ejemplo evitando que retire dinero, o que realice una transferencia debido a una falla en la comunicación con el sistema central, dichos impedimentos tienen el objetivo de resguardar el capital de los usuarios, garantizando así la integridad y seguridad de sus cuentas. Se pueden encontrar en la práctica muchos otros sistemas que implementan tolerancia *failsafe*, ya que de alguna manera en muchos casos es primordial garantizar las propiedades de *safety* y no así *liveness* pudiendo tolerar cierto detrimento en el comportamiento esperado, pero dejando al sistema en un estado seguro.

- ***Sin Tolerancia a Fallas***: Es el caso extremo, donde no se preservan las propiedades de *safety* ni de *liveness*, por lo que ante la presencia de fallas los sistemas no garantizan ningún grado de tolerancia, pudiendo o no brindar el comportamiendo esperado.

Notar que un sistema puede implementar más de un tipo de tolerancia a la vez, de acuerdo a sus requerimientos y características. Un buen ejemplo es el sistema de control de lanzamiento del cohete *Ariane 5*, en el cual se utilizó *masking* para tolerar la falla de un componente, y en caso de que dos o más componentes fallaran sucesivamente, se diseñó un mecanismo de *failsafe* para asegurar que el cohete quede en un estado seguro, frenando su lanzamiento y previniendo una posible catástrofe. Ya que estos cohetes han sufrido varias fallas en el pasado. La explosión que se produjo el 4 de junio de 1996 del *Ariane 5* (vuelo 501) segundos después de su despegue figura en la lista de los peores errores de *software* de la historia. Dicha falla fue causada por un *bug* en una rutina aritmética del controlador de vuelo, donde por reutilizar parte del código de su predecesor, *Ariane 4*, el cual utilizaba otro sistema numérico se produzco un error de *overflow*, causando que la computadora que controla el vuelo entrara en un estado de *failure* y provocando la consecuente desintegración del cohete. Errores como este, han impulsado el desarrollo de formalismos y mecanismos que permitan garantizar la correctitud de un sistema.

2.6. Sistemas Probabilistas

A lo largo de este capítulo hemos presentado brevemente diferentes formalismos que son utilizados para verificar automáticamente propiedades de sistemas. En particular, *model checking*, se enfoca en determinar de manera absoluta si se cumple o no cierta propiedad, pero a la hora verificar sistemas reales se hace muy difícil o casi imposible trasladar esta garantía absoluta, ya que naturalmente intervienen situaciones de características *estocásticas* (por ejemplo: fallas de componentes, pérdida de mensajes, etc) y es allí donde aparece el concepto de probabilidad para capturar este tipo de situaciones. En esta sección introduciremos algunos conceptos básicos necesarios para realizar la verificación de sistemas probabilistas, si se desea profundizar en ellos se puede encontrar una completa descripción de los mismos y de otros formalismos que se utilizan a la hora de verificar este tipo de sistemas, en el capítulo 10 del libro "*Principles of Model Checking*", de *Christel Baier* y *Joost-Pieter Katoen* [24].

A la hora de capturar estas situaciones o eventos de naturaleza *estocásticas* es necesario enriquecer los sistemas de transición con probabilidades. Las *Cadenas de Markov* y los *Procesos de Decisión de Markov* son dos modelos probabilísticos muy utilizados para este fin. Ambos modelos reciben su nombre en honor a su creador, el matemático ruso *Andréi Márkov*. En particular en esta tesis hacemos uso de los primeros, para interpretar las fórmulas de nuestro formalismo lógico.

2.6.1. Cadenas de Márkov

Las *Cadenas de Márkov* o *modelos de Márkov* son procesos estocásticos discretos que se caracterizan porque la probabilidad de la ocurrencia de un evento depende solamente del evento inmediatamente anterior, esta característica de falta de memoria recibe el nombre de propiedad de *Márkov*.

Formalmente podemos definir una *cadena de Márkov* sobre un conjunto AP de proposiciones atómicas como una tupla $\langle S, P, L, s_0 \rangle$, donde S es un conjunto finito de estados, $P : S \times S \to [0, 1]$ es una matríz de probabilidades, $L : S \to 2^{AP}$ es la función de etiquetado y $s_0 \in S$ es un estado inicial. Para un estado $s \in S$ denotamos M_s a la *cadena de Márkov* obtenida por M con el estado inicial s.

En la figura 2.3 se presenta un modelo que consta de 2 estados, $S = \{s_0, s_1\}$. La proposición atómica p vale únicamente en el estado s_0, es decir $L = \{(s_0, p), (s_1, \neg p)\}$. El estado inicial de la cadena de *márkov* es el estado s_0, cabe destacar que para modelar sistemas con más de un estado inicial se suele definir un vector con una

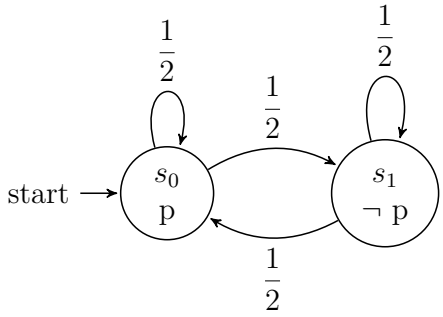


Figura 2.3: Ejemplo de una cadena de *Márkov*

distribución inicial, de la forma $init = S \to [0,1]$ donde $\sum_{s \in S} init(s) = 1$. La matriz de probabilidades queda definida de la siguiente manera:

$$P = \begin{bmatrix} 1/2 & 1/2 \\ 1/2 & 1/2 \end{bmatrix}$$

2.6.2. Una Lógica Temporal con probabilidades: PCTL

PCTL es una lógica temporal ramificada con probabilidades, la misma est á basada en CTL, con la diferencia que elimina los cuantificadores E y A, e incorpora un operador probabilístico $\mathcal{P}_J(\Psi)$, donde Ψ es una fórmula de *camino*, J es un valor real definido en el intervalo $[0,1]$, el cual representa una medida de probabilidad. Las fórmulas PCTL establecen o describen propiedades sobre los estados de cadenas de *Márkov* finitas, y son interpretadas sobre las mismas, para determinar si vale o no dicha fórmula.

- $(\mathbf{CTL}_{-\{\mathsf{E},\mathsf{A}\}})$ + $\mathcal{P}_J(\Psi)$ (Ψ es una fórmula de *camino*).

- Permite medir de manera cuantitativa propiedades sobre trazas.

Las fórmulas se definen sobre un conjunto AP de la siguiente manera:

$$\begin{array}{lcl} J & ::= & \{>, \geq\} \times [0,1] \\ \Phi & ::= & \top \mid \bot \mid p_i \mid \neg p_i \mid \Phi_1 \vee \Phi_2 \mid \Phi_1 \wedge \Phi_2 \mid \mathcal{P}_J(\Psi) \\ \Psi & ::= & \mathsf{X}\Phi \mid \Phi \,\mathcal{U}\, \Phi \mid \Phi \,\mathcal{W}\, \Phi \end{array}$$

Si analizamos el modelo de la figura 2.3, podríamos definir por ejemplo las siguientes propiedades:

- $\mathcal{P}_{\geq \frac{1}{2}}(\mathsf{X}p)$
- $p \rightarrow \mathcal{P}_{\geq 1}(\mathsf{F}(\neg p))$

Donde la primera propiedad expresa que *hay* 0,5 *de probabilidad que en el siguiente estado valga* p. En cambio, en la segunda fórmula se encuentra caracterizada la situación de que: *si vale* p*, entonces con probabilidad igual a* 1 *eventualmente valdrá* $\neg p$.

A lo largo del Capítulo 6 trabajaremos con propiedades que caracterizan patrones que se repiten indefinidamente a lo largo del tiempo, y de hecho probaremos que es posible demostrar que muchas de estas propiedades no se pueden expresar en CTL ni en PCTL. Debido a que este tipo de propiedades capturan patrones que ocurren sobre conjuntos cuya medida de probabilidad es 0.

2.6.3. μPCTL

La lógica μPCTL extiende a la lógica PCTL adicionándole variables de punto fijo y operadores de mínimo y máximo punto fijo, en el artículo [31], se puede encontrar la definición completa de la misma.

Ahora describiremos brevemente la sintaxis y semántica de los operadores de μPCTL: Sea AP un conjunto de proposiciones atómicas $\{p_0, p_1, \dots\}$ y sea $\mathcal{V} = \{V_0, V_1, V_2, \dots\}$ un conjunto enumerable de variables; los conjuntos Φ y Ψ de fórmulas de estado y fórmulas de camino, respectivamente, son definidas recursivamente de la siguiente manera:

$$\begin{array}{lcl} J & ::= & \{>, \geq\} \times [0,1] \\ \Phi & ::= & \top \mid \bot \mid p_i \mid \neg p_i \mid V_i \mid \Phi_1 \vee \Phi_2 \mid \Phi_1 \wedge \Phi_2 \mid \mathcal{P}_J(\Psi) \mid \nu V_i.\Phi \mid \mu V_i.\Phi \\ \Psi & ::= & \mathsf{X}\Phi \mid \Phi \,\mathcal{U}\, \Phi \mid \Phi \,\mathcal{W}\, \Phi \end{array} \tag{2.1}$$

Como es usual nosotros introducimos los operadores F y G. La semántica e intuiciones de las fórmulas PCTL son las usuales, si se desea se puede consultar [24].

Otro punto importante a tener en cuenta es que asumimos que en cada fórmula no se repiten las variables *ligadas*; es fácil de ver que se puede reescribir cada fórmula de manera de satisfacer este requerimiento. En general estamos interesados en fórmulas

en las cuales todas sus variables se encuentran bajo el alcanze de algún operador, es decir fórmulas que no tienen variables *libres*.

Una fórmula μPCTL caracteriza un conjunto de estados de una cadena de Markov en los cuales se satisface dicha fórmula. Considera una cadena de Márkov $M = \langle S, P, L, s_0\rangle$, la semántica de las subfórmulas podría depender de la valuación que asocia un conjunto de estados con cada variable que aparece en la misma.

Formalmente, una valuación es una función $\tau : \mathcal{V} \to 2^S$. Denotamos por $\tau[S'/V]$ la valuación tal que $\tau(V) = S'$ y por cada $V' \neq V$ tenemos $\tau[S'/V](V') = \tau(V)$.

La semántica de una fórmula φ, denotada $[\varphi]^M_\tau$ la definimos de la siguiente manera:

$$\begin{aligned}
[p_i]^M_\tau &= L(p_i)\\
[\neg p_i]^M_\tau &= S \setminus L(p_i)\\
[V_i]^M_\tau &= \tau(V_i)\\
[\varphi_1 \wedge \varphi_2]^M_\tau &= [\varphi_1]^M_\tau \cap [\varphi_2]^M_\tau\\
[\varphi_1 \vee \varphi_2]^M_\tau &= [\varphi_1]^M_\tau \cup [\varphi_2]^M_\tau\\
[\mathcal{P}_J(\Psi)]^M_\tau &= \{s \in S \mid measure_M(s, \Psi)J\}\\
[\nu V_i.\Phi]^M_\tau &= \mathsf{gfp}\{S' \subseteq S \mid S' = [\Phi]^M_{\tau[S'/V_i]}\}\\
[\mu V_i.\Phi]^M_\tau &= \mathsf{lfp}\{S' \subseteq S \mid S' = [\Phi]^M_{\tau[S'/V_i]}\}
\end{aligned}$$

Notar que 2^S es un retículo y que todos los operadores son *monotónicos*. Gracias al teorema de *Knaster-Tarski* podemos asegurar que existen el máximo y mínimo punto fijo.

2.6.4. p-Autómatas: Una nueva clase de Autómatas probabilísticos.

Existe un tipo de autómata, denominado *p-Autómata*, los cuales aceptan lenguajes de *cadena de Márkov* ([65]).

Un p-Autómata A es una 5-upla, de la forma $\langle \Sigma, Q, \delta, \varphi^{in}, \alpha\rangle$,

- El lenguaje aceptado por A son todas aquellas cadenas de Markov, y sus bisimilares, donde se cumple la condición de aceptación $\alpha \subseteq Q$.
- $\delta : Q \times \Sigma \to B^+(Q \cup [\![Q]\!])$, es la función de transición.

La intuición atrás de la función de transición δ es la siguiente: Dado q_1 un estado del *p-Autómata*, a un estado de la cadena de Márkov y definida la función

de transición de la siguiente forma $\delta(q_1, \{a\}) = q_1 \wedge [\![q_1]\!]_{\geq \frac{1}{2}}$, la semántica de dicha transición es:

- "a" se cumple en q_1 y más de $\frac{1}{2}$ de los sucesores de los estados leídos por q_1 van a cumplir con la propiedad de que valga "a".

Este tipo de autómata permiten caracterizar subconjuntos de trazas y definir probabilidades locales sobre estos. El conjunto de lenguajes de los *p-automata* es cerrado para operaciones *Booleanas*. Otra característica que tienen este tipo de autómatas es que el lenguaje de cada *p-autómata* es cerrado bajo *bisimulación probabilística*.

En lo que se refiere a la condición de aceptación, de manera similar a lo que hacen los *tree automata* cuya condición de aceptación se encuentra definida a través de la teoría de juegos, utilizando juegos de dos jugadores (*two-player games*); en el caso particular de los *p-autómatas*, la misma se encuentra definida utilizando también juegos de dos jugadores pero su versión estocástica (*two-player stochastic games*).

Cabe aclarar que en esta tesis sólo hemos utilizado las nociones básicas de *p-Autómatas* para ayudarnos en un comienzo a definir y estudiar la semántica de nuestro formalismo lógico RPCTL. De todas maneras se puede encontrar la documentación completa y los detalles técnicos de dichos auómatas en el artículo llamado *p-Automata: New foundations for discrete-time probabilistic verification*, el cual fue publicado en 2012 por Michael Huth, Nir Piterman y Daniel Wagner ([65]).

Capítulo 3

dCTL, Una Lógica Adecuada Para Tolerancia A Fallas

En este capítulo presentaremos un nuevo formalismo lógico al cual denominamos dCTL. Mostraremos además que el mismo resulta adecuado para la especificación , y luego la verificación, de propiedades de sistemas tolerantes a fallas. Las fórmulas en esta lógica predican acerca de propiedades sobre ejecuciones de estructuras de *Kripke* coloreadas, como las definidas en el capítulo anterior, en las cuales se puede distinguir entre estados *normales* y *anormales*, como así también entre *trazas normales* y *trazas anormales*. Por otro lado, analizaremos y demostraremos algunas de las principales características de la misma, respecto a expresividad y complejidad.

3.1. Deontic Computation Tree Logic: dCTL

La lógica dCTL es una extensión de la lógica CTL con operadores deónticos $\mathbf{O}(\psi)$, $\mathbf{P}(\psi)$ y $\mathbf{R}(\psi)$ los cuales aplican sobre una cierta clase de fórmulas de *camino* ψ. La intención de estos operadores es capturar la noción de *obligación*, *permiso* y *reparación* sobre trazas, respectivamente. Intuitivamente, estos operadores tienen el siguiente significado:

- $\mathbf{O}(\psi)$: La propiedad ψ está obligada en cada estado futuro alcanzable a través de transiciones *normales* (sin fallas).
- $\mathbf{P}(\psi)$: Existe una ejecución *normal*, es decir sin la presencia de fallas, que comienza desde el estado actual y en la cual se cumple la propiedad ψ en cada instante.

- $\mathbf{R}(\psi)$: La propiedad ψ vale en cada estado *fallido* futuro, es decir aquellos estados resultantes después de la ocurrencia de una falla.

Es fácil ver que los operadores *obligación* y *permiso* nos permiten expresar propiedades que deben cumplirse en *toda*, ó *alguna*, ejecución *normal* del sistema, respectivamente.

Además de los operadores de obligación y permiso, nuestra lógica brinda un operador adicional denominado operador de *reparación* ($\mathbf{R}$). Dicho operador nos permite expresar propiedades que deben cumplirse cuando las fallas ocurren; por lo que resulta adecuado para imponer restricciones o condiciones mínimas acerca de qué debería cumplirse luego de la ocurrencia de una falla, de manera tal, que ciertas propiedades puedan ser garantizadas o preservadas. En particular creemos que permite capturar el concepto de *hipótesis de falla*. Las hipótesis de falla son utilizadas frecuentemente para razonar acerca del comportamiento *anormal* y las fallas en tolerancia a fallas. Esencialmente, una hipótesis de falla es una especie de suposición acerca de los escenarios donde ocurren fallas, las cuales permiten garantizar algunas propiedades de tolerancia a fallas en los sistemas. En particular, en la fórmula $\mathbf{R}(\psi \rightsquigarrow \psi')$, ψ puede ser pensada como la hipótesis de falla.

Antes de analizar sus características, veamos la sintaxis de nuestra lógica.

Definición 3.1 (Sintaxis)**.** *Sea* $AP = \{p_0, p_1, \dots\}$ *un conjunto de proposiciones atómicas; definiremos recursivamente los conjuntos* Φ *y* Ψ*, de fórmulas de* estado *y de* camino*, respectivamente; a través de las siguientes gramáticas:*

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

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

Como ya vimos que ocurre con otras lógicas, otros operadores *booleanos*, o también llamados operadores de *estado*, tal como "$\wedge$" (Conjunción), "$\vee$" (Disyunción), etc., pueden definirse en función de los dados ("$\rightarrow$" y "$\neg$"). Para el caso de los operadores temporales, sucede lo mismo, "G" (siempre en el futuro) y "F" (en algún instante futuro) pueden ser expresados como: $\mathsf{G}(\phi) \equiv \phi \,\mathcal{W}\, \bot$, y $\mathsf{F}(\phi) \equiv \top \,\mathcal{U}\, \phi$.

Los operadores *booleanos* y los cuantificadores "A" y "E"de CTL, tienen la semántica usual, tal cual la describimos en el capítulo anterior. Es importante notar que en el caso de los operadores deónticos sólo pueden ser aplicados sobre fórmulas que involucran el operador "$\rightsquigarrow$". Este operador relaciona dos fórmulas de *camino* 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'. Desde una perspectiva más formal, lo cual será clarificado luego que presentemos su semántica, el operador "$\rightsquigarrow$"permite restringir la manera en la cual se combinan las fórmulas de *camino*, bajo el alcance de un operador de *estado*. Algo similar a lo que hacen otras lógicas, como por ejemplo CTL^2. Este punto es clave para aumentar la expresividad de CTL, y para de alguna manera mantener la complejidad de *Model Checking*. Dicho operador es esencial para la descripción de ejecuciones con fallas, dado que el condicional nos permite centrar nuestra atención en ciertas trazas *normativas*, restringiendo los cuantificadores a algunas de estas trazas.

Veamos ahora la semántica de este nuevo formalismo lógico. Comenzaremos definiendo la relación de *satisfactibilidad* $\vDash$, la cual formaliza la *satisfacción* de una fórmula de *estado* sobre estructuras de Kripke coloreadas.

Definición 3.2 (Semántica fórmulas de estado)**.** *Dada una estructura de Kripke coloreada* $M = \langle S, I, R, L, \mathcal{N} \rangle$ *, y un estado* $s \in S$*, la relación de satisfacibilidad está definida de la siguiente manera:*

- $M, s \vDash \top$.
- $M, s \vDash p_i \Leftrightarrow p_i \in L(s)$*, donde* $p_i \in AP$.
- $M, s \vDash \neg\varphi \Leftrightarrow$ *no se cumple* $M, s \vDash \varphi$.
- $M, s \vDash \varphi \rightarrow \varphi' \Leftrightarrow (M, s \vDash \neg\varphi)$ *or* $(M, s \vDash \varphi')$.
- $M, s \vDash \mathsf{A}(\psi) \Leftrightarrow M, \sigma \vDash \psi$ *para toda las trazas* σ *tal que* $\sigma[0] = s$.
- $M, s \vDash \mathsf{E}(\psi) \Leftrightarrow M, \sigma \vDash \psi$ *para alguna traza* σ *tal que* $\sigma[0] = s$.
- $M, s \vDash \mathbf{O}(\psi \rightsquigarrow \psi') \Leftrightarrow$ *para cada* $\sigma \in \mathcal{NT}$ *tal que* $\sigma[0] = s$*, tenemos que para cada* $i \geq 0$*,* $M, \sigma[i..] \vDash \psi$ *implica* $M, \sigma[i..] \vDash \psi'$.
- $M, s \vDash \mathbf{P}(\psi \rightsquigarrow \psi') \Leftrightarrow$ *para algún* $\sigma \in \mathcal{NT}$ *tal que* $\sigma[0] = s$*, tenemos que para cada* $i \geq 0$*,* $M, \sigma[i..] \vDash \psi$ *implica* $M, \sigma[i..] \vDash \psi'$.
- $M, s \vDash \mathbf{R}(\psi \rightsquigarrow \psi') \Leftrightarrow$ *para toda traza* σ *tal que* $\sigma[0] = s$*, tenemos que para cada* $i \geq 0$*: Si* $\sigma[i] \notin \mathcal{N}$*, entonces* $M, \sigma[i..] \vDash \psi$ *implica* $M, \sigma[i..] \vDash \psi'$.

Donde $\mathcal{N} \subseteq S$ *es el conjunto de estados* normales *y* $\mathcal{NT}$ *denota el conjunto de ejecuciones normales.*

Definición 3.3 (Semántica fórmulas de camino)**.** *Dada una estructura de Kripke coloreada $M = \langle S, I, R, L, \mathcal{N} \rangle$, y una traza σ de M, la semántica de una fórmula de camino está definida de la siguiente manera:*

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

Notar que aquí, y para el resto de este trabajo, utilizaremos φ y φ' para fórmulas de *estado* y ψ para fórmulas de *camino*.

Como es usual, vamos a denotar por $M \vDash \varphi$ el hecho que $M, s \vDash \varphi$ se cumpla en todo estado s de M, y por $\vDash \varphi$ el hecho de que valga $M \vDash \varphi$, para toda estructura de Kripke coloreada M.

Cabe aclarar que durante esta tesis, por una cuestión de claridad, frecuentemente vamos a utilizar una versión abreviada de las fórmulas que involucran cuantificadores u operadores deónticos, por ejemplo $\mathbf{O}(\psi)$, la cual es fácil ver que es equivalente a $\mathbf{O}(\top \rightsquigarrow \psi)$. También solemos aplicar operadores de *camino* a fórmulas de *estado* (de hecho en $\mathbf{O}(\top \rightsquigarrow \psi)$, $\top$ es una fórmula de estado), lo cual no estaría permitido por nuestra sintaxis, y esto puede hacerse gracias a que toda fórmula de *estado* φ puede ser expresada como una fórmula de *camino* equivalente, de la forma: $\bot \,\mathcal{U}\, \varphi$.

3.1.1. Primeras Intuiciones de su Aplicación

Para comenzar a entender la semántica de estos operadores deónticos veamos dos pequeños ejemplos. En primer lugar, consideremos la estructura de Kripke ilustrada en la Figura 3.1, en la cual se representa el comportamiento de un proceso cuando intenta acceder a un recurso compartido, lo cual se denomina en el área como *región crítica*. El conjunto de variables proposicionales que utiliza este modelo es: $\{NCritical, Waiting, Critical, Down, Dead\}$, donde cada estado es etiquetado con un conjunto de variables proposicionales que son verdaderas en dicho estado. En este caso:

$NCritical$ será verdadera cuando el proceso esté fuera de la región crítica.

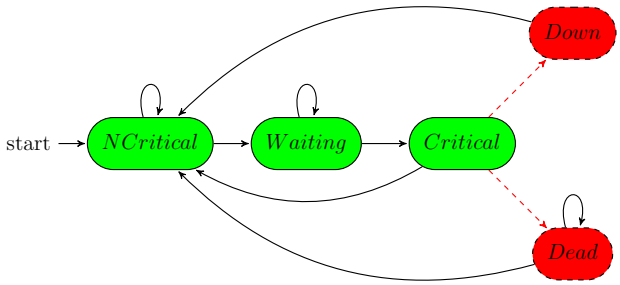


Figura 3.1: Estructura de Kripke que modela Exclusión Mutua de un Proceso.

Waiting aparecerá en aquellos estados donde el proceso este esperando por entrar a dicha región.

Critical valdrá cuando el proceso ya ingresó a la región crítica.

Down representa la ocurrencia de una falla y dicha proposición será verdadera cuando el proceso está "inactivo", pero que podría volver a retomar su función en un futuro.

Dead al igual que en el caso anterior, indica la ocurrencia de una falla pero con la diferencia de que el proceso en este caso no se espera que "despierte" en un futuro.

Notar que este modelo representa los estados de un único proceso, sin embargo esta versión simplicada del modelo de exclusión mutua nos permite explorar y analizar a los operadores deónticos y su poder expresivo.

En este ejemplo, siguiendo las convenciones explicadas en la página 24, los estados y arcos de líneas punteadas representan estados y transiciones *anormales*; es decir estados a los que se llega luego de la ocurrencia de una falla. El resto de los estados y transiciones representan el comportamiento *normal*.

En el caso de este ejemplo hay dos estados *anormales*, etiquetados con *Down* y *Dead*, respectivamente. Si nos concentramos en mirar la sección sin fallas del sistema, podemos notar que hay ciertas ejecuciones en las cuales el proceso nunca accede a la región crítica, ya que podría darse el caso de que se quede esperando, en el estado *Waiting*, indefinidamente. Por lo cual no se cumple la propiedad $\mathbf{O}(\mathsf{F}Critical)$, ya que existen trazas normativas en las cuales nunca se visita el estado *Critical*; sin

embargo $\mathbf{P}(\mathsf{F}Critical)$ es una propiedad válida en dicho sistema.

Por otro lado, una propiedad que puede capturar aquellas situaciones donde el proceso esté inactivo pero con posibilidades de recuperarse , podría especificarse con la siguiente fórmula: $\mathbf{R}(Down \rightsquigarrow \mathsf{EF}Critical)$. El operador $\mathbf{R}()$ justamente nos habilita a imponer ciertas restricciones, de manera de poder garantizar alguna propiedad deseable en un sistema a la hora de una falla. En este caso, dicha fórmula expresa el hecho en el cual el sistema si bien está en un estado *Down*, para que la formula resulte verdadera exige la existencia de una traza *normal* en la cual eventualmente el proceso logra acceder al componente o información compartida. Es decir el proceso eventualmente se recuperará. Notar también, que si quisieramos escribir dicha fórmula respetando la sintaxis dada por la gramática presentada en 3.1, dicha fórmula debería ser la siguiente: $\mathbf{R}(\bot\ \mathcal{U}\ Down \rightsquigarrow \bot\ \mathcal{U}\ (\mathsf{E}(\top\ \mathcal{U}\ Critical)))$. Por cuestiones de claridad, se utilizará las versiones abreviadas, como se explicó anteriormente.

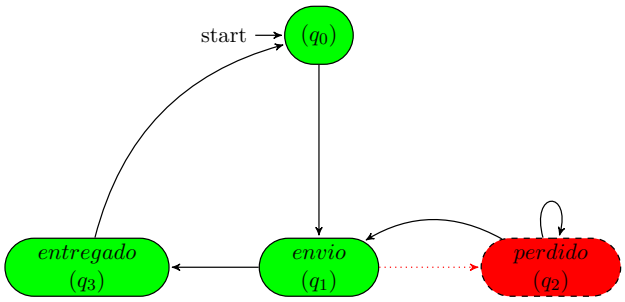


Figura 3.2: Un simple protocolo para envío de mensajes

En segundo lugar, si retomamos la estructura de Kripke presentada en la página 24 (ver Figura 3.2) la misma representa un simple protocolo de comunicación donde los mensajes que se envían pueden perderse debido a que el canal utilizado para la comunicación no es un medio seguro. En este modelo una propiedad que resulta interesante es la de garantizar que ante la ausencia de fallas, los mensajes que se envían sean eventualmente recibidos o entregados a su destinatario. Es decir que en este caso nos quisieramos centrar en los estados *normativos* del modelo, con lo cual el operador $\mathbf{O}()$ resulta adecuado para predicar sobre todas las trazas que no involucran falla alguna. La fórmula $\mathbf{O}(envio \rightsquigarrow \mathsf{F}entregado)$, justamente captura esta situación.

Vale la pena señalar que alguna de estas fórmulas no pueden ser capturadas con los operadores clásicos de CTL. Por ejemplo, la fórmula $\neg\mathbf{P}(recibido \rightsquigarrow \mathsf{X}\neg recibido)$ no puede expresarse en CTL, notar que dicha formula puede expresarse en CTL^* como: $\neg(\mathsf{E}(\mathsf{G}n \wedge \mathsf{G}(recibido \rightarrow \mathsf{X}recibido))) \equiv \mathsf{A}(\mathsf{F}\neg n \vee \mathsf{F}(recibido \wedge \mathsf{X}recibido))$, donde n es una variable proposicional, la cual "marca" el comportamiento *normal* del sistema.

Finalmente, notar que otros operadores deónticos, especialmente la *prohibición*, pueden ser expresados utilizando los operadores introducidos recientemente. *Prohibición* puede ser caracterizado como $\mathbf{F}(\psi) = \neg\mathbf{P}(\psi)$. Intuitivamente, una propiedad está prohibida cuando no puede ser *verdadera* en el comportamiento *normal* del sistema. En otras palabras, si la propiedad que está prohibida es cierta a lo largo de una traza, entonces esa traza contiene fallas.

3.1.2. Propiedades

Los operadores deónticos introducidos poseen varias propiedades interesantes, en el siguiente teorema demostramos algunas de ellas:

Teorema 3.1. *Las siguientes fórmulas son válidas en dCTL:*

O1. $\mathbf{O}(\bot) \equiv \mathbf{O}(\psi) \wedge \mathbf{O}(\overline{\psi})$, *donde* $\overline{\psi}$ *denota la negación de* ψ*, obtenida utilizando los operadores temporales duales de* ψ *y desplazando la negación hacia dentro.*

O2. $\mathbf{O}(\top) \equiv \top$,

O3. $\mathbf{P}(\bot) \equiv \bot$,

O4. $\mathbf{R}(\bot) \vDash \mathsf{AG}\varphi \leftrightarrow \mathbf{O}(\varphi)$,

O5. $\mathbf{R}(\bot) \vDash \mathsf{EG}\varphi \leftrightarrow \mathbf{P}(\varphi)$

O6. $\mathbf{R}(\top) \equiv \top$

O7. $\mathbf{P}(\top)$

O8. $\mathbf{O}(\varphi) \wedge \mathbf{O}(\varphi') \rightarrow \mathbf{O}(\varphi \wedge \varphi')$,

O9. $\mathbf{O}(\varphi) \vee \mathbf{O}(\varphi') \rightarrow \mathbf{O}(\varphi \vee \varphi')$,

O10. $\mathbf{P}(\varphi \wedge \varphi') \rightarrow \mathbf{P}(\varphi) \wedge \mathbf{P}(\varphi')$

O11. $\mathbf{P}(\varphi) \vee \mathbf{P}(\varphi') \rightarrow \mathbf{P}(\varphi \vee \varphi')$

O12. $\mathbf{R}(\varphi) \wedge \mathbf{R}(\varphi') \rightarrow \mathbf{R}(\varphi \wedge \varphi')$

O13. $\mathbf{R}(\varphi) \vee \mathbf{R}(\varphi') \rightarrow \mathbf{R}(\varphi \vee \varphi')$

Proof. *Antes de comenzar con las pruebas de cada propiedad, definamos algunos conjuntos que nos ayudarán en las mismas.*

Sea $M = \langle S, I, R, L, \mathcal{N} \rangle$ *cualquier estructura de Kripke coloreada, un estado* $s \in S$*, llamaremos* $Tr(s)$ *al conjunto de todas las trazas de* M *que comienzan en el estado* s*, es decir* $Tr(s) = \{\sigma / \sigma \in \mathcal{U}_M \wedge \sigma[0] = s\}$*. Donde* $\mathcal{U}_M$ *es el conjunto de todas las trazas de* M *(ver definición 2.2). Finalmente denominaremos* $\mathcal{NT}(s)$*, con* $s \in S$ *al conjunto de todas las trazas normativas que comienzan en un estado* s*.*

- *Para la propiedad* ***O1****, dada cualquier fórmula de camino* ψ*, es directo ver que se cumple* $M, \sigma \nvDash \psi \wedge \overline{\psi}$*. Notar que* $\overline{\psi}$ *puede ser obtenida a partir de* ψ *utilizando los operadores temporales duales y negando la subformula interna. Teniendo en cuenta que debe existir al menos una traza normal, tenemos que* $\mathbf{O}(\psi) \wedge \mathbf{O}(\overline{\psi}) \equiv \bot$*, similarmente ocurre para* $\mathbf{O}(\bot)$*, por lo tanto:* $\mathbf{O}(\psi) \wedge \mathbf{O}(\overline{\psi}) \equiv \mathbf{O}(\bot)$*.*

- *Para* ***O2****, dada cualquier ejecución normal* σ*, tenemos que se cumple* $M, \sigma \vDash \top$*, y entonces vale que* $M, s \vDash \mathbf{O}(\top)$*.*

- *Para* ***O3****, dada cualquier ejecución normal* σ*, tenemos que se cumple* $M, \sigma \nvDash \bot$*, luego vale que* $M, s \nvDash \mathbf{P}(\bot)$*.*

- *Para* ***O4****, en primer lugar notar que* $\mathbf{R}(\bot)$ *es una abreviación de* $\mathbf{R}(\bot \mid \top)$*, si* $M, s \vDash \mathbf{R}(\bot)$*, entonces para cada* $\sigma \in Tr(s)$ *tenemos que, si* $\sigma[i] \notin \mathcal{N}$*, entonces* $M, \sigma[i..] \vDash \bot$*, esto implica que* $\sigma \in \mathcal{NT}$*, lo cual significa que* $\mathcal{NT}(s) = Tr(s)$ *para cualquier* s*, de lo cual se deduce la propiedad esperada.*

- *Para* ***O5****, continuando con el razonamiento de la propiedad anterior , sabemos que si se cumple* $\mathbf{R}(\bot)$ *tenemos que* $\mathcal{NT}(s) = Tr(s)$ *para cualquier* s *, por lo tanto en particular se cumple* $\mathsf{EG}\varphi \leftrightarrow \mathbf{P}(\varphi)$*.*

- *Para* ***O7****, sabiendo que para cada modelo existe al menos una traza normativa, entonces se cumple* $M, s \vDash \mathbf{P}(\top)$

- *Para* ***O8****, supongamos que* $M, s \vDash \mathbf{O}(\varphi) \wedge \mathbf{O}(\varphi')$*, luego para algún* $\sigma \in \mathcal{NT}(s)$ *tenemos que* $M, \sigma \vDash \varphi$ *y* $M, \sigma \vDash \varphi'$*, luego entonces vale* $M, s \vDash \mathbf{O}(\varphi \wedge \varphi')$*.*

- *Para **O9**, al igual que para la propiedad anterior, supongamos que* $M, s \vDash \mathbf{O}(\varphi) \vee \mathbf{O}(\varphi')$*, luego para algún* $\sigma \in \mathcal{NT}(s)$ *tenemos que* $M, \sigma \vDash \varphi$ *ó* $M, \sigma \vDash \varphi'$*, luego entonces vale* $M, s \vDash \mathbf{O}(\varphi \vee \varphi')$.
- *Para **O10**, supongamos que* $M, s \vDash \mathbf{P}(\varphi \wedge \varphi')$*, luego para algún* $\sigma \in \mathcal{NT}(s)$ *tenemos que* $M, \sigma \vDash \varphi \wedge \varphi'$*, luego entonces vale* $M, s \vDash \mathbf{P}(\varphi) \wedge \mathbf{P}(\varphi')$.
- *Para **O11**, supongamos que* $M, s \vDash \mathbf{P}(\varphi) \vee \mathbf{P}(\varphi')$*, luego para algún* $\sigma \in \mathcal{NT}(s)$ *tenemos que* $M, \sigma \vDash \varphi$ *ó bien* $M, \sigma \vDash \varphi'$*, luego entonces vale* $M, s \vDash \mathbf{P}(\varphi \vee \varphi')$.
- *Para **O12**, supongamos que* $M, s \vDash \mathbf{R}(\varphi) \wedge \mathbf{R}(\varphi')$*, luego existe algún* $\sigma \in \mathcal{NT}(s)$ *tal que cumple que* $M, \sigma \vDash \varphi$ *y también* $M, \sigma \vDash \varphi'$*, luego entonces vale* $M, s \vDash \mathbf{R}(\varphi \wedge \varphi')$.
- *Para **O13**, supongamos que* $M, s \vDash \mathbf{R}(\varphi) \vee \mathbf{R}(\varphi')$*, luego existe algún* $\sigma \in \mathcal{NT}(s)$ *tal que cumple* $M, \sigma \vDash \varphi$ *ó bien cumple* $M, \sigma \vDash \varphi'$*, luego entonces vale* $M, s \vDash \mathbf{R}(\varphi \vee \varphi')$.

Intuitivamente, las propiedades expresan lo siguiente:

La Propiedad **O1** expresa que si tenemos obligaciones que se contradicen entre si, eso es equivalente a decir que *falso* está obligado, es decir que eventualmente ocurrirá una falla en cada ejecución de dicho sistema. En cambio la Propiedad **O2** se refiere al hecho de que decir que *verdadero* está obligado, es equivalente a *verdadero*. Y propiedades similares se cumplen para el operador de *Permiso*. En el caso de la Propiedad **O3**, la misma indica que *falso* nunca puede estar permitido.

Las reglas de deducción establecen que, en la ausencia de fallas, los operadores deónticos pueden ser expresados utilizando los operadores estándar de CTL.

Las propiedades del operador deóntico **R** expresan que *verdadero* siempre se cumple luego de una falla, mientras que $\mathbf{R}(\bot)$ expresa el hecho de que en algún momento futuro dejarán de ocurrir fallas, esta última afirmación implica $\mathbf{P}(\top)$; es decir que existe alguna ejecución *normal* .

Por último de la propiedad **O8** a la **O13**, se establece una relación entre los operadores deónticos y los operadores *booleanos*.

3.2. Expresividad y Complejidad de dCTL

En esta sección presentamos algunos resultados sobre la *expresividad* y *complejidad* de dCTL. Los resultados de complejidad nos habilitan a mostrar que, no solamente se pueden verificar automáticamente las propiedades de sistemas tolerantes a

fallas, sino que también esta tarea puede ser llevada a cabo en un tiempo *polinomial*, con respecto al tamaño del modelo y de la fórmula que se de desea chequear.

Comenzaremos demostrando que las fórmulas de dCTL pueden ser verificadas, a través de *model checking*, dando una traducción formal de nuestra lógica en una lógica más expresiva como lo es CTL*, la cual es una lógica clásica en el ámbito de model checking [24] y en segundo lugar la compararemos con otras muy conocidas lógicas tal como lo son CTL, CTL$^+$, ECTL, entre otras. Un detalle técnico a tener en cuenta es que para poder llevar al cabo las comparaciones, necesitamos codificar nuestras estructuras de *Kripke* coloreadas en modelos de *Kripke* estándar.

En la literatura acerca de lógicas temporales para *model checking* [39], la noción de expresividad tiene un significado bien preciso:

Definición 3.4 (Expresividad)**.** *Se considera que una lógica L' es más expresiva que otra lógica L, si para cada fórmula ϕ de L existe una fórmula ϕ' en L', tal que $M \vDash \phi$ si y sólo si $M \vDash \phi'$, es decir, si existe una traducción entre las formulas que preserve la relación de* satisfactibilidad*.*

En esta definición, las estructuras semánticas consideradas son exactamente las mismas, es decir estructuras de *Kripke*.

Sin embargo algo que ocurre frecuentemente a la hora de comparar diferentes clases de lógicas, en muchos casos con semánticas distintas, es que necesitamos también hacer una traducción entre las estructuras semánticas de las mismas. Por ejemplo, en [109], definen esta relación de la siguiente manera:

Definición 3.5 ([109])**.** *Dada 2 lógicas $L = \langle \mathcal{M}, Form, \vDash \rangle$ y $L = \langle \mathcal{M}', Form', \vDash' \rangle$ diremos que L' es más expresiva que L sii:*

- *Existe una traducción $\alpha : Form \to Form'$,*
- *Existe una traducción $\gamma : \mathcal{M}' \to \mathcal{M}$,*
- *Se cumplen las siguientes condiciones: $\gamma(M) \vDash \phi \Leftrightarrow M \vDash \alpha(\phi)$, para cada M y ϕ,*
- *γ es suryectiva.*

En ésta definición las lógicas son consideradas tuplas $\langle \mathcal{M}, Form, \vDash \rangle$, donde $\mathcal{M}$ es una colección de modelos de las estructuras, $Form$ es el conjunto de formulas y $\vDash$ es la relación de *satisfactibilidad*.

Si comparamos la interpretación de *expresividad* que se utiliza frecuentemente en *model checking* la misma es coherente con esta definición ya que en ese caso la traducción entre modelos sería la identidad. Notar también que en el caso de las lógicas empleadas para realizar *model checking*, cambiar el modelo tiene poco sentido, ya que la estructura de *kripke* es una definición abstracta del sistema que se desea verificar. Por lo que a la hora de comparar dCTL con otras lógicas lo haremos con respecto a estructuras de *Kripke* estándar.

3.2.1. Traducción de fórmulas dCTL a fórmulas CTL*

Primero, definimos una traducción de fórmulas dCTL a fórmulas CTL*. Notar que esta tradución, la cual no es díficil de realizar, introduce una nueva variable proposicional, a la cual denotamos n, para "marcar" el comportamiento *normal* del sistema.

Definición 3.6. *La traducción τ de fórmulas dCTL sobre un alfabeto AP, a fórmulas CTL* con un alfabeto $AP \cup \{n\}$, para una variable libre ($n \notin AP$), está definida de la siguiente forma:*

- $\tau(\top) = \top$.
- $\tau(p_i) = p_i$.
- $\tau(\neg\varphi) = \neg\tau(\varphi)$.
- $\tau(\varphi \to \varphi') = \tau(\varphi) \to \tau(\varphi')$.
- $\tau(\mathsf{A}(\psi)) = \mathsf{A}(\tau(\psi))$.
- $\tau(\mathsf{E}(\psi)) = \mathsf{E}(\tau(\psi))$.
- $\tau(\mathbf{O}(\psi \rightsquigarrow \psi')) = \mathsf{A}(\mathsf{G}n \to \mathsf{G}(\tau(\psi) \to \tau(\psi')))$
- $\tau(\mathbf{P}(\psi \rightsquigarrow \psi')) = \mathsf{E}(\mathsf{G}n \wedge \mathsf{G}(\tau(\psi) \to \tau(\psi')))$
- $\tau(\mathbf{R}(\psi \rightsquigarrow \psi')) = \mathsf{A}(\mathsf{G}(\neg n \to (\tau(\psi) \to \tau(\psi'))))$
- $\tau(\mathsf{X}\varphi) = \mathsf{X}(\tau(\varphi))$.
- $\tau(\varphi \,\mathcal{U}\, \varphi') = \tau(\varphi) \,\mathcal{U}\, \tau(\varphi')$.

- $\tau(\varphi \mathcal{W} \varphi') = \tau(\varphi) \mathcal{W} \tau(\varphi')$.

De manera informal se podría decir, que las obligaciones son traducidas como un cuantificador universal sobre trazas que siempre satisfacen n (y por lo tanto las mismas pueden pensarse como ejecuciones normativas), sobre esas trazas requerimos que se satisfagan las correspondientes fórmulas temporales. Lo mismo ocurre para los operadores de *permiso* y *recuperación*.

La siguiente transformación entre estructuras de Kripke y estructuras de Kripke coloreadas, nos permiten argumentar justamente sobre la preservación semántica de la traducción de dCTL a CTL*.

3.2.2. Traducción entre modelos

Definición 3.7. *Dada $M = \langle S, R, L\rangle$ una estructura de Kripke definida sobre el alfabeto $AP \cup \{n\}$. A partir de M, definimos la estructura de Kripke coloreada $M^* = \langle S, R, L', \mathcal{N}\rangle$ sobre el alfabeto AP, de la siguiente forma:*

- *L' es L restringido a AP.*
- *$s \in \mathcal{N} \Leftrightarrow M, s \vDash n$.*

Debajo demostraremos que esta traducción es *sound* con respecto a la consecuencia lógica (es decir, preserva las consecuencias de una fórmula). Primero es interesante hacer notar que las estructuras de *Kripke* coloreadas pueden traducirse muy fácilmente a estructuras de *Kripke* estándares utilizando n para "marcar" los estados verdes (normales). Como es esperable, esta traducción es 1 a 1, por lo cual podemos definirla de la siguiente manera:

Definición 3.8. *Sea $M = \langle M, R, L, \mathcal{N}\rangle$ un modelo de Kripke coloreado, entonces definimos la siguiente estructura de* Kripke *estándar: $M^+ = \langle M, R, L'\rangle$ donde: $L'(p) = L(p)$ para cualquier $p \neq n$, y $L'(n) = \{s \mid s \in \mathcal{N}\}$.*

Notar también que, cuando definimos estructuras coloreadas, asumimos que cada estado *normal* tiene al menos un sucesor que también lo es; esta asunción está fundamentada por el hecho de que especificamos sistemas tolerantes a fallas, y es natural esperar que ante la ausencia de fallas los sistemas se comporten de la manera correcta, es decir que tenga al menos una traza *normal*. Sin embargo, en el caso de aquellos estados *normativos* que no tengan sucesores se tratan como es usual a la hora de realizar *model checking*, se les agrega un ciclo al mismo estado.

Esto implica, que la traducción $()^+$ no es suryectiva, porque podríamos tener una estructura de *Kripke* donde un estado *normal* que satisface n no tenga un sucesor que haga a n verdadera, lo cual no cumpliría nuestra restricción. Pero dado que estamos comparando *expresividad* de diferentes lógicas con respecto a este modelo particular de tolerancia a fallas, a partir de este momento vamos a considerar que cada estado de nuestra estructura de *Kripke* satisface la fórmula $\Phi = n \leftrightarrow EX(n)$, es decir que cumple con la asunción propuesta.

Teniendo en cuenta esto, tenemos las siguientes propiedades:

Teorema 3.2. *Sea $\mathcal{F}$ un conjunto de estructuras de Kripke sobre $AP \cup \{n\}$, entonces para cada $M \in \mathcal{F}$:*

- $(M^*)^+ = M$,
- $(M^+)^* = M$,

Es decir, $()^*$ y $()^+$ son funciones *inversas* entre si y son *biyectivas*

Ahora probaremos que las condiciones de la Definición 3.5 se cumplen para CTL^* y dCTL, es decir CTL^* es más expresiva que dCTL

Teorema 3.3. *Para cada $M = \langle S, R, L\rangle$ definido sobre el alfabeto $AP \cup \{n\}$, y cada fórmula dCTL φ sobre AP, se cumple:*

$$M^*, s \vDash_{dCTL} \varphi \Leftrightarrow M, s \vDash_{CTL^*} \tau(\varphi).$$

Proof. *La prueba se realiza por inducción estructural sobre φ. El caso base se define de la siguiente manera. Dada $p_i \in AP$ una variable proposicional (distinta de n) y asumiendo $M^*, s \vDash p_i$, esto es equivalente a que $p_i \in L'(s)$, pero dado que L' es una restricción de L sobre AP, esto es lo mismo que $p_i \in L(s)$, por lo que tenemos $M, s \vDash p_i$.*

Los casos inductivos los definimos de la siguiente manera. Primero cabe aclarar, que para fórmulas de estado φ_0, φ_1 que satisfacen la propiedad, se cumple lo siguiente: $M, \sigma \vDash \varphi_0 \,\mathcal{U}\, \varphi_1$ sii $M^*, \sigma \vDash \varphi_0 \,\mathcal{U}\, \varphi_1$, y similarmente para $\mathcal{W}$ y X.

Probemos esta propiedad para weak until, para los demás operadores puede realizarse la prueba de manera similar. Supongamos que se cumple $M^*, s \vDash \varphi_0 \,\mathcal{W}\, \varphi_1$, es decir $\exists i : \forall j < i : M^*, \sigma[j] \vDash \varphi_0$ y $M^*, \sigma[j] \vDash \varphi_1$, por la asunción que tenemos sobre φ_0 y φ_1; ademas el hecho de que M y M^* tienen el mismo conjunto de ejecuciones, tenemos que $\exists i : \forall j < i : M, \sigma[j] \vDash \tau(\varphi_0)$ y $M^*, \sigma[j] \vDash \tau(\varphi_1)$ y por lo tanto se cumple $M, s \vDash \varphi_0 \,\mathcal{U}\, \varphi_1$.

Ahora, probemos el caso inductivo para los operadores deónticos y de camino. Para $\neg$ *y* $\wedge$ *la prueba es directa. Para* $\mathsf{A}(\varphi_0\ \mathcal{U}\ \varphi_1)$*, asumimos* $M^*, s \vDash \mathsf{A}(\psi)$*, por definición de* $\vDash$ *sabemos que para todo* $\sigma \in Tr(s)$ *tenemos que se cumple* $M^*, \sigma\psi$*, dado que el conjunto de trazas de* M *y* M^* *son los mismos y la propiedad que acabamos de probar arriba , tenemos que* $M, s \vDash \mathsf{A}(\psi)$*. La prueba para* $\mathsf{E}(\psi)$ *y* $\mathsf{X}\psi$ *son similares.*

Probemos la propiedad para los operadores deónticos. Para $\mathbf{P}(\psi \rightsquigarrow \psi')$*,* $M^*, s \vDash \mathbf{P}(\psi \rightsquigarrow \psi')$ *sii existe* $\sigma \in \mathcal{NT}(s)$ *tal que, si* $M^*, \sigma[i..] \vDash \psi$*, entonces* $M^*, \sigma[i..] \vDash \psi'$*. Para la definición de* M^**, tenemos que para cada* i $n \in L(\sigma[i])$*; considerando la propiedad que probamos anteriormente para fórmulas de camino, tenemos que* $M^*, \sigma[i..] \vDash \tau\psi'$*, entonces* $M^*, \sigma[i..] \vDash \tau\psi$*, lo cual significa* $M, s \vDash \mathsf{A}(\mathsf{G}n \rightarrow \mathsf{G}(\tau(\psi) \rightarrow \tau(\psi')))$*. Las demostraciones para el resto de los operadores deónticos son similares.*

La relación $\vDash$, y las funciones $\tau, (-)^*$ forman una conexión de *Galois*. Usando este teorema podemos probar otras propiedades interesantes de esta transformación.

Una propiedad que podemos obtener de manera directa es la que establece que tanto los modelos coloreados como los "sin colorear" satisfacen las mismas fórmulas CTL^* sobre el alfabeto AP.

Corolario 3.1. *Para cualquier* CTL^* *fórmula* ϕ *con variables proposicionales del alfabeto* AP *tenemos que se cumple*

$$M^*, s \vDash \phi \Leftrightarrow M, s \vDash \phi$$

para cualquier estado s*.*

Proof. *Notar que* $\tau(\varphi) = \varphi$ *para las fórmulas* CTL^* *con variables proposicionales en* AP*.*

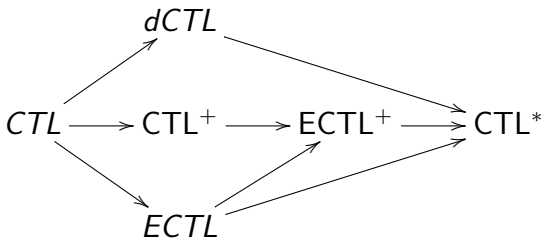
Estos resultados acerca de la traducción τ nos permiten comparar la expresividad de dCTL con respecto a CTL y otros formalismos relacionados.

3.2.3. Expresividad

Como ya demostramos previamente, existe una traducción formal de nuestra lógica en una lógica más expresiva como lo es CTL^*, la cual es más expresiva que CTL, en esta sección analizaremos la expresividad de CTL en comparación con otras lógicas temporales. Notar que también podríamos hacer una comparación con μ-*calculus*, o algún fragmento del mismo, aunque dicha lógica no forma parte del conjunto de lógicas clásicas a la hora de realizar especificaciones de *software*.

Cabe destacar que nuestra traducción de los operadores deónticos de dCTL a CTL* involucra algunas fórmulas CTL* que no se pueden expresar en CTL. En particular, la fórmula $\neg\mathbf{P}(p \rightsquigarrow \mathsf{X}p)$, la cual es traducida a $\mathsf{A}(\mathsf{F}\neg n \vee \mathsf{F}(p \wedge \mathsf{X}\neg p))$ en CTL*, no es expresable en CTL, ni en ninguna de sus extensiones CTL+, ECTL y ECTL+. Este resultado de expresividad, que se deduce de propiedades presentadas en [51], se puede resumir en el siguiente teorema.

Teorema 3.4. *El poder expresivo de las lógicas CTL, dCTL, CTL+, ECTL, ECTL+ y CTL*, se puede representar mediante el siguiente diagrama de inclusión:*



Demostración. La prueba de que la fórmula $\mathsf{AF}(p \wedge \mathsf{X}\neg p)$ no se puede expresar en ECTL+ se puede encontrar en [51] (dentro del Teorema 5).

Podemos utilizar argumentos similares para probar que $\mathsf{A}(\mathsf{F}(\neg n) \vee \mathsf{F}(p \wedge \mathsf{X}\neg p))$ tampoco se puede expresar en ECTL+. Consideremos las siguientes secuencias de modelos $N_1, N_2, N_3, \ldots$ y $M_1, M_2, M_3, \ldots$ utilizadas en el Teorema 5 de [51], y fijemos $n = true$ en cada estado de estos modelos. Dado que tenemos que $M_i, a_i \vDash \mathsf{AF}(p \wedge \mathsf{X}\neg p)$, también tenemos que $M_i, a_i \vDash \mathsf{A}(\mathsf{F}(\neg n) \vee \mathsf{F}(p \wedge \mathsf{X}\neg p))$. Por otro lado ya que este también es el caso de que $N_i, a_i \nvDash \mathsf{AF}(p \wedge \mathsf{X}\neg p)$, para cada i, entonces este debería ser el caso en que $N_i, a_i \nvDash \mathsf{A}(\mathsf{F}(\neg n) \vee \mathsf{F}(p \wedge \mathsf{X}\neg p))$. Teniendo en cuenta que estas estructuras no pueden ser distinguidas por ninguna fórmula ECTL+, es directo ver que $\neg\mathbf{P}(p \rightarrow \mathsf{X}p)$ no puede ser expresada en ECTL+. Más aún, esta fórmula no se puede expresar en ninguna de las siguientes lógicas: CTL, ECTL ó CTL+, las cuales son sublógicas de ECTL+. □

Este mismo teorema también puede extenderse para probar que algunas fórmulas de dCTL no se pueden expresar en otras lógicas relacionas, particularmente en CTL2. El razonamiento para probarlo es básicamente el mismo que utilizamos en la prueba anterior.

CTL2 es una lógica propuesta por *Kupferman* and *Grumberg* [111], esta lógica resulta particularmente interesante dado que es estrictamente más expresiva que

CTL y su *model checking* está en P. La gramática de esta lógica es la siguiente [118]:

$$\begin{array}{lcl} S & ::= & p_i \mid \neg S \mid S \wedge S \mid S \vee S \mid \mathsf{E}P \mid \mathsf{A}P \\ P & ::= & P_1 \mid P_2 \\ P_1 & ::= & \mathsf{X}S \mid S\,\mathcal{U}\,S \mid \neg P_1 \\ P_2 & ::= & \mathsf{X}P_1 \mid P_1\,\mathcal{U}\,S \mid S\,\mathcal{U}\,P_1 \mid \neg P \mid S \wedge P_1 \mid P_1 \wedge S \end{array}$$

Entonces en esta lógica, podemos escribir, por ejemplo: $\mathsf{E}((p\,\mathcal{W}\,q)\,\mathcal{W}\,t)$, es decir la misma nos permite tener un segundo nivel de fórmulas de camino. Como hacen notar en [51], dicha lógica puede ser traducida a ECTL^+, por lo cual considerando nuestro teorema de expresividad 3.4, obtenemos el siguiente resultado:

Corolario 3.2. *CTL^2 y dCTL son incomparables.*

Finalmente, nos gustaría hacer notar que el problema de *model checking* para CTL, ECTL, CTL^+ y CTL^2 está en P. Mientras que para ECTL^+ y CTL^* este problema es *PSPACE-complete*. Como vimos nuestra lógica es más expresiva que CTL y permite expresar fórmulas que no pueden ser expresadas en ECTL^+, entonces una pregunta que surge naturalmente es si el problema de model checking para dCTL también es *PSPACE-complete*, lo cual sería algo que no deseamos, pero como sabemos muchas veces es el precio a pagar por ganar en expresividad. Sin embargo, como mostramos a continuación en el caso de nuestra lógica el problema de model checking mantiene la complejidad polinomial. Esto, combinado con el hecho de que dCTL permite expresar propiedades que no pueden expresarses en otras sublógicas de CTL^*, consideradas "eficientes" (en el sentido que sus algoritmos de model checking están en P), hacen que nuestra lógica sea un fragmento novedoso de CTL^*.

En la siguiente sección analizaremos y probaremos que el problema de *model checking* para dCTL se encuentra en P.

3.2.4. Complejidad del Algoritmo de *Model Checking*

Teorema 3.5. *El problema de model checking para dCTL está en P.*

Demostración. La idea principal atrás de esta prueba es la adaptación del algoritmo de model checking de CTL, descripto en [24], para poder chequear fórmulas deónticas. Estos procesos adicionales pueden ser completados en tiempo polinomial, utilizando algoritmos de alcanzabilidad. Además, notar que el hecho de que la lógica se pueda traducir a μ-*calculus* con un anidamiento de a los sumo dos puntos fijos también nos garantiza que el algoritmo sea polinomial.

Modelos temporales son implementados como grafos, por lo que nuestro conjunto $\mathcal{N}$, de las estructuras de *Kripke* coloreadas, puede ser capturado simplemente agregando una variable booleana n, la cual se la setea en *true* en todos aquellos estados que pertenecen al conjunto $\mathcal{N}$ (Notar que de acuerdo a nuestra restricción sobre las estructuras de kripke coloreadas, si n es verdadero en algún estado s, entonces n también es verdadera en algún estado sucesor de s). Para poder verificar si se cumple $M \vDash \varphi$, comenzaremos calculando los siguientes conjuntos $Sat(\psi) = \{s \mid M, s \vDash \psi\}$, para cada subfórmula ψ de φ, comenzando por las subfórmulas ubicadas en las hojas del árbol sintáctico de φ. La principal dificultad técnica es evitar la *explosión* en la traducción de las fórmulas de la forma $\mathsf{A}(\phi)$ and $\mathsf{E}(\phi)$, etc. En nuestro caso esta *explosión* es evitada debido a que estos cuantificadores siempre se encuentran aplicados a combinaciones Booleanas de a lo sumo una fórmula de camino. Estas fórmulas pueden ser chequeadas utilizando los procedimientos para fórmulas equivalentes en CTL.

Resta demostrar como chequear fórmulas que involucren los operadores deónticos, en primer lugar veamos el caso de las fórmulas de la forma $\mathbf{O}(\psi \rightsquigarrow \psi')$ y $\mathbf{P}(\psi \rightsquigarrow \psi')$. Por cuestiones de simplicidad, pero sin perder generalidad, podemos restringir el análisis para operadores deónticos aplicados a una única fórmula de camino (recordar que cuando hay implicación entre fórmulas de camino bajo el alcance de un cuantificador, esta puede ser traducida a una fórmula de estado de una longitud fija). Consideremos las siguientes fórmulas:

- $\mathbf{O}(\psi)$: Esta fórmula es equivalente a $\mathsf{A}(\mathsf{G}n \rightarrow \mathsf{G}(\psi))$. Para poder calcular el conjunto $Sat(\mathbf{O}(\psi))$, podemos restringirnos a aquellos estados donde n es verdadera, y chequear si este conjunto satisface $\mathsf{A}(\psi)$. Esto puede ser fácilmente verificado utilizando el algoritmo de model checking de CTL.

- $\mathbf{P}(\psi)$: Dicha fórmula es equivalente a la fórmula de CTL^* $\mathsf{E}(\mathsf{G}n \wedge \mathsf{G}\psi)$. Para calcular el conjunto $Sat(\mathbf{P}(\psi))$, vamos a chequear si existe algún camino de estados que satisfacen n, y donde se cumple $\mathsf{G}\psi$; esto puede ser llevado a cabo chequeando $\mathsf{E}\psi$ para los nodos donde n es verdadera (esto puede ser realizado en tiempo polinomial, inductivamente). Entonces, $s \in Sat(\mathbf{P}(\psi))$, si existe algún sucesor de estos estados en el cual se cumpla n y $\mathsf{E}\psi$, simultáneamente

- $\mathbf{R}(\psi)$ demanda una técnica muy similar a similar a las planteadas anteriormente.

Resumiendo, estos procesos pueden ser llevados a cabo usando *depth-first search*, y los algoritmos para chequear fórmulas de CTL. El resto de nuestros algoritmos

también son polinomial, por lo que el algoritmo completo de model checking para dCTL es también polinomial con respecto al tamaño del modelo y de la longitud de la fórmula.

En el siguiente capítulo nos avocaremos a conocer una de las implementaciones realizadas de este algoritmo, utilizando técnicas de *model checking simbólico* de manera de obtener una mayor eficiencia y poder trabajar con modelos de mayor escala.

□

3.3. Resumen del capítulo

En este capítulo hemos presentado una lógica temporal ramificada especialmente enriquecida para describir propiedades de sistemas tolerantes a fallas, empleando operadores deónticos para este propósito. Los operadores deónticos, son quienes nos ayudan a hacer la distinción entre estados y comportamiento normal y anormal del sistema, ya que los mismos nos proveen de una expresividad lo suficientemente *rica* para describir muchas de las propiedades de interés en el área de tolerancia a fallas. Demostramos que algunas fórmulas que pueden expresarse en nuestra lógica no pueden ser expresadas en otros fragmentos de CTL^*, incluida ECTL^+ y sus sublógicas. Sin embargo, para nuestra alegría, a pesar de ganar en poder expresivo el problema de *model checking* de nuestra lógica está en P, cosa que no sucede para ECTL^* y CTL^*, para las cuales este problema es *PSPACE-complete*. En el siguiente capítulo analizaremos el algoritmo de model checking en detalle, para poder entender su funcionamiento con más detenimiento.

Estos resultados, junto con nuestros argumentos acerca de la utilidad de la misma para la especificación de sistemas tolerantes a fallas, hacen de dCTL un fragmento interesante de CTL^*. Para poder exhibir justamente esta utilidad en el Capítulo 5, desarrollaremos varios casos de estudio, que nos permitirán mostrar la expresividad de esta lógica. Antes, expresar propiedades temporales que predicaran acerca de situaciones tolerantes a fallas podía lograrse con un enfoque más a *bajo* nivel, es decir, directamente se refería a los estados con fallas a través de algunas fórmulas atómicas que justamente caracterizaban dichos estados. Creemos que los operadores deónticos aquí propuestos proveen una manera *indirecta*, o de más *alto* nivel de referirse a las fallas a la hora de especificar propiedades de tolerancia a fallas, ya que capturan algunos *patterns* útiles en este contexto. Más aún, algunas propiedades de los operadores deónticos permiten razonar acerca de descripciones formales a un

alto nivel de abstracción.

Capítulo 4

dCTL Model Checking

En este capítulo describiremos el algoritmo de *model checking* de dCTL. Deteniéndonos en la manera de computar los operadores de la misma.

También introduciremos un simple lenguaje concurrente, el cual está basado en *Unity* [32] y en el lenguaje definido por *Arora* [15] los cuales son lenguajes comunes a la hora de escribir sistemas concurrentes y tolerantes a fallas; a través de guardas nos permitirá de una manera sencilla especificar los modelos de sistemas tolerantes a fallas sobre los cuales queremos verificar si cumple o no cierta propiedad. Por otro lado, se utilizarán algunos pequeños casos de estudio a fin de ayudar a entender el proceso de verificación que realizamos.

Dedicaremos una sección para analizar la expresividad de nuestro formalismo lógico con respecto a CTL, haciendo hincapié en las limitaciones que presenta ésta última a la hora de capturar algunas nociones clásicas de tolerancia a falla como *permiso*.

Por último, cabe destacar que hemos desarrollado una implementación de este algoritmo utilizando JAVA, el principal objetivo de este desarrollo fue poder evaluar a nivel práctico la *performance* de los algoritmos aquí presentados. Dicha herramienta se implementó utilizando una técnica conocida como *model checking simbólico*, la cual a diferencia del enfoque clásico, utiliza una representación abstracta de los estados y las transiciones del sistema, con el fin de mejorar la escalibilidad y performance de la misma; los cuales son dos puntos claves a la hora de verificar sistemas. Para este capítulo asumimos un conocimiento básico en *model checking simbólico*, semánticas de punto fijo, y en *Diagramas de Decisión Binarios*, en inglés denominados *BDD*. Los lectores pueden encontrar una breve introducción a dichos temas en la sección 2.7, o bien pueden consultar bibliografía clásica del área como [39, 26, 24]

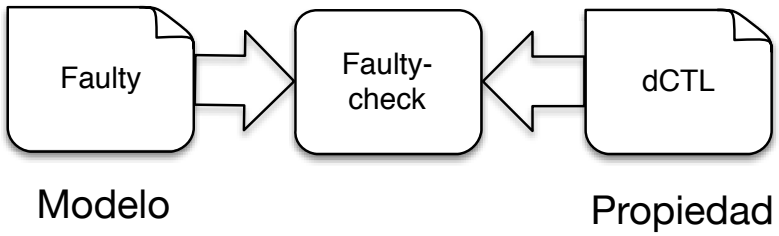


Figura 4.1: Esquema general

para más detalles.

Este model checker, al cual denominamos *FaultyCheck*, es *open source* y se encuentra disponible en [1]. En la documentación que lo acompaña, se pueden encontrar detalles del diseño y la implementación del mismo; sin embargo en este capítulo se describirán de manera general algunas de las características de ésta herramienta, sobre todo en lo que se refiere a las técnicas y estrategias adoptadas, con el objetivo de cuidar la performance de la misma.

4.1. Model Checker Simbólico para dCTL

Como ya adelantamos, este tipo de técnica trabaja con una representación simbólica del sistema y de las fórmulas que vamos a verificar, si bien existen distintas maneras para implementar este enfoque, una manera clásica de hacerlo es utilizando *BDD*.

El uso de fórmulas y *BDD* a la hora de representar programas concurrentes es bien conocido. En nuestro trabajo adoptamos la manera de construir las estructuras que representan el sistema similar a las ideas presentadas en la tesis doctoral de *Kenneth L. McMillan* [105].

Sin embargo para poder especificar los sistemas sobre los cuales queremos verificar si cumple o no cierta propiedad, es necesario poseer un lenguaje adecuado que nos ayude en dicha tarea. En nuestro caso definimos un simple lenguaje concurrente de guardas, el cual nos permite expresar programas, el mismo utiliza un estilo similar a *Unity* [32], sin embargo el mismo está especialmente diseñado para poder distinguir el comportamiento normal de los sistemas de aquellas situaciones en las cuales hay fallas presentes. En la Figura 4.1 se presenta el esquema general del model checker.

4.1.1. Lenguaje Faulty

En el lenguaje *Faulty* cada programa está compuesto por un número finito de procesos $p_0, \ldots, p_n$ ejecutándose de manera paralela, bajo una semántica de *interleaving*. Los procesos están constituidos principalmente por un conjunto de declaraciones de variables y un conjunto de *guardas*. Notar que además cada proceso contiene dos secciones `Initial` y `Normative` en las cuales se especifica, a través de fórmulas, las condiciones que satisfacen los estados "iniciales" y los estados "normales", respectivamente.

En el siguiente ejemplo se presenta un proceso llamado *P1*, el cual contiene dos variables Booleanas llamadas *a* y *b*, su condición *inicial* y *normativa*, conjuntamente con las dos guardas que especifican la función del proceso dado.

```
Process P1 {
    a: BOOL;
    b: BOOL;

    Initial: a && b;
    Normative: a || b ;

    a -> a=!b;
    !a -> a=!a, b=(a||b);
}
```

En este lenguaje se consideran dos tipos básicos: `Bool` y `Integer`. Si nos detenemos a analizar cada una de las guardas se puede apreciar que cada una tiene la forma $G_i \rightarrow C_i$, donde G_i es la condición de la guarda (expresión booleana) y C_i es un conjunto de asignaciones del estilo $x_i = E_i$, con x_i variable y E_i una expresión del tipo correcto. Las fórmulas que aparecen en la sección *Normativa* podrían ser, en principio, cualquier fórmula temporal de CTL, sin embargo por cuestiones de simplificación en este trabajo sólo consideraremos fórmulas proposicionales.

En lo que se refiere a la comunicación entre procesos, la misma se puede realizar a través de canales o memoria compartida . Por último cabe destacar que cada programa tiene un proceso `Main` que el cual es responsable de comenzar a ejecutar las instancias de los procesos, mediante la directiva `run`.

Veamos a continuación la gramática de dicho lenguaje, para conocer su sintaxis:

Notación:

$\langle$simb$\rangle$	$\langle$simb$\rangle$ es un no-terminal.
simb	**simb** es un terminal
$\left[x\right]$	cero o una ocurrencia de x; notar que corchetes entre comillas '[' ']' son terminales.
x^*	cero o más ocurrencias de x.
x^+,	una lista de una o más ocurrencias de x's separadas por coma.
$\left\{ \right\}$	llaves son usadas para agrupar; notar que llaves entre comillas '{' '}' son terminales.
$\mid$	separa alternativas.

$\langle$specification$\rangle$ $\rightarrow$ $\langle$enumDecl$\rangle^*$ $\langle$globalVarDecl$\rangle^*$ $\langle$channelDecl$\rangle^*$ $\langle$process$\rangle^+\langle$program$\rangle$

$\langle$enumDecl$\rangle$ $\rightarrow$ **Enum** $\langle$id$\rangle$ = '{' $\langle$idNames$\rangle$ '}' ;

$\langle$idNames$\rangle$ $\rightarrow$ $\langle$id$\rangle$ $\left\{ ,\langle\text{id}\rangle \right\}^*$

$\langle$globalVarDecl$\rangle$ $\rightarrow$ **Global** $\langle$idNames$\rangle$: $\langle$type$\rangle$;

$\langle$channelDecl$\rangle$ $\rightarrow$ **Channel** $\langle$id$\rangle$ '[' $\langle$integer$\rangle$ ']' **of** $\langle$type$\rangle$;

$\langle$process$\rangle$ $\rightarrow$ **Process** $\langle$id$\rangle$ **uses** $\langle$idNames$\rangle$ $\langle$bodyProcess$\rangle$
| **Process** $\langle$id$\rangle$ ($\langle$parameters$\rangle$) **uses** $\langle$idNames$\rangle$ $\langle$bodyProcess$\rangle$
| **Process** $\langle$id$\rangle$ ($\langle$parameters$\rangle$) $\langle$bodyProcess$\rangle$
| **Process** $\langle$id$\rangle$ $\langle$bodyProcess$\rangle$

$\langle$parameters$\rangle$ $\rightarrow$ $\langle$param$\rangle$ $\left\{ ,\langle\text{param}\rangle \right\}^*$

$\langle$param$\rangle$ $\rightarrow$ $\langle$id$\rangle$: $\langle$type$\rangle$

⟨bodyProcess⟩ → '{' ⟨varDecl⟩* ⟨initialCond⟩ ⟨normCond⟩ ⟨branch⟩+'}'

⟨initialCond⟩ → **Initial** : ⟨expr⟩ ;

⟨normCond⟩ → **Normative** : ⟨expr⟩ ;

⟨varDecl⟩ → $\left\{\langle \text{id} \rangle \texttt{,}\right\}^*$ ⟨id⟩ : ⟨type⟩ ;

⟨type⟩ → **INT**
| **BOOL**
| ⟨id⟩

⟨branch⟩ → ⟨expr⟩ `->` ⟨assignation⟩ $\left\{\texttt{,}\langle \text{assignation} \rangle\right\}^*$;

⟨assignation⟩ → ⟨id⟩ `=` ⟨expr⟩
| ⟨id⟩ `.` **put** `(` ⟨expr⟩ `)`

⟨expr⟩ → ⟨expr⟩ `||` ⟨expr⟩
| ⟨expr⟩ `&&` ⟨expr⟩
| ⟨expr⟩ `==` ⟨expr⟩
| ⟨expr⟩ `<` ⟨expr⟩
| ⟨expr⟩ `>` ⟨expr⟩
| ⟨expr⟩ `+` ⟨expr⟩
| ⟨expr⟩ `-` ⟨expr⟩
| ⟨expr⟩ `*` ⟨expr⟩
| ⟨expr⟩ `/` ⟨expr⟩
| `!` ⟨expr⟩
| `-` ⟨expr⟩
| ⟨primary⟩

⟨primary⟩ → ⟨integer⟩
| ⟨true⟩
| ⟨false⟩
| ⟨id⟩
| ⟨id⟩ `.` **get** `(` `)`
| `(` ⟨expr⟩ `)`

$$
\begin{array}{rcl}
\langle \text{integer} \rangle & \rightarrow & \langle \text{digit} \rangle \ \langle \text{digit} \rangle^{*} \\
\langle \text{digit} \rangle & \rightarrow & \texttt{0} \mid \texttt{1} \mid \texttt{2} \mid \ldots \mid \texttt{9} \\
\langle \text{true} \rangle & \rightarrow & \textbf{true} \\
\langle \text{false} \rangle & \rightarrow & \textbf{false} \\
\langle \text{id} \rangle & \rightarrow & \langle \text{alpha} \rangle \ \langle \text{alpha\_num} \rangle^{*} \\
\langle \text{alpha\_num} \rangle & \rightarrow & \langle \text{alpha} \rangle \mid \langle \text{digit} \rangle \mid \_ \\
\langle \text{alpha} \rangle & \rightarrow & \texttt{a} \mid \texttt{b} \mid \ldots \mid \texttt{z} \mid \texttt{A} \mid \texttt{B} \mid \ldots \mid \texttt{Z}
\end{array}
$$

$$
\begin{array}{rcl}
\langle \text{program} \rangle & \rightarrow & \textbf{Main} \ \texttt{(}\,\texttt{)} \ '\{' \ \langle \text{body} \rangle \ '\}' \\
\langle \text{body} \rangle & \rightarrow & \langle \text{processDecl} \rangle^{+} \langle \text{processInvk} \rangle^{+} \\
\langle \text{processDecl} \rangle & \rightarrow & \langle \text{varDecls} \rangle \ \texttt{:} \ \langle \text{id} \rangle \ \texttt{;} \\
\langle \text{processInvk} \rangle & \rightarrow & \textbf{run} \ \langle \text{id} \rangle \ \texttt{(}\,\texttt{)} \ \texttt{;} \\
 & \mid & \textbf{run} \ \langle \text{id} \rangle \ \texttt{(} \ \langle \text{invkParam} \rangle \ \left\{ \texttt{,} \langle \text{invkParam} \rangle \right\}^{*} \ \texttt{)} \ \texttt{;}
\end{array}
$$

$$
\begin{array}{rcl}
\langle \text{invkParam} \rangle & \rightarrow & \langle \text{integer} \rangle \\
 & \mid & \langle \text{true} \rangle \\
 & \mid & \langle \text{false} \rangle \\
 & \mid & \langle \text{id} \rangle
\end{array}
$$

A continuación veremos una especificación de un simple programa con el lenguaje *Faulty* (ver Figura 4.2) dicho programa está compuesto de un único proceso y modela una memoria con redundancia de datos. Las operaciones de escritura se realizan simultáneamente sobre los 3 *bits*, y en el caso de las operaciones de lectura, las mismas retornan el valor que aparece al menos dos veces (este mecanismo es conocido bajo el nombre de *voting*), y una vez leído dicho valor, se reescriben los tres *bits* (ver Sección 5.3 para más detalles de este modelo).

Notar que en este modelo de memoria las fallas están representadas por el hecho de que cualquier *bit* puede *perder* inesperadamente su carga hasta convertirse en 0, lo cual se encuentra especificado por las 3 últimas transiciones del proceso

```
Process Memory {
    w: BOOL; // ultimo valor escrito,
    r: BOOL; // valor obtenido de leer la memoria utilizando voting
    c0: BOOL; // primer bit
    c1: BOOL; // segundo bit
    c2: BOOL; // tercer bit

    Initial: w && c0 && c1 && c2 && r;
    Normative: (c0==c1) && (c1==c2) && (c0==c2);

    true -> w=!w, c0=!c0, c1=!c1, c2=!c2, r =(!c0&&!c1) || (!c1&&!c2) || (!c0&&!c2)
        ;
    true -> c0=!c0, r = (!c0&&c1) || (c1&&c2) || (!c0&&c2);
    true -> c1=!c1, r = (c0&&!c1) || (!c1&&c2) || (c0&&c2);
    true -> c2=!c2, r = (c0&&c1) || (c1&&!c2) || (c0&&!c2);
}

Main(){
    m1: Memory;
    run m1();
}
```

Figura 4.2: Código *Faulty* para una memoria con triple redundancia.

Memory, la cuales se encuentran habilitadas en cualquier instante para ser ejecutadas, modelando de esa manera la aleatoriedad de la falla. En el siguiente capítulo retomaremos este caso de estudio para analizarlo con mayor profundidad, en este punto sólo queremos acercarnos al lenguaje *Faulty* para ir ganando familiaridad con el mismo.

4.1.2. Representación Simbólica de Programas *Faulty*

Es fácil ver que se pueden representar eficientemente la estructura de los programas *Faulty* utilizando estructuras de *Kripke* coloreadas. Sin embargo al crecer el número de procesos y variables, caemos en las limitaciones usuales de *model checking*, por lo cual una representación simbólica aparece como una mejor opción. Y como adelantamos al comienzo de este capítulo se utilizarán *diagramas de decisión Binarios* (BDDs) para su representación. Un programa *Faulty* es representado por un tupla $\langle \{R_i\}, \mathbf{n}, I \rangle$, donde R_i es un conjunto de listas de BDDs, cada uno de los cuales representa un proceso. Intuitivamente, cada R_i representa un proceso, pero en lugar de representarlo con un único BDD, tenemos una lista de BDDs, cada uno capturando los diferentes componentes del mismo. El segundo y tercer componente de la tupla corresponden, respectivamente, a la condición *inicial* y la condición que

caracteriza los estados *normales* del sistema, cada una de ellas tiene un BDD que las codifica. En el caso de las guardas del proceso, una lista de BDDs es la que las representa, donde cada BDD de dicha lista captura una guarda particular. Una extensión interesante que se podría considerar es la de tener reglas modulares, de manera que un proceso pueda encontrarse en un estado de falla mientras otros estén en sus estados *normales*, dejaremos esto para un trabajo futuro.

En lo que se refiere a como construimos el BDD que captura una guarda en particular, asumimos un conjunto de variables *Booleanas* $VAR = \{v_0, \ldots, v_n\}$ y procedemos de la siguiente manera:

Una guarda $G_i \to x = E$ es capturada a través de la siguiente fórmula:

$$[\![G_i]\!] \wedge (x' = [\![E]\!]) \wedge \bigwedge_{y \neq x} y = y',$$

donde $[\![G_i]\!]$ es el BDD que representa la condición de la guarda, $[\![E]\!]$ es el BDD correspondiente a las expresiones de las asignaciones, y las variables y son aquellas que pertenecen a VAR pero que son distintas de x. De hecho, estas variables pueden ser restringidas para que sean solamente aquellas que aparecen en la condición de la guarda (otras variables pueden tratarse sin necesidad de tener que agregarlas de manera explícita a la fórmula).

Sin embargo, un punto importante en este proceso es cómo se computa el conjunto de predecesores de un conjunto de estados dado, lo cual es una operación básica en cualquier algoritmo de *model checking*. En nuestro caso el mismo se calcula de la siguiente manera, utilizando una fórmula *booleana* cuantificada [17]:

$$Prev(S) = \vee_{B_i \text{ is a branch}} (\exists z_0, \ldots, z_m, x'_0, \ldots, x'_k : ([\![B_i]\!] \wedge S'))[z'_i := z_i],$$

donde $[\![B_i]\!]$ es el BDD correspondiente a la guarda B_i, en el caso de S' es la version *primada* del BDD S, el cual captura el conjunto de estados sobre el cual se calcula la *preimagen*. Las variables *primadas* que aparecen involucradas en la expresión $[\![B_i]\!] \wedge S'$ están representadas por $x'_0, \ldots, x'_k$. Todas aquellas otras variables que no aparecen en una guarda dada, las denominamos variables externas a la guarda y son simbolizadas con letras z's. Notar que todas las variables externas conservan sus valores luego de la ejecución de la guarda. En nuestro caso en lugar de agregar las asignaciones correspondientes a estas fórmulas, aplicamos sustituciones, lo cual en términos de las librerías BDD es significativamente más eficiente.

Algo importante de destacar es que los cuantificadores existenciales son ubicados lo más profundo posible dentro de la fórmula, distribuyéndolos con respecto a

la *disjunción*; esta técnica es conocida como *Early Quantification* [105]. En nuestro caso, la estructura de los programas nos permite distribuir los cuantificadores dentro de las guardas. Otro punto importante de hacer notar es que, como se hace usualmente en *model checking* simbólico, no se genera la relación total, mejorando de esta manera la performance de nuestra herramienta.

Además, algo interesante de mencionar es que en todas aquellas guardas que representan fallas, es decir aquellas donde se viola el predicado "normativo", no son tomados en cuenta a la hora de calcular el conjunto de estados previos ($Prev$) para los operadores deónticos, es decir: S sólo va a contener estados correctos y por lo tanto $[\![B_i]\!] \wedge S'$ será inconsistente. Esto mejora el tiempo de ejecución cuando realizamos model checking sobre operadores deónticos.

4.1.3. *Model Checking* para Fórmulas Deónticas

El principal mecanismo detrás del *model checking* para dCTL es un procedimiento recursivo el cual calcula, para cualquier fórmula φ, su correspondiente BDD $[\![\varphi]\!]$ representando aquellos estados que la satisfacen. El mismo es construido calculando el conjunto de todas las subfórmulas de φ, y entonces se verifica cuando la conjunción de I y $[\![\varphi]\!]$ resulta satisfacible.

Centrémonos en el cómputo de aquellos BDDs correspondientes a los operadores Deónticos sin expresiones condicionales, debido a que resultan más simples de presentar. Sin embargo, las definiciones dadas más abajo se extienden directamente a los operadores condicionales (es decir: $\mathbf{O}(\psi \rightsquigarrow \psi')$, $\mathbf{P}(\psi \rightsquigarrow \psi')$ y $\mathbf{R}(\psi \rightsquigarrow \psi')$). Para hacer esto principalmente nos basamos en las semánticas de punto fijo de las fórmulas, y luego utilizamos el algoritmo de *Knaster-Tarski* para computar los mismos. Como ya adelantamos al comienzo de este capítulo asumimos un conocimiento básico acerca de μ-cálculo. Algunas convenciones que tomaremos serán las siguientes:

- μ es el operador de *mínimo* punto fijo.
- ν es el operador de *máximo* punto fijo.
- vamos a utilizar $X, Y, Z, \ldots$ para denotar las variables cuantificadas.
- $\mathsf{EX}\phi$ lo utilizaremos como el operador "diamante" (*eventually*) de la lógica de *Hennessy-Milner*, y similarmente para $\mathsf{AX}\phi$ (operador "box").
- Por cuestiones de simplicidad, utilizaremos la fórmula $\mathsf{EG}(\mathbf{n})$ cuando sea conveniente,en lugar de su obvia traducción a μ-cálculo.

Si algún lector necesita o está interesado en profundizar los contenidos de esta breve introducción a μ-cálculo puede referirse a [118].

$\mathbf{O}(\phi\,\mathcal{U}\,\psi) = \neg(\mu X.\nu Y.(\mathsf{EG}(\mathbf{n}) \wedge \neg\phi \wedge \neg\psi) \vee (\neg\psi \wedge \mathbf{n} \wedge \mathsf{EX}(Y)) \vee (\mathbf{n} \wedge \mathsf{EX}(X)))$,

$\mathbf{O}(\phi\,\mathcal{W}\,\psi) = \neg(\mu Y.(\mu X.(\neg\phi \wedge \neg\psi \wedge \mathsf{EG}(\mathbf{n})) \vee (\neg\psi \wedge \mathbf{n} \wedge \mathsf{EX}(X)) \vee (\mathbf{n} \wedge \mathsf{EX}(Y))))$

$\mathbf{O}(\mathsf{X}\phi) = \neg\mathbf{n} \vee \neg\mathsf{EX}(\neg\phi \wedge \mathsf{EG}(\mathbf{n}))$,

$\mathbf{P}(\phi\,\mathcal{U}\,\psi) = \nu X.\mathbf{n} \wedge (\mu Y.(\psi \wedge \mathsf{EX}(X)) \vee (\phi \wedge \mathsf{EX}(\mathbf{n} \wedge Y))$,

$\mathbf{P}(\phi\,\mathcal{W}\,\psi) = \nu X.\mathbf{n} \wedge (\nu Y.(\psi \wedge \mathsf{EX}(X)) \vee (\phi \wedge \mathsf{EX}(\mathbf{n} \wedge Y)))$,

$\mathbf{P}(\mathsf{X}\psi) = \mathbf{n} \wedge \mathsf{EX}(\psi \wedge \mathsf{EG}(\mathbf{n}))$,

$\mathbf{R}(\phi\,\mathcal{U}\,\psi) = \nu X.(\mathbf{n} \wedge \mathsf{AX}(X)) \vee (\neg\mathbf{n} \wedge (\mu Y.((\psi \wedge \mathsf{AX}(X)) \vee (\phi \wedge \mathsf{AX}(Y)))))$,

$\mathbf{R}(\phi\,\mathcal{W}\,\psi) = \nu X.(\mathbf{n} \wedge \mathsf{AX}(X)) \vee (\neg\mathbf{n} \wedge (\nu Y.((\psi \wedge \mathsf{AX}(X)) \vee (\phi \wedge \mathsf{AX}(Y)))))$,

$\mathbf{R}(\mathsf{X}\phi) = \mathsf{AG}(\mathbf{n} \vee \mathsf{AX}(\phi))$.

Notar que alguna de estas fórmulas no se encuentran en *forma normal positiva* (*positive normal form - PNF*), pero pueden ser fácilmente convertidas en *PNF* utilizando dualidades, colocando las negaciones dentro de los operadores, como se hace usualmente. Además que aparece la fórmula $\mathsf{EG}(\mathbf{n})$ en las definiciones anteriores, intuitivamente la misma es utilizada para indicar que el sistema se encuentra en una ejecución correcta.

Los algoritmos para calcular los BDDs correspondientes a dichas fórmulas pueden ser implementados siguiendo básicamente su semántica de punto fijo. Sin embargo es posible mejorar la manera en la cual calculamos la misma. Por ejemplo, si consideramos la fórmula $\mathbf{O}(\phi\,\mathcal{U}\,\psi)$, utilizando el teorema de *Knaster-Tarski*, podemos calcular el punto fijo aproximando el valor final de X partiendo desde un conjunto más cercano a la solución . Debajo presentamos el algoritmo mejorado para calcular el conjunto de estados que satisfacen dicha fórmula.

En primer lugar si nos detenemos a analizar la definición de $\mathbf{O}(\varphi\,\mathcal{U}\,\psi)$ se puede apreciar que la variable X no aparece en la subfórmula que esta cuantificada por νY. Es decir, X puede ser calculada en un ciclo independiente, y hacerlo sólo cuando se necesitan. Otra mejora que puede realizarse es en la manera de inicializar los valores para calcular los puntos fijos, es decir una implementación clásica cuando se calculan

Algorithm 1 Model Checking para $\mathbf{O}(\phi \,\mathcal{U}\, \psi)$

$Y := \mathsf{EG}(\mathbf{n}) \wedge \neg\phi \wedge \neg\psi$
$Y_{old} := false$
while $Y \neq Y_{old}$ **do**
  $Y_{old} := Y$
  $Y := (\mathsf{EG}(\mathbf{n}) \wedge \neg\phi \wedge \neg\psi) \vee (\mathbf{n} \wedge \neg\psi \wedge \mathsf{EX}(Y))$
end while
$X := Y$
$X_{old} := true$
while $X \neq X_{old}$ **do**
  $X_{old} := X$
  $X := Y \vee (\mathbf{n} \wedge \mathsf{EX}(X))$
end while
return $\neg X$

puntos fijos es donde el conjunto Y se inicializa con el conjunto vacío (es decir representa el predicado $falso$), y X es inicializada con $true$. En nuestro caso esto puede ser mejorado, ya que si analizamos el primer ciclo, podemos notar que estamos calculando un conjunto de ejecuciones normativas de una manera monótamente decreciente, por lo que entonces el conjunto Y puede ser inicializado directamente con $\mathsf{EG}(\mathbf{n}) \wedge \neg\psi \wedge \neg\phi$. Por último en el caso del conjunto X el mismo será un superconjunto del valor final de Y; por lo tanto, X se puede inicializar con Y, mejorando de esta forma el número de iteraciones realizadas por el algoritmo. Más adelante cuando presentemos algunos casos de estudio analizaremos los tiempo de ejecución de este algoritmo.

Estas mismas técnicas se pueden utilizar con el resto de los operadores. Por ejemplo, anteriormente presentamos la manera en la cual se pueden calcular los permisos. En el algoritmo 2, se presenta el método empleado para computar el BDD de la fórmula $\mathbf{P}(\varphi \,\mathcal{U}\, \psi)$.

En este ultimo caso las variables X, Y no se pueden calcular de manera separada, pero X puede ser inicializada con el conjunto de estados *normativos*, ya que los estados donde se satisface la fórmula de permiso seguro es un subconjunto de $norm$. En el caso de la variable Y, un valor inicial más preciso es $X_{old} \wedge \psi$, y esto se puede hacer ya que el conjunto de estados que satisfacen $\mu Y.(\psi \wedge X) \vee (\phi \wedge EX(\mathbf{n} \wedge Y))$ es un subconjunto de aquellos estados que satisfacen la fórmula $X \wedge \psi$ en cualquier iteración del ciclo mas externo. Las fórmulas condicionales son un poco más complejas, por

Algorithm 2 Model Checking $\mathbf{P}(\varphi\,\mathcal{U}\,\psi)$

$X := \mathbf{n}$
$X_{old} := false$
while $X \neq X_{old}$ **do**
  $X_{old} \neq X$
  $Y := X \wedge \psi$
  $Y_{old} := true$
  while $Y := Y_{old}$ **do**
    $Y_{old} := Y$
    $Y := Y \vee (\phi \wedge EX(Y \wedge \mathbf{n}))$
  end while
  $X := norm \wedge Y$
end while
return X

ejemplo la semántica en μ-cálculo para la fórmula $\mathbf{P}(\phi\ \mathcal{U}\ \psi \rightsquigarrow \phi'\ \mathcal{U}\ \psi')$ es la siguiente:

$$\begin{aligned}\mathbf{P}(\phi\,\mathcal{U}\,\psi \rightsquigarrow \phi'\,\mathcal{U}\,\psi') = \quad & \nu Y.\mathbf{n} \wedge (\nu X.(\neg\phi \wedge \neg\psi \wedge \mathsf{EX}(Y)) \\ & \vee(\neg\phi \wedge \mathsf{EX}(\mathbf{n} \wedge X))) \\ & \vee(\mu Z.(\psi' \wedge \mathsf{EX}(Y)) \\ & \vee(\phi' \wedge \mathsf{EX}(\mathbf{n} \wedge Z)))\end{aligned}$$

Ésta fórmula puede ser calculada de una forma similar a lo que hicimos anteriormente utilizando ciclos, y optimizaciones equivalentes pueden realizarse en el correspondiente código.

4.1.4. dCTL vs CTL

Como ya expresamos con anterioridad en el capítulo previo, algunas fórmulas de dCTL no puede ser traducidas a CTL [29]; sin embargo, hay muchas otras fórmulas de dCTL que si admiten un traducción a CTL, esto resulta muy útil a la hora de comparar la performance del algoritmo definido anteriormente con respecto a otros *model checkers*. Por ejemplo, en el caso de la obligación la misma puede ser traducida a CTL de la siguiente forma:

$$\mathbf{O}(\phi\,\mathcal{U}\,\psi) = \neg\mathsf{E}(\mathbf{n}\,\mathcal{U}\,\mathsf{E}(\neg\psi \wedge \mathbf{n}\,\mathcal{W}\,\neg\psi \wedge \neg\phi \wedge \mathsf{EG}(\mathbf{n})))$$

$$\mathbf{O}(\phi \mathcal{W} \psi) = \neg \mathsf{E}(\mathbf{n} \mathcal{U} \mathsf{E}(\neg\psi \wedge \mathbf{n} \mathcal{U} \neg\phi \wedge \neg\psi \wedge \mathsf{EG}(\mathbf{n})))$$

$$\mathbf{O}(\mathsf{X}\phi) = \neg \mathsf{EF}(\mathbf{n} \wedge \mathsf{EX}(\neg\phi \wedge \mathsf{EG}(\mathbf{n})))$$

A la hora de traducir fórmulas que contienen obligaciones condicionales se obtienen fórmulas CTL bastantes largas. Esta es una de las razones por la que consideramos que la lógica dCTL mejora la *claridad* de propiedades que predican acerca de tolerancia a fallas. Por ejemplo la traducción de $O(\phi \mathcal{U} \psi \rightsquigarrow \phi' \mathcal{U} \psi')$ es la siguiente fórmula CTL:

$$\begin{array}{rl} = & \neg \mathsf{E}(\mathbf{n} \mathcal{U} \mathsf{E}(\phi \wedge \neg\psi' \mathcal{U} \psi \wedge \mathsf{E}(\neg\psi' \wedge \mathbf{n} \mathcal{W} \neg\psi' \wedge \phi' \wedge \mathsf{EG}(\mathbf{n})))) \\ & \wedge \neg \mathsf{E}(\mathbf{n} \mathcal{U} \mathsf{E}(\neg\psi' \wedge \mathbf{n} \mathcal{U} \neg\psi' \wedge \neg\phi' \wedge \mathsf{E}(\phi \wedge \mathbf{n} \mathcal{U} \psi \wedge \mathsf{EG}(\mathbf{n})))) \end{array}$$

Algo para hacer notar es que no existe una traducción obvia para fórmulas que involucran el operador de permiso, debido a que el mismo se basa en el anidamiento de dos puntos fijos, el cual se traduce en μ-cálculo en alternancia 1, mientras que las fórmulas CTL son traducidas en la jerarquía de μ-cálculo *libre de alternancia*.

En el siguiente capítulo presentamos algunos ejemplos en los que tenemos modelos de fórmulas de permiso verificadas en un tiempo aceptable para estructuras grandes (más de 2^{32} estados); es decir, estas fórmulas aumentan la expresividad de la lógica sin afectar seriamente la eficacia del algoritmo de verificación de modelos en la práctica.

4.2. FaultyCheck

En esta sección presentaremos brevemente el diseño adoptado para dicho *model checker*. Sin embargo, si el usuario desea analizar con más detalles la implementación realizada puede encontrar todo el código fuente y resto de la documentación en el siguiente repositorio [1].

La Figura 4.3 representa la arquitectura básica de *FaultyCheck*, tal como se puede apreciar en el diagrama, nuestra herramienta de verificación recibe como entrada el modelo del sistema especificado en el lenguaje *Faulty* y la fórmula que codifica la propiedad que se desea verificar. Luego mediante un proceso de parseo de dichas entradas se las traduce en la representación simbólica elegida; en nuestro caso, como explicamos recientemente, se contruyen una serie de *Diagramas de Decisión Binarios* (BDD) para dicho fin. Estas estructuras simbólicas son las que constituyen las entradas del algoritmo de *model checking*, el cual a través de los operadores de punto fijo determina si la propiedad se satisface o no.

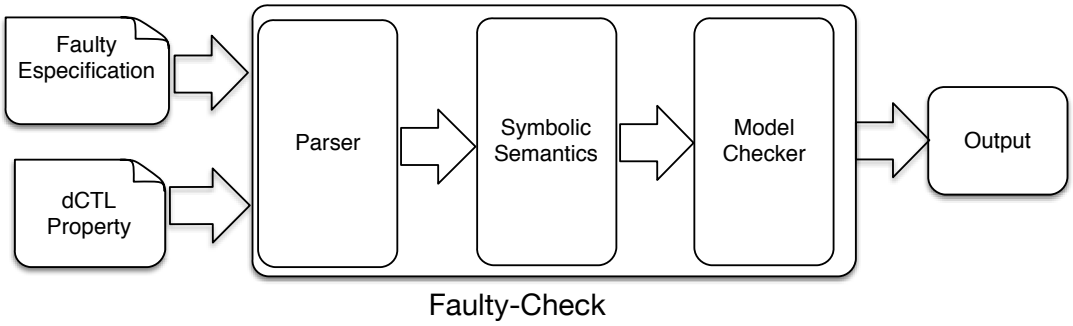


Figura 4.3: Arquitectura de *FaultyCheck*

En lo que se refiere al diseño para separar la lógica de la estructura utilizamos el patrón *Visitor* [90]. En la siguiente Figura 4.4 presentamos la estructura básica de como implementamos este patrón.

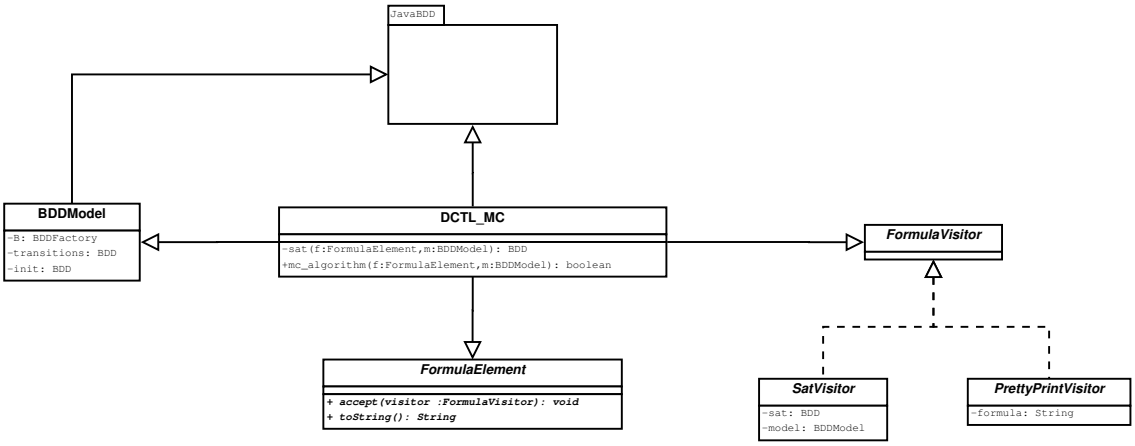


Figura 4.4: Diagrama de Clases (Patrón Visitor)

El uso del patrón *visitor* permite que cada visitante de las fórmulas concretas puedan calcular *Sat* de manera individual, delegando el calculo de las súbformulas a los visitantes de las mismas, simplificando asi la lógica de todo el algoritmo y despegando la implementación del algoritmo de Sat de la jerarquía de *FormulaElement*. La jerarquía de clases utilizada para representar las fórmulas dCTL se puede apreciar en la Figura 4.5.

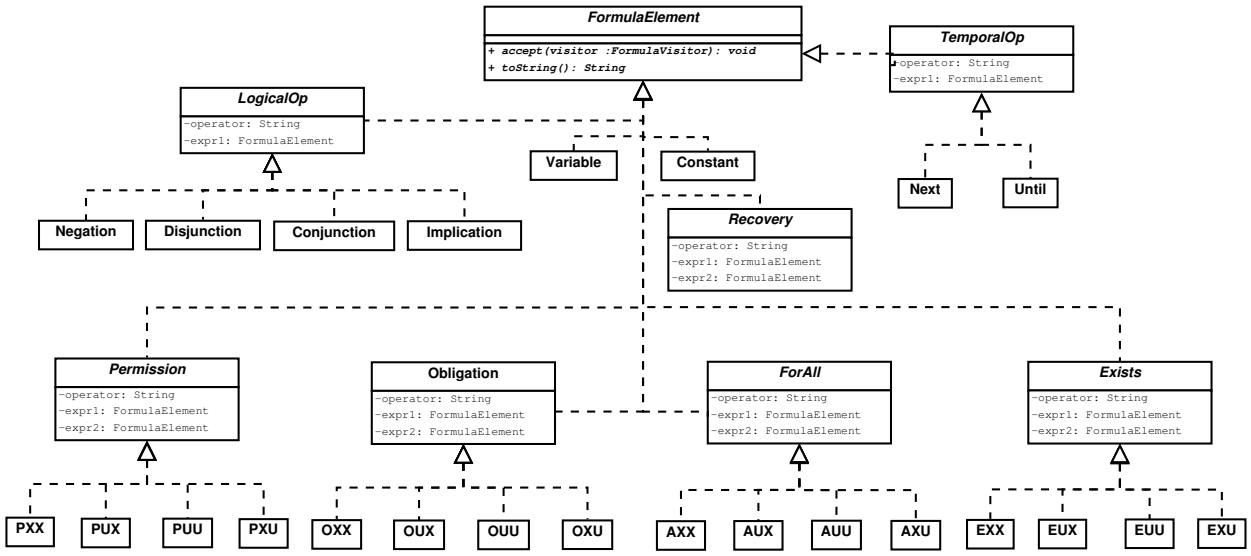


Figura 4.5: Diagrama de Clases Fórmulas.

Por último la jerarquía de clases que implementa el lenguaje de guardas se encuentra reflejada en la Figura 4.6.

4.3. Resumen del capítulo

En este capítulo hemos presentado las principales características del algoritmo de *model checking* de dCTL.

Enfocándonos principalmente en los diferentes actores que intervienen en este proceso de verificación.

En primer lugar introdujimos un simple lenguaje de guardas, denominado *Faulty*, el cual es utilizado para especificar los modelos de sistemas tolerantes a fallas de una manera simple, permitiendo discriminar las propiedades que se preservan cuando el programa no presenta fallas.

Luego nos abocamos a detallar la manera adoptada a la hora de representar eficientemente la estructura de los programas *Faulty*, lo cual se lleva a cabo utilizando una serie de BDDs, estos permiten una representación simbólica de las distintas partes que componen la especificación del programa dado. Cabe destacar que en el caso de los cuantificadores existenciales utilizamos una técnica conocida como *Early Quantification* [105], en la cual estos cuantificadores son ubicados lo mas

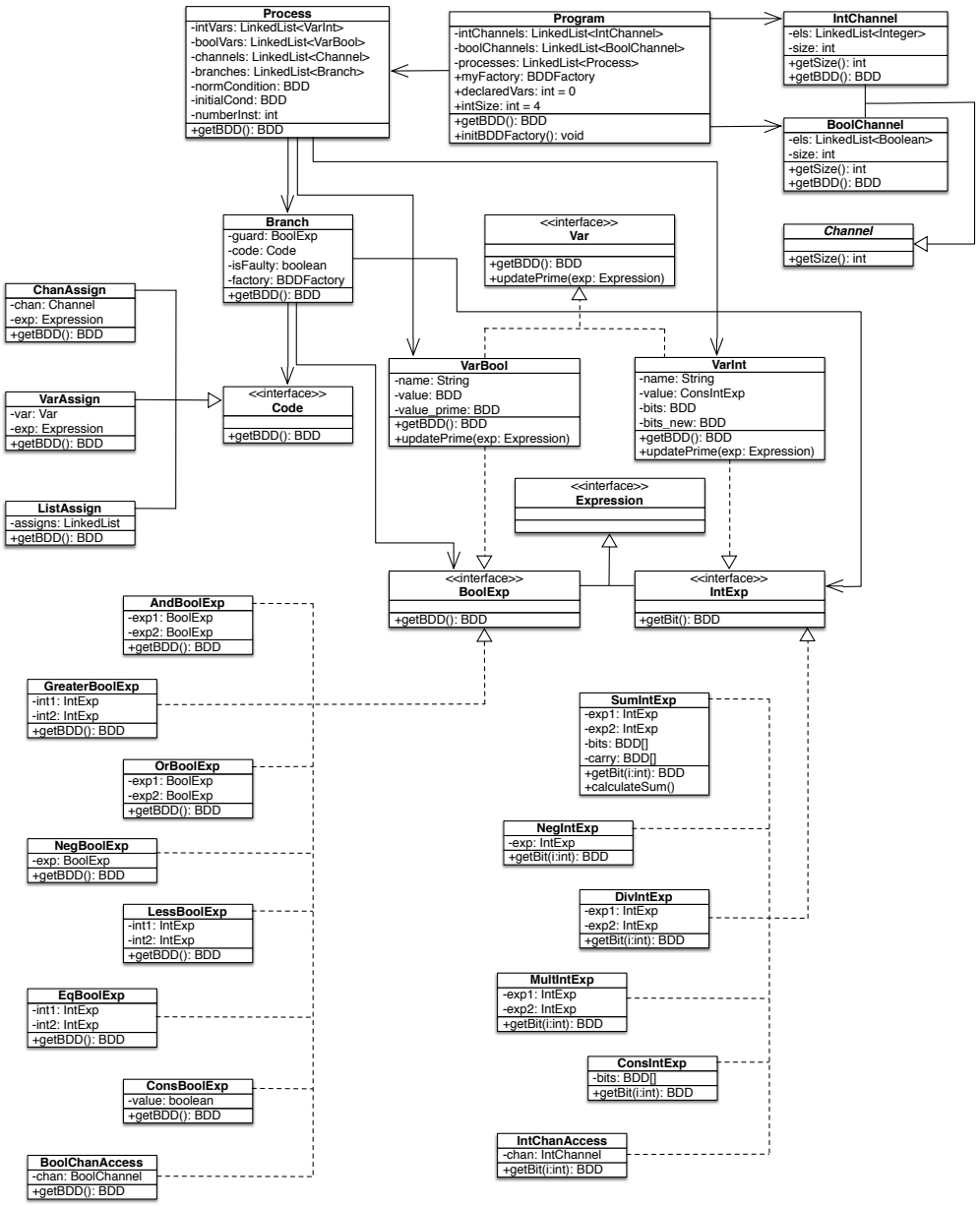


Figura 4.6: Diagrama de Clases del Lenguaje Faulty.

profundo posible dentro de la fórmula, distribuyéndolos con respecto a la *disjunción*. En nuestro caso, la estructura de los programas *Faulty* nos permite distribuirlos

dentro de las guardas.

Por último presentamos el algoritmo de *model checking* para dCTL, el cual es un procedimiento recursivo que calcula para cualquier fórmula φ, su correspondiente BDD $[\![\varphi]\!]$ representando aquellos estados que la satisfacen y luego verifica cuando la conjunción de los estados iniciales I y $[\![\varphi]\!]$ resulta satisfacible. Se dieron las definiciones para calcular cada fórmula de acuerdo a los operadores que involucra, sin embargo nos detuvimos mayormente a explicar la manera de computar aquellos BDDs correspondientes a los operadores Deónticos. Lo cual se basa principalmente en las semánticas de punto fijo de las fórmulas. Por otro lado cabe recalcar que además de dar las definiciones clásicas para calcular dichos operadores, también presentamos algunas variantes con el objetivo de mejorar la performance de cada algoritmo, las cuales se basan principalmente en la manera de inicializar los conjuntos utilizados para calcular los diferentes puntos fijos de acuerdo al operador, como así también siempre que es posible, calcular en ciclos independientes, y bajo demanda (es decir hacerlo sólo cuando se necesitan) aquellas fórmulas que se encuentran libres, es decir fuera del alcance de algún cuantificador.

Finalmente se describieron brevemente algunos detalles de la implementación de este algoritmo, el cual fue realizado utilizando JAVA, con el objetivo de poder evaluar a nivel práctico la *performance* de los algoritmos desarrollados.

Capítulo 5

Casos de Estudio

En los capítulos previos hemos presentado nuestra lógica y demostrado sus propiedades y características desde un punto de vista más teórico. En este capítulo, nos abocaremos a exhibir, mediante algunos casos de estudio, cómo dCTL resulta adecuada para expresar propiedades de sistemas en los cuales pueden ocurrir fallas, y cómo estos operadores deónticos permiten capturar algunas situaciones que se presentan en este tipo de sistemas.

Comenzaremos describiendo los casos de estudio que se analizaron, algunos a pesar de ser bastantes sencillos y con modelos con un número reducido de estados y transiciones, resultan muy interesantes para mostrar y analizar la clase de propiedades y/o patrones que permite caracterizar este nuevo formalismo lógico. Hacia el final del capítulo utilizaremos algunos de estos ejemplos para comparar la *performance* de nuestra herramienta de *model checking* con respecto a NuSMV, el cual es un reconocido model checker simbólico para CTL, y así tener un primer punto de comparación para analizar la eficiencia práctica del algoritmo implementado.

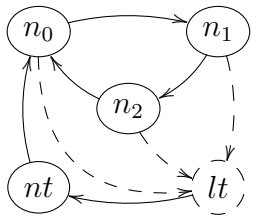
5.1. Un protocolo muy simple: *Token Ring*

Veamos un pequeño ejemplo, en este caso un simple protocolo de comunicación. Supongamos que tenemos una pequeña red compuesta por tres nodos conectados utilizando de la topología de anillo (*ring*), y donde toda comunicación esta regulada a través del protocolo *token ring*. En el sistema original un *token* es pasado a través de los nodos, de manera tal que el nodo que posee el *token* en un momento determinado, tiene acceso a un recurso particular. Por ejemplo: permiso para enviar un mensaje a través de la red. Una vez finalizado su turno, el nodo que posee el *token* lo envía

al nodo vecino. Es fácil imaginar algunas propiedades deseables para este sistema. Por ejemplo, algunos requerimientos podrían ser:

- *Que en todo momento exista exactamente un nodo que tenga asignado el token.*
- *Si un nodo tiene asignado el token, que eventualmente el mismo pase al próximo nodo del anillo.*

Una simple falla que podríamos concebir en este contexto, es una en la cual debido a que el medio de comunicación no es seguro, el token puede "*perderse*"mientras es pasado de un nodo a otro. Si se fija el período de tiempo que cada nodo puede tener asignado dicho *token*, entonces es fácil implementar un mecanismo para detectar esta falla. Ya que se podría calcular el tiempo máximo que se puede esperar por el token (*timeout*), y en este caso: si un nodo no ha recibido el *token*, por un tiempo mayor al máximo establecido para cada nodo, multiplicado por el número total de nodos, entonces significa que ha ocurrido una falla y el token se ha perdido.



n_i: Nodo i tiene el token
lt: Token perdido
nt: Nuevo token creado

Figura 5.1: Modelo de una red Token ring, donde los tokens pueden perderse.

Una abstracción de esta situación, incluido este mecanismo de detección de fallas y un modo de recuperacón de la misma está representada en la Figura 5.1. Los estados n_i representan la situación en donde el *token* se encuentra en el nodo i; en el caso que el *token* se pierda (nodo lt) y que el mecanismo de detección alertó de ésto, un nuevo *token* es creado (nodo nt) y asignado por defecto al nodo n_0 de la red. Notar que en este gráfico los estados y las transiciones con fallas se encuentran graficadas con líneas de puntos, en lugar de utilizar colores para representarlos.

Algunos requerimientos de este sistema pueden especificarse utilizando CTL, a través de las siguientes fórmulas:

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

donde $\oplus$ es la operación de *suma* módulo 3. Notar también que por simplicidad hemos asumido que cada nodo tiene el token por exactamente un instante (en el siguiente paso, el *token* pertenece al siguiente nodo de la topología). Si uno quiere chequear que estas propiedades del sistema original se siguen cumpliendo en nuestro modelo, en el caso que no ocurra ninguna falla, las mismas podrían expresarse usando dCTL de manera muy similar al ejemplo anterior, mediante el uso del operador *Obligación* :

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

Notar que en el caso del segundo requerimiento del sistema original, a pesar de cumplirse en nuestro modelo en la ausencia de fallas, es *falso* en cualquier escenario donde ocurra al menos una falla. Por lo que este es un caso donde es necesario *debilitar* dicha propiedad para que pueda llegar a satisfacerse aún ante la presencia de fallas. El requerimiento de que el *token* debe ser pasado al próximo nodo de la red en el *siguiente* instante se puede debilitar, pidiendo ahora que el *token* sea pasado en *algún instante futuro* al próximo nodo. Formalmente lo podríamos expresar de la siguiente manera:

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

Algo para remarcar es que esta es una propiedad de *progreso*, ya que la misma garantiza que nuestro sistema se comporte de una manera "*justa*"(*fair*). Notar que a pesar de que *strong fairness* no se puede expresar en CTL, varios *model checkers* incorporan la misma para la verificación de propiedades de *liveness*, y nuestro caso al ser una propiedad de *progreso*, un caso particular de *liveness*, también lo podemos hacer.

Otra propiedad de tolerancia a falla que resulta interesante en este caso, puede ser aquella que expresa que las fallas son las únicas responsables de que el *token* se pierda. En otras palabras, si el *token* se encuentra en un nodo particular n_i, sólo pueden ocurrir dos cosas: o el *token* es pasado al próximo nodo ó se produce una falla. Esto puede se situación se puede capturar en dCTL de la siguiente manera:

$$\mathsf{AG}(n_i \rightarrow \mathsf{X}(\neg \mathbf{P}(\top) \vee n_{i\oplus 1}))$$

5.2. Algunas características de dCTL

Con estas simples propiedades presentadas en la sección previa, intentamos mostrar algunos ejemplos de propiedades que comunmente resultan de interés para sistemas tolerantes a fallas, como por ejemplo cual es el comportamiento deseado ante la ausencia de fallas, y que en nuestra opinión, pueden ser expresadas naturalmente en dCTL. Otras propiedades más generales de tolerancia a fallas, como es el caso de propiedades de *closure* y *convergencia*, tal como se describen en [15], también pueden ser expresadas de manera directa. Veamos un poco más de las mismas.

Closure

Closure sirve para expresar que dada una fórmula de *estado* ϕ, la cual caracteriza un requisito del sistema, se preserva *inductivamente* en las transiciones sin falla. En dCTL esto puede ser expresado como :

$$\mathbf{O}(\phi \rightsquigarrow \mathsf{X}(\phi))$$

Convergencia

En el caso de convergencia, por otro lado, permite expresar que desde cualquier estado que satisface cierta propiedad φ' (por ejemplo: indicando que un sistema se encuentra en un estado *anormal*), si no ocurren mas fallas, entonces el sistema siempre puede retornar a un estado donde se satisface la propiedad φ (por ejemplo: eventualmente se recupera de la falla.). Este tipo de propiedades pueden expresarse en dos sub-propiedades:

- *Desde cualquier estado normal, donde se cumpla la propiedad φ', si el sistema se mueve únicamente a través de transiciones normales, entonces eventualmente alcanzará un estado donde vale φ*, es decir: $\mathbf{O}(\varphi' \rightsquigarrow \mathsf{AF}(\varphi))$
- *Sea cuando sea que ocurran las fallas, si el sistema se mantiene moviéndose a través de estados normales o estados donde se cumple la propiedad φ', entonces eventualmente alcanzará un estado en el cual es verdadero:* $\mathbf{R}(\mathsf{G}(\varphi' \vee n) \rightsquigarrow \mathsf{F}\varphi)$. Aquí n se cumple en los estados sin falla y puede ser definido como $n = \mathbf{P}(\top)$.

Notar que muchas de las propiedades que presentamos a lo largo de esta tesis pueden ser pensadas como variantes de estos dos conceptos.

5.3. Celda de Memoria

Consideremos un sistema que modela una celda de memoria, la cual almacena simplemente un *bit* de información y soporta operaciones de *lectura* y *escritura*. Este simple sistema puede representarse mediante una estructura de Kripke como la que se describe en la Figura 5.2, donde cada estado mantiene el valor corriente almacenado en la celda de memoria (m_i, para $i = 0, 1$). Obviamente, en este sistema el valor que resulta de la lectura depende totalmente del valor almacenado en la celda.

Por lo que una primer propiedad que uno podría desear para este sistema es que el valor *leído* coincida con el valor de la última operación de *escritura* realizada en el sistema. Esta propiedad puede especificarse de manera muy sencilla en CTL, a través de la siguiente fórmula:

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

Donde m_0 y m_1 representan que el valor *leído* de la celda fue un 0 ó un 1, respectivamente. En el caso de las variables w_0 y w_1, las mismas indican que el último valor *escrito* en la celda fue un 0 ó un 1, respectivamente.

Esto puede ser considerado parte de la *especificación* de los *requerimientos* que la implementación descripta en la Figura 5.2 debería satisfacer. Como se puede apreciar, este sistema describe el comportamiento *ideal* del sistema, ya que no tiene en cuenta *fallas* de ninguna clase, por lo que, tal cual como está este modelo, no justifica el uso de operadores deónticos y no nos permitiría exhibir la ventaja de nuestro enfoque. Veamos entonces una variante de este modelo donde pueden ocurrir algunas fallas.

5.3.1. Una variante con Fallas

Supongamos ahora que cuando el valor de un *bit* es 1, el mismo puede *perder* inesperadamente su carga hasta convertirse en 0. En este caso, la implementación del modelo original, no puede garantizar que la propiedad deseada se cumpla, dado que puede darse el escenario donde después de escribir un 1 en la celda, inmediatamente ocurra la falla descripta y si una operación de lectura es realizada en ese momento, se leerá el valor 0, siendo que el último valor escrito había sido un 1.

Por lo que el modelo presentado anteriormente (Figura 5.2), necesita ser modificado para poder contemplar la ocurrencia de la falla descripta y a su vez garantizar

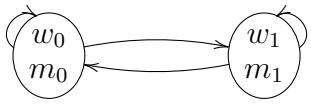


Figura 5.2: Un simple modelo de una celda de memoria (sin fallas).

la propiedad deseada. Como vimos en el capítulo 2, un típico mecanismo, para sobrellevar este tipo de situaciones en tolerancia a fallas, es través de *redundancia*. Por ejemplo, en este caso podríamos decidir implementar el mismo sistema, pero ahora utilizando 3 *bits* de memoria, en lugar de 1.

Algunas consideraciones que vamos a asumir en este nuevo modelo:

- Las operaciones de escritura se realizan simultáneamente sobre los 3 *bits*.
- Las operaciones de lectura retornan el valor que aparece al menos dos veces (este mecanismo es conocido bajo el nombre de *voting*), y una vez leído dicho valor, lo reescriben en los tres *bits*.
- Las operaciones de lectura y escritura nunca fallan.

El modelo resultante se puede apreciar en la Figura 5.3.
Cada estado de este modelo está compuesto de las siguiente variables:

- w_i : almacena cual fué la última operación de escritura realizada (w_0 significa que la última vez se *escribió* un cero y w_1 representa la escritura de un uno).
- b_0, b_1 y b_2: estas variables representan el valor de los tres *bits*.

La ocurrencia de una falla, la cual cambia el valor de un *bit* de 1 a 0, está representada por lineas punteadas en la figura. En el caso de los estados *anormales* o estados de *falla* (representados con círculos de líneas punteadas) como es usual, son aquellos estados que se alcanzan luego de la ocurrencia de una falla, en este caso en particular son aquellos estados donde se va *degradando* el valor original de la memoria, hasta convertirse totalmente en 0. Las líneas continuas denotan transiciones *normales* entre estados, representando las operaciones de *lectura* o *escritura*.

Notar que la operación de lectura necesita ser redefinida, en la presencia de redundancia; ya que el valor leído es el valor que más veces se repite, entonces la lectura del valor 1, puede expresarse a través de la siguiente fórmula lógica:

$$r_1 = (b_0 \wedge b_1) \vee (b_0 \wedge b_2) \vee (b_1 \wedge b_2)$$

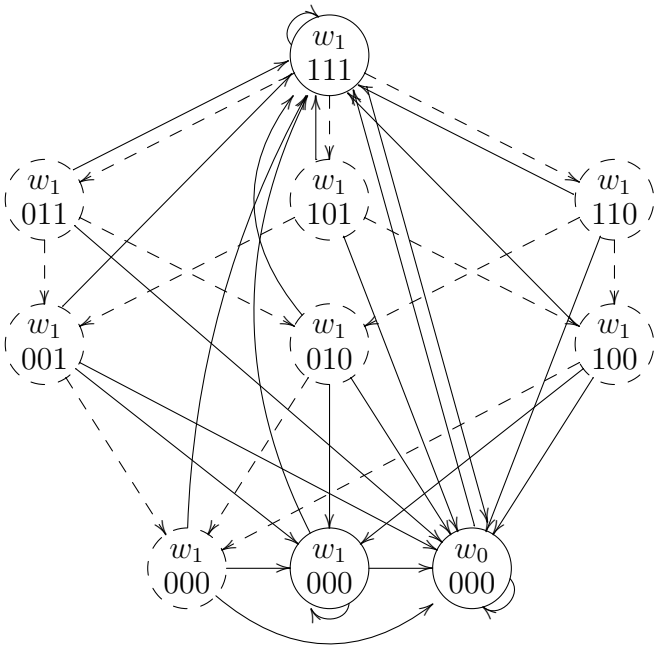


Figura 5.3: Modelo de la celda de memoria, con fallas

Es decir, que el valor leído es un 1 si al menos hay dos *bits* con valor 1 de la celda de memoria con redundancia. Teniendo r_1 definido, podemos definir r_0 simplemente como su negación.

Respecto a las propiedades que uno *esperaría* que este nuevo modelo de celda de memoria tuviera, es fácil ver que los requerimientos de nuestro modelo original necesitan ser adaptados para predicar acerca de esta nueva implementación, por lo que podríamos redefinirla de la siguiente manera:

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

Donde el valor almacenado m_i es ahora reemplazado por la lectura del valor r_i, tal como lo definimos previamente. Por supuesto, que en el caso que ocurra una falla, esta propiedad no se cumple en el nuevo modelo. De todas maneras, no deja de ser interesante intentar verificarla, por ejemplo a través de *model checking*, se podrían obtener *contraejemplos* que nos ayuden a entender en que situaciones nuestro modelo viola los requerimientos y los escenarios en los cuales pueden producirse las *fallas*.

Por lo cual una propiedad obvia que uno podría estar interesado en garantizar es que ante la ausencia de fallas, se cumpla el requerimiento original. La misma se podría pensar como forma de verificar que el mecanismo de tolerancia a falla,

incorporado al modelo original, no afecta la validez de los requerimientos, cuando no hay fallas presentes. Esta primera propiedad de tolerancia a fallas puede ser expresada naturalmente utilizando el operador de *obligación*, de la siguiente manera:

$$\mathbf{O}((r_0 \rightarrow w_0) \wedge (r_1 \rightarrow w_1))$$

Notar que otra manera de expresar esta misma propiedad es la siguiente: $\mathbf{O}(r \equiv w)$, donde r es el valor de lectura de los 3 *bits* y w representa el último valor escrito.

Propiedades en escenarios con fallas

Pero si ahora quisieramos predicar acerca de propiedades presentes en los escenarios donde hay fallas, el operador de *recuperación* aparece como un candidato natural para expresar las mismas. La motivación para la incorporación de mecanismos de tolerancia a falla, es que el sistema pueda seguir comportándose de manera *correcta* aún ante la presencia de fallas. Por supuesto, que esto no puede garantizarse para todo escenario con fallas, por lo que este invariante general que teniamos originalmente se convertirá en un invariante *condicional*, el cual se cumplirá bajo ciertas condiciones sobre las fallas.

Por ejemplo, para la celda de memoria con redundancia, podríamos querer decir que "*El valor leído coincide con el valor escrito, aún ante la presencia de fallas, siempre y cuando una vez que se produzca una falla no ocurre ninguna más, hasta que una operación de lectura o escritura sea realizada (las cuales restablecen el valor)*". Utilizando dCTL, esto podría expresarse como sigue:

$$\mathbf{R}((\textsf{not-too-broken}\ \mathcal{U}\ \textsf{bits-coincide}) \rightsquigarrow (r_i \rightarrow w_i))$$

donde $\textsf{not-too-broken} = n \vee r_1$ (a lo sumo una falla ha ocurrido desde la última operación de lectura/escritura), y $\textsf{bits-coincide} = (b_0 \leftrightarrow b_1) \wedge (b_0 \leftrightarrow b_2)$ (captura la situación en la cual los tres *bits* coinciden, siempre como una consecuencia de una lectura o escritura). Notar que la subfórmula a la derecha de $\rightsquigarrow$ no se encuentra restringida al comportamiento *normal*, ya que no utiliza operadores de *obligación* o *permiso*. Pero en el caso de la fórmula que se encuentra a la izquierda del $\rightsquigarrow$, la misma restringe que debería ocurrir cuando una falla ocurre; es decir que lo que queremos capturar es que: sea cuando sea que ocurran las fallas, el sistema debería *visitar* transiciones *normales*, hasta que una operación de *lectura* ó *escritura* sea realizada. Por último, este también emplea la expresión $n = \mathbf{P}(\top)$, la cual expresa que el estado *actual* es un estado *normal* (Recordar que tenemos la restricción que

dice que en toda estructura de Kripke los estados normales siempre tienen al menos un estado sucesor que también lo es).

La fórmula anterior, es un ejemplo de cómo se puede utilizar el operador de *recuperación*. Pero algo importante para destacar es que el patrón $\mathbf{R}(\psi \rightsquigarrow \phi)$ resulta útil para expresar el nivel de tolerancia a falla deseado; ya que el mismo permite capturar la situación en la donde se garantiza que la fórmula ϕ es verdadera, incluso ante la presencia de fallas, y que el sistema se comporta como ψ indica, independientemente del momento en que ocurran las fallas. Por lo que de alguna manera nos permite especificar los requerimientos mínimos que deberían cumplirse en nuestro sistema ante la presencia de fallas. Surgen varias preguntas de ingeniería de software interesantes relacionadas con este *patrón*, las cuales cabe aclarar que no se abordan en esta tesis, por ejemplo: dada la fórmula de estado ϕ, uno podría estar interesado en *sintetizar* la fórmula ψ mas débil, tal que el *patrón* se cumpla.

Otra propiedad interesante que dCTL nos permite expresar es acerca de cuantas fallas consecutivas puede *tolerar* nuestro sistema una vez que ingreso en un modo de falla; por ejemplo podemos querer expresar lo siguiente: si una falla acaba de ocurrir (tal vez inmediatamente después de otra falla), el sistema siempre tiene la posibilidad de comportarse de manera tal que los requerimientos del sistema sean reestablecidos. En este caso un sistema que cumple con esta propiedad estaría garantizando que a pesar de que ocurran fallas, siempre existe un *futuro bueno* donde podrá proveer el comportamiento esperado. Esto se puede expresar naturalmente utilizando el operador de *permiso*, de la siguiente manera:

$$\mathsf{AG}(\neg n \wedge \mathsf{EX} n \rightsquigarrow \mathsf{EG}\phi)$$

La fórmula $\neg n \wedge \mathsf{EX} n$ captura la propiedad de que un estado sea *fallido*, pero en el cual el sistema se puede recuperar. Notar que esta manera de "evitar" el estado de *error* se cumple en nuestro modelo, ya que las operaciones de lectura y escritura, siempre reestablecen los valores almacenados en la celda de memoria, lo cual justamente nos vuelve a un estado en el cual se cumplen los requerimientos originales del sistema.

Dejaremos planteadas algunas propiedades que resultan interesantes para ser verificadas en las secciones futuras sobre este modelo :

P1. $\mathbf{O}(r \equiv w)$, *ante la ausencia de fallas, el valor del bit leído coincide con el último valor de bit escrito*,

P2. $\mathbf{R}(r\ \mathcal{U}\ \mathbf{n} \rightsquigarrow (r \equiv w)\ \mathcal{U}\ \mathbf{n})$,

luego de la ocurrencia de una falla, si la operación de lectura lee un 1 *hasta el momento que el sistema regrese a un estado correcto, entonces el valor leído es correcto.*

P1 asegura el comportamiento correcto de la memoria ante la ausencia de fallas, mientras que la propiedad **P2** establece que esta implementación tolera hasta la ocurrencia de una falla, es fácil ver que mientras más cantidad de bits se repliquen, mayor cantidad de fallas serán toleradas.

5.4. Muller C-element

Este caso tomamos un simple circuito conocido como "*Muller C-Element*"[126], el cual ha jugado un rol muy importante en el desarrollo de circuitos asincrónicos y curiosamente ha sido utilizado en la implementación de la *Unidad Aritmética y Lógica* (ALU) de la computadora *ILLIAC II* [74].

En la versión original el circuito recibe dos entradas *Booleanas* y produce una salida, también de tipo lógico. La salida será *verdadera* cuando ambas entradas también lo sean, y será *falsa* cuando al menos una de las entradas sea *falsa*. Lo interesante de este circuito es que el valor de salida se mantiene hasta que ambas entradas cambien su valor, es decir que cambien de estado, este tipo de circuito se conocen con el nombre de circuitos "*delay-insensitive*". Notar que esto puede ser generalizado para recibir un mayor número de entradas componiendo varios circuitos de esta clase. En [15] se puede encontrar una intuitiva descripción de este caso de estudio.

A lo largo de los años se han definido variantes de este circuito, una de ellas agrega una tercera entrada y luego lo implementa utilizando una especie de *voting* (en alguna bibliografía también se puede encontrar este ejemplo bajo el nombre de *majority circuit*). Para entender mejor este tipo de implementación veremos un ejemplo del mismo y que tipo de propiedades resultan interesante en este modelo. Llamaremos "z" a la salida del circuito. Las entradas del circuito original estarán representadas por las variables "x" e "y". Y la entrada extra del circuito la denotaremos como "u".

El predicado $maj(x, y, u)$ retorna el valor que más veces aparece y suponemos que siempre funciona correctamente. Formalmente lo podemos definir de la siguiente forma:

$$maj(x, y, u) \Leftrightarrow (x \wedge y) \vee (x \wedge u) \vee (y \wedge u)$$

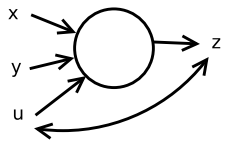


Figura 5.4: Circuito implementado con *majority*.

5.4.1. Fallas de este modelo

En este escenario podríamos considerar 2 tipos de fallas, las cuales van a estar representadas por las constantes v_1 y v_2, para ayudarnos a especificarlas. Podemos definir cada una de ellas de la siguiente forma:

1. $(x \equiv z) \wedge (y \equiv z) \rightarrow \neg v_1$.
2. $u \equiv z \rightarrow \neg v_2$.

La primera fórmula se refiere al momento en cuando "z" , "x" e "y" coinciden, entonces no hay violación de tipo v_1. Es decir que v_1, representa una falla en la sincronización de los valores de entrada.

En el segundo caso se expresa que cuando "u" y "z" tienen el mismo valor, entonces no hay violación de tipo v_2, representando la falla en la retroalimentación de la salida a la entrada adicional.

Notar que esta implementación tolera *delays* en las entradas "x" e "y", pero no tolera *delays* en la entrada "u" . En la Figura 5.5 se representa el modelo de dicho circuito para 2 entradas, para ayudar al lector se etiqueraron los estados del mismo con las constantes v_1 y v_2 para indicar aquellos estados donde se produce cada tipo de violación, sin embargo las mismas no forman parte de la especificación del sistema. Cada estado está determinado por el valor de las variables "x", "y", "u" y "z", por ejemplo el estado 101/0 representa el estado del circuito donde $x = 1$, $y = 0$, $u = 1$ y $z = 0$, respectivamente. En los estados iniciales de dicho modelo se cumple la siguiente propiedad:

$$(x \equiv y) \wedge (y \equiv z) \wedge (u \equiv z) \wedge \neg v_1 \wedge \neg v_2$$

Muller C-Element versión restringida

En la versión anterior si no hay *delay* en la retroalimentación entre "z" y "u" el sistema debería ser *masking tolerant*. En el siguiente modelo asumimos justamente

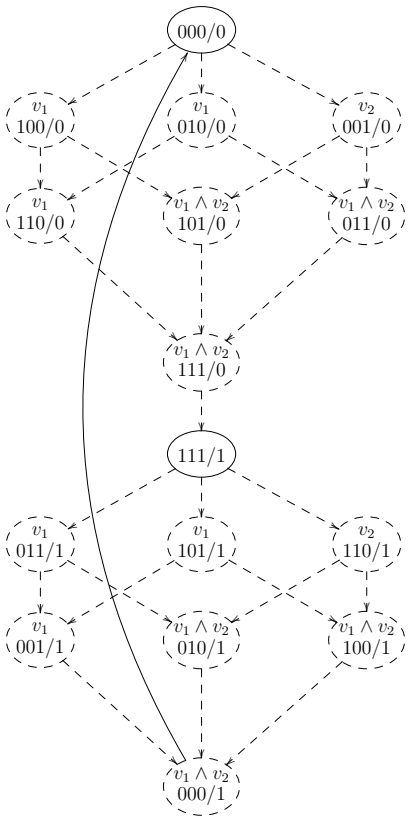


Figura 5.5: Circuito con fallas

que no se presentan violaciones del tipo v_2, es decir que dicha sincronización nunca falla, logrando reducir el tamaño del modelo. En la Figura 5.6 se puede apreciar este modelo simplificado.

En esta nueva versión del sistema algunas propiedades que resultan interesantes de verificar, podrían ser las siguientes:

P3. $\mathbf{O}(z \ \mathcal{W} \ \neg x \wedge \neg y)$, *cuando no ocurren fallas, la salida del circuito no cambia hasta que no cambien todas las entradas.*

P4. $\mathbf{R}(\mathsf{F}(x \equiv z \wedge x \equiv y))$, *Luego de la ocurrencia de una falla, eventualmente un estado correcto será alcanzado.*

Observando el modelo presentado podemos afirmar que **P3** se cumple dado que

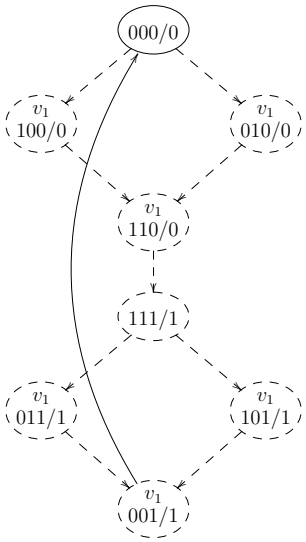


Figura 5.6: Otro modelo más restrigido, donde nunca falla la sincronización de u y z

en los estados correctos del sistema ambas entradas cambian al mismo tiempo, es decir $\neg z$ es verdadero únicamente cuando $\neg x$ y $\neg y$ son verdaderos.

Por otro lado, la fórmula **P4** representa una propiedad que es usualmente utilizada en CTL para expresar propiedades de recuperación (*Recovery*) [18]:

- $\mathsf{AG}(\neg \mathbf{n} \rightarrow \mathsf{AF}(x \equiv z \wedge x \equiv y))$.

También podemos observar que esta propiedad es falsa para el circuito a menos que se impongan restricciones de *fairness* sobre el modelo, ya que en una ejecución "injusta" el circuito podría nunca alcanzar nuevamente un estado correcto dado que otros componentes sean siempre favorecidos por la política de *scheduling*. Sin embargo se podría implementar *fairness* de una manera bastante directa en nuestro algoritmo de *model checking*.

Cabe recalcar que ambas propiedades serán verificadas utilizando dos *model checkers*, cuando pongamos a prueba la performance de nuestra herramienta y podremos comprobar si el análisis realizado es correcto.

Figura 5.7

5.5. Filósofos Comensales

El problema de los *Filósofos Comensales* es uno de los ejemplos más utilizados en el área de concurrencia, en particular resulta muy útil para ilustrar la idea de *deadlock* y *livelock*. En esta sección, nosotros consideramos una variante de este problema en la cual los filósofos pueden presentar "fallas" (como se presenta en [116]), es decir, en cualquier momento un filósofo puede entrar en un estado de error y permanecer en dicho estado para siempre. La principal dificultad en este nuevo escenario viene dada por el hecho de que si un filósofo falla, es decir un proceso se "cae", puede prevenir que otros procesos accedan a los recursos.

Aquí analizamos la implementación dada en [97], en la cual se dividen a los filósofos en dos grupos: pares e impares, en el caso del primer grupo toman el tenedor de su derecha y luego toman el tenedor de su izquierda. Por otro lado los filósofos impares, los toman de manera inversa, primero toman el tenedor de su izquierda y luego recién el de su derecha. Se puede demostrar que con esta política se evita la ocurrencia de *deadlock*.

Algunas propiedades interesantes acerca de este sistema son:

P5 *Ante la ausencia de fallas, cualquier filósofo tiene la posibilidad de comer en algún momento futuro*, esta propiedad puede ser capturada con la siguiente fórmula dCTL:

- $\mathbf{P}(\mathsf{F} ph_i.eat)$.

P6 *Únicamente las fallas de los filósofos vecinos pueden afectar a un filósofo dado*, la cual se puede representar con la siguiente fórmula:

- $\mathbf{R}(\neg ph_{i-1}.crash \wedge \neg ph_{i+1}.crash \rightsquigarrow \mathsf{EF}(ph_{i-1}.eat \vee ph_i.eat \vee ph_{i+1}.eat))$.

En el caso de **P5**, la misma implica una simple propiedad de permiso la cual puede ser traducida a CTL con la siguiente fórmula: $\mathsf{EG}(\mathsf{E}(\mathbf{n}\ \mathcal{U}\ phi_i.eat))$. La cual expresa el hecho de que en algún momento futuro el sistema se reestablecerá.

La segunda propiedad expresa la *localidad de las fallas* de este modelo: *Las fallas son encapsuladas a los vecinos inmediatos*. Es decir que las fallas de los nodos adyacentes del modelo pueden afectar a un nodo dado.

5.6. Análisis de Performance

En esta sección, se tomaron 3 de los casos de estudios, que presentamos previamente para evaluar la performance de nuestro *model checker* con respecto *NuSMV* [2], el cual es otro model checker simbólico clásico del área.

5.6.1. Entorno de Ejecución

Todos los experimentos que realizamos fueron ejecutados en una computadora con las siguientes características:

- Procesador: Intel i5 1.3Ghz
- RAM: 4GB
- Sistema Operativo: Mac OSX 10.9.5
- Para compilar y ejecutar nuestra herramienta utilizamos Oracle Java 1.8.0_05.
- Se utilizó la versión 2.6.0 de NuSMV.

Los tiempos que figuran en las tablas de cada experimento están expresados en segundos, y se utilizó el comando `time` de Unix para realizar las mediciones. La manera que lo calculamos es la siguiente, se realizó la diferencia entre el tiempo actual y el tiempo del sistema como una aproximación del tiempo real (el tiempo de usuario no es tenido en cuenta a la hora de realizar la aproximación del tiempo real, ya que el mismo involucra el tiempo consumido por otros procesos).

5.6.2. Propiedades a Verificar

Repasemos las propiedades que vamos a verificar de cada modelo.

Celda de Memoria

Para el sistema descripto en 5.3, las propiedades que vamos a verificar son las siguientes.

P1. $\mathbf{O}(r \equiv w)$, *ante la ausencia de fallas, el valor del bit leído coincide con el último valor de bit escrito*,

P2. $\mathbf{R}(r\,\mathcal{U}\,\mathbf{n} \rightsquigarrow (r \equiv w)\,\mathcal{U}\,\mathbf{n})$, *luego de la ocurrencia de una falla, si la operación de lectura lee un* 1 *hasta el momento que el sistema regrese a un estado correcto, entonces el valor leído es correcto.*

Estas fórmulas pueden ser traducidas a las siguientes fórmulas CTL:

P1'. $\neg\mathsf{E}(\mathbf{n}\,\mathcal{U}\,(r \not\equiv w) \wedge \mathsf{EG}(\mathbf{n}))$

P2'. $\neg\mathsf{EF}(\neg n \wedge \mathsf{E}(r \wedge \neg\mathbf{n}\,\mathcal{U}\,\mathbf{n} \wedge r \not\equiv w))$

Muller C-element

En el caso del circuito presentado en 5.4, se eligieron las siguientes propiedades para verificar:

P3. $\mathbf{O}(z\ \mathcal{W}\ \neg x \wedge \neg y)$, *cuando no ocurren fallas, la salida del circuito no cambia hasta que no cambien todas las entradas.*

P4. $\mathbf{R}(\mathsf{F}(x \equiv z \wedge x \equiv y))$, *Luego de la ocurrencia de una falla, eventualmente un estado correcto será alcanzado.*

La traducción a CTL de las mismas es la siguiente:

P3'. $\neg\mathsf{E}(\mathbf{n}\,\mathcal{U}\,\mathsf{E}((x \vee y) \wedge \mathbf{n}\,\mathcal{U}\,(x \vee y) \wedge \neg z \wedge \mathsf{EG}(\mathbf{n})))$

P4'. $\mathsf{AG}(\neg\mathbf{n} \Rightarrow \mathsf{AF}(x \equiv z \wedge x \equiv y))$

Filósofos Comensales

Las propiedades que fueron seleccionadas para verificar para el modelo de los Filósofos comensales(especificado en 5.5) son:

P5 *Ante la ausencia de fallas, cualquier filósofo tiene la posibilidad de comer en algún momento futuro*, esta propiedad puede ser capturada con la siguiente fórmula dCTL:

- $\mathbf{P}(\mathsf{F}ph_i.eat)$.

P6 *Únicamente las fallas de los filósofos vecinos pueden afectar a un filśofo dado*, la cual se puede representar con la siguiente fórmula:

- $\mathbf{R}(\neg ph_{i-1}.crash \wedge \neg ph_{i+1}.crash \rightsquigarrow \mathsf{EF}(ph_{i-1}.eat \vee ph_i.eat \vee ph_{i+1}.eat))$.

Estas fórmulas pueden ser traducidas a las siguientes fórmulas CTL:

P5'. $\mathsf{EG}(\mathsf{E}(\mathbf{n}\,\mathcal{U}\,ph_i.eat))$

P6'. $\neg\mathsf{EF}(\neg\mathbf{n}\wedge\neg ph_{i-1}.crash\wedge\neg ph_{i+1}.crash\wedge\mathsf{AG}(\neg ph_{i-1}.eat\wedge\neg ph_i.eat\wedge\neg ph_{i+1}.eat))$

```
Process Memory {
    w: BOOL; // ultimo valor escrito,
    r: BOOL; // valor obtenido de leer la memoria utilizando voting
    c0: BOOL; // primer bit
    c1: BOOL; // segundo bit
    c2: BOOL; // tercer bit

    Initial: w && c0 && c1 && c2 && r;
    Normative: (c0==c1) && (c1==c2) && (c0==c2);

    true -> w=!w, c0=!c0, c1=!c1, c2=!c2, r =(!c0&&!c1) || (!c1&&!c2) || (!c0&&!c2)
        ;
    true -> c0=!c0, r = (!c0&&c1) || (c1&&c2) || (!c0&&c2);
    true -> c1=!c1, r = (c0&&!c1) || (!c1&&c2) || (c0&&c2);
    true -> c2=!c2, r = (c0&&c1) || (c1&&!c2) || (c0&&!c2);
}

Main(){
    m1: Memory;
    run m1();
}
```

Figura 5.8: Código *Faulty* para una memoria con triple redundancia

5.6.3. Modelos

Para poder realizar la comparación de ambas herramientas fue necesario, además de realizar la traducción de cada propiedad a una fórmula CTL equivalente, especificar cada sistema en el lenguaje de especificación definido para NuSMV.

```
MODULE Memory
VAR
   b0: boolean; -- first bit
   b1: boolean; -- second bit
   b2: boolean; -- third bit
   w : boolean; -- bit indicating the last value written
   r: boolean;  -- the read value from the memory

INIT
   b0 & b1 & b2 & w & r

TRANS
   next(w)=FALSE  & next(b0)=FALSE & next(b1)=FALSE & next(b2)=FALSE & next(r)=
       FALSE
   |
   next(w)=TRUE  & next(b0)=TRUE & next(b1)=TRUE & next(b2)=TRUE & next(r)=TRUE
   |
   next(b0)=!b0 & next(b1)=b1 & next(b2)=b2 & next(w)=w & next(r)= (!b0&b1 | b1&b2
       | !b0&b2)
   |
   next(b0)=b0 & next(b1)=!b1 & next(b2)=b2 & next(w)=w & next(r)= (b0&!b1 | !b1&b2
        | b0&b2)
   |
   next(b0)=b0 & next(b1)=b1 & next(b2)=!b2 & next(w)=w & next(r)= (b0&b1 | b1&!b2
       | b0&!b2)


DEFINE
   n := b0=b1 & b1=b2;

MODULE main
VAR
  m1 : process Memory();
```

Figura 5.9: Especificación de una memoria con triple redundancia - *NuSMV* .

A modo de ejemplo en las Figuras 5.8 y 5.9 podemos encontrar la especificación del caso de estudio de la celda de memoria, implementada utilizando 3 bits.

Notar que en la especificación *Faulty*, presentada en la Figura 5.8, las fallas de este modelo de memoria están representadas por el hecho de que cualquier *bit* puede *perder* inesperadamente su carga hasta convertirse en 0, lo cual se encuentra especificado por las 3 últimas transiciones del proceso *Memory*, la cuales se encuentran

habilitadas en cualquier instante para ser ejecutadas, modelando de esa manera la aleatoriedad de la falla.

Respecto a la especificación del modelo para *NuSMV*, se puede apreciar que es bastante similar al lenguaje *Faulty*, por lo cual la traducción del modelo fue bastante directa, con la salvedad que se requirió introducir un nuevo símbolo para marcar los estados normativos, definiendolo a través de una macro, en la Figura 5.9 se puede observar dicha especificación.

El resto de las especificaciones de cada uno de los sistemas para las diferentes entradas y de las propiedades definidas para cada uno de ellos se puede encontrar en el repositorio de *Faulty* [1].

5.6.4. Resultados

Celda de Memoria

En este caso hemos verificado dichas propiedades para memorias con más de 16 bits. Los resultados obtenidos y la comparación con los tiempos utilizados por NuSMV para las propiedades equivalentes, se presentan en las tablas 5.1 y 5.2.

		Faulty	NuSMV
		$\mathbf{O}(r \equiv w)$	$\neg\mathsf{E}(\mathbf{n}\,\mathcal{U}\,(r \not\equiv w) \wedge \mathsf{EG}(\mathbf{n}))$
#bits	#Estados	Tiempo (segundos)	Tiempo (segundos)
3	2^4	0.328	0.046
5	2^6	0.369	0.050
7	2^8	0.479	0.053
9	2^{10}	0.587	0.134
11	2^{12}	0.911	0.460
13	2^{14}	1.905	2.88
15	2^{16}	9.50	17.21
17	2^{18}	55.6	160.1

Cuadro 5.1: Performance para Propiedad **P1** y su traducción a CTL.

P1 asegura el comportamiento correcto de la memoria ante la ausencia de fallas, mientras que la propiedad **P2** establece que esta implementación tolera hasta la ocurrencia de una falla, es fácil ver que mientras más cantidad de bits se repliquen, mayor cantidad de fallas serán toleradas.

		Faulty	NuSMV
		$\mathbf{R}(r\ \mathcal{U}\ \mathbf{n} \rightsquigarrow (r \equiv w)\ \mathcal{U}\ \mathbf{n})$	$\neg\mathsf{E}(\neg\mathbf{n} \wedge \mathsf{E}(r \wedge \neg\mathbf{n}\ \mathcal{U}\ \mathbf{n} \wedge r \not\equiv w))$
#bits	#Estados	Tiempo(segundos)	Tiempo(segundos)
3	2^4	0.302	0.038
5	2^6	0.316	0.041
7	2^8	0.374	0.057
9	2^{10}	0.489	0.105
11	2^{12}	0.736	0.551
13	2^{14}	1.94	3.052
15	2^{16}	9.36	16.41
17	2^{18}	46.84	155.47

Cuadro 5.2: Performance para Propiedad **P2** y su traducción a CTL.

Notar que esta propiedad es falsa para este modelo, dado que existe al menos una traza en la cual se que se puede alcanzar un estado donde los tres *bits* hayan cambiado a *cero* debido a fallas, pero sin embargo el último valor escrito era *uno*. Esto nos lleva a pensar que la manera que están caracterizados los estados *normativos* debe ser refinada, dado que actualmente es imposible distinguir aquellos estados donde si bien los tres bits coinciden, lo hacen como resultado de fallas ocurridas en el sistema. Esto se podría lograr enriqueciendo el predicado que caracteriza los estados *normativos*, por ejemplo, agregando un nuevo bit que discrimine si se alcanzó dicho estado luego de la ocurrencia de una falla o no.

En el caso de las tablas 5.1 y 5.2, si analizamos el tiempo requerido por cada herramienta para verificar dichas propiedades, podemos observar que *NuSMV* obtuvo mejores tiempos que *FaultyCheck* para modelos pequeños, pero a medida que el espacio de estados iba creciendo nuestra herramienta comenzó a mejorar su performance.

Muller C-element

Con el objetivo de medir la performance de nuestra herramienta hemos implementado este circuito para 32 entradas tanto para *FaultyCheck* como para *NuSMV*. Las siguientes fórmulas, y sus correspondientes generalizaciones para n-entradas, fueron verificadas con ambos *model checkers*:

Estas fórmulas, y sus correspondientes traducciones a CTL, fueron verificadas

		Faulty	NuSMV
		$\mathbf{O}(z\ \mathcal{W}\ \neg x \wedge \neg y)$	$\neg\mathsf{E}(\mathbf{n}\ \mathcal{U}\ \mathsf{E}((x \vee y) \wedge \mathbf{n}\ \mathcal{U}\ (x \vee y) \wedge \neg z \wedge \mathsf{EG}(\mathbf{n})))$
#Entradas	#Estados	Tiempo (segundos)	Tiempo (segundos)
2	2^2	0.429	0.015
4	2^6	0.491	0.018
6	2^{11}	0.506	0.031
8	2^{14}	0.557	0.049
10	2^{16}	0.731	0.54
12	2^{20}	0.923	2.61
14	2^{24}	0.971	11.3
16	2^{28}	1.092	92.47

Cuadro 5.3: Performance para la Propiedad **P3** y su traducción a CTL.

		Faulty	NuSMV
		$\mathbf{R}(\mathsf{F}(x \equiv z \wedge x \equiv y))$	$\mathsf{AG}(\neg\mathbf{n} \Rightarrow \mathsf{AF}(x \equiv z \wedge x \equiv y))$
#Entradas	#Estados	Tiempo (segundos)	Tiempo (segundos)
2	2^2	0.530	0.08
4	2^6	0.674	0.09
6	2^{11}	0.646	0.015
8	2^{14}	0.701	0.035
10	2^{16}	1.017	0.42
12	2^{20}	1.34	2.53
14	2^{24}	1.57	11.3
16	2^{28}	1.7	132.7

Cuadro 5.4: Performance para la Propiedad **P4** y su traducción a CTL.

utilizando Faulty y NuSMV. Los resultados obtenidos para $2, 4, 6, 8, 10, 12, 14$ y 16 entradas son presentados en la Tabla 5.3 y la Tabla 5.4. Déjenos destacar que para el caso de 16 entradas el modelo tiene 2^{26} estados alcanzables, y nuestra herramienta fue capaz de verificar esto en pocos segundos. También verificamos la propiedad **P3** para un circuito de 32 entradas, para el cual demoró alrededor de 10 minutos. En dicho caso el modelo tenía más de 2^{50} estados alcanzables; cabe mencionar también

que NuSMV le tomó más de 3 horas verificar la traducción a CTL de la propiedad **P3** para el circuito de 32 entradas.

Filósofos Comensales

Las fórmulas **P5** y **P6**, y sus correspondientes traducciones a CTL, fueron verificadas utilizando Faulty y NuSMV. Los resultados obtenidos hasta 10 filósofos son presentados en la Tabla 5.5 y la Tabla 5.6, respectivamente. Podemos observar que si bien en un comienzo *NuSMV* obtiene mejores tiempos para espacio de estados pequeños, pero a partir de modelo con 2^{24} estados alcanzables, nuestra herramienta fue capaz de verificar dichas propiedades en tiempos menores que *NuSMV*.

		Faulty	NuSMV
		$\mathbf{P}(\mathsf{F}ph_i.eat)$	$\mathsf{EG}(\mathsf{E}(\mathbf{n}\,\mathcal{U}\,ph_i.eat))$
#Filósofos	#Estados	Tiempo (segundos)	Tiempo (segundos)
2	2^5	0.410	0.017
3	2^7	0.585	0.019
4	2^{10}	0.635	0.021
5	2^{13}	0.680	0.036
6	2^{15}	0.711	0.100
7	2^{18}	0.726	0.227
8	2^{19}	0.782	0.647
9	2^{24}	0.801	1.918
10	2^{26}	0.833	7.35

Cuadro 5.5: Performance para Propiedad **P5** y su traducción a CTL.

5.7. Resumen del capítulo

A lo largo de este capítulo, nos abocamos a exhibir, mediante algunos casos de estudio, cómo dCTL resulta adecuada para expresar ciertos patrones que caracterizan propiedades de sistemas tolerantes a fallas, y cómo a través de la utilización de operadores deónticos se pueden capturar algunas situaciones que se presentan en este tipo de sistemas.

En las últimas secciones hemos seleccionado algunas de las propiedades y casos de estudio para comparar la *performance* de nuestro algoritmo de *model checking*

		Faulty	NuSMV
		P6	$\neg\mathsf{EF}(\neg\mathbf{n} \wedge \neg ph_{i-1}.crash \wedge \neg ph_{i+1}.crash \wedge \mathsf{AG}(\neg ph_{i-1}.eat \wedge \neg ph_i.eat \wedge \neg ph_{i+1}.eat))$
#Phils	States	Time (seconds)	Time (seconds)
2	2^5	0.372	0.011
3	2^7	0.405	0.014
4	2^{10}	0.470	0.028
5	2^{13}	0.532	0.053
6	2^{15}	0.540	0.110
7	2^{18}	0631	0.313
8	2^{19}	0.851	0.793
9	2^{24}	0.912	2.050
10	2^{26}	1.803	7.715

Cuadro 5.6: Performance para Propiedad **P6** y su traducción a CTL.

con respecto a NuSMV, el cual es un reconocido model checker simbólico para CTL. Hemos podido apreciar que a medida que crece el número de estados de los modelos nuestra herramienta ha obtenido mejores o iguales tiempos que NuSMV, en lo que se refiere al tiempo que demanda la traducción de la fórmula y verificación de la misma. Recordemos que en el capítulo 3 se puede encontrar el detalle para realizar una traducción de algunas de las fórmulas dCTL a CTL.

Creemos que la eficiencia de nuestra herramienta esta dada principalmente por dos razones, por un lado al utilizar una representación simbólica para los modelos y las fallas mejoramos la *performance* de lo algoritmos de punto fijo utilizados por el *model checker*. Por otro lado la estructura de nuestro lenguaje de modelado nos permitió aplicar la técnica de *early quantification*, la cual reduce el costo algorítmico del proceso de verificación.

El código fuente de nuestra herramienta, asi como la descripción completa de estos casos de estudio, pueden encontrarse en el repositorio de *Faulty* [1].

Capítulo 6

Sistemas Probabilistas

En este capítulo presentamos el trabajo realizado en colaboración con otros investigadores, en donde definimos un nuevo formalismo lógico el cual incorpora nociones de probabilidad para la verificación de propiedades relacionadas con *tolerancia a fallas*. Además de su definición teórica, el aporte realizado, en el marco de esta tesis, está enfocado en el uso de dicho formalismo para el análisis de propiedades de tolerancia a fallas. Dicha lógica probabilística es un fragmento de μPCTL, la cual fue el resultado de un trabajo previo con este mismo equipo de investigadores; en la sección 2.6.3 se encuentra una breve caracterización de la misma.

A lo largo de las diferentes secciones de este capítulo presentamos las principales características de este fragmento lógico, pero sobre todo haremos hincapié en su utilidad para la verificación de sistemas a través de algunos casos de estudio.

6.1. Probabilidad: un concepto natural en tolerancia a fallas

En la práctica la noción de robustez, o de tolerancia fallas, en particular, no es en la mayoría de los casos una noción absoluta (cualitativa), sino que este tipo de noción es cuantificable (es decir, los sistemas suelen tener un *grado* de robustez/tolerancia). En estas situaciones surge naturalmente la idea de utilizar probabilidades para capturar la cuantificación de la noción de robustez deseada. Las probabilidades, usadas de la forma en que se hace en *model checking* probabilista, parecen adecuadas para ésta tarea. En este capítulo se presentan algunos avances preliminares hacia

dichos objetivos .

En particular, presentamos una lógica la cual es un fragmento de μPCTL. Este nuevo formalismo consta de un operador recursivo, el cual es un máximo punto fijo.

Dicho operador permite expresar la noción de recursión dentro de la lógica, y al mismo tiempo limita sintácticamente el anidamiento del operador de punto fijo, lo cual limita la complejidad de la lógica.

Sin embargo, al introducirlo a través de operadores recursivos aprovechamos la naturalidad del concepto de recursión tan presente en ciencias de la computación y al no permitir el anidamiento de diferentes patrones de recursión (mínimos y máximos puntos fijos) limitamos la complejidad de la lógica. Otros puntos a favor que tiene este nuevo formalismo, al cual llamamos RPCTL, es que el mismo tiene mayor expresividad que PCTL y permite caracterizar varios patrones de repetición interesantes para sistemas probabilísticos.

Adaptamos los resultados demostrados acerca de la expresividad de μPCTL al contexto de RPCTL y de hecho mostramos que ésta última es aún más expresiva que PCTL. Por otro lado, demostramos que la complejidad del algoritmo de *model checking* para RPCTL coincide con la complejidad de *model checking* para PCTL. De hecho el algoritmo para este nuevo formalismo invoca repetitivamente al *model checker* de PCTL. Otra característica de este algoritmo para remarcar, es que el mismo es polinomial con respecto al tamaño de la cadena de Markov subyacente.

En lo que se refiere al campo de aplicación de ésta nueva lógica, creemos que resulta adecuada para la verificación de propiedades relacionadas con *tolerancia a fallas*. El grado de tolerancia exibido por un sistema tolerante a fallas dado puede ser caracterizado utilizando conjunto de estados seguros. Por ejemplo, se dice que un sistema es *fail-safe* si ante la ocurrencia de una falla todas sus ejecuciones permanecen en un conjunto de estados seguros [15]; en cambio, se dice que es *non-masking* cuando revisita con infinita frecuencia el conjunto de estados seguros (o deseables) [15].

Algo muy importante a tener en cuenta, es que en el caso de los sistemas probabilísticos (y de lógicas probabilísticas en general), la caracterización de este tipo de propiedades no puede hacerse de una manera directa. Esto es debido a que la probabilidad de que un sistema permanezca en un conjunto de estados *seguros* es 0 (en la mayoría de los casos), en el caso de que la probabilidad de la ocurrencia de una falla sea un valor positivo, ya que en ese caso el sistema eventualmente saldrá de ese conjunto de estados "buenos" con probabilidad 1. Mas adelante en este capítulo analizaremos con mayor detenimiento esta situación a través de algunos ejemplos. Es

por ello que en lugar de utilizar los cuantificadores clásicos para máximo y mínimo punto fijos utilizaremos una sintaxis enriquecida con los operadores recursivos, cuyo objetivo es facilitar la manera de especificar, y verificar, este tipo de propiedades.

6.2. RPCTL una lógica temporal probabilista con operadores recursivos

En esta sección presentamos una lógica probabilística la cual es un fragmento de μPCTL. En este capítulo presentamos las principales características de la misma, pero sobre todo haremos hincapié en su utilidad para la verificación de sistemas a través de algunos casos de estudio. A *groso modo* se podría decir que se incluyen operadores recursivos dentro de PCTL, lo cual permite expresar propiedades acerca de la estabilidad de las ejecuciones del sistema; dichas propiedades son muy comunes en aquellos escenarios donde uno está interesado en verificar cuando un sistema permanece, o revisita recurrentemente, un conjunto de estados *seguros*.

Las fórmulas de ésta nueva lógica pueden contener llamadas recursivas, esto se logra a través de dos nuevos operadores rec y call los cuales son una manera más amigable de utilizar el máximo punto fijo, el cual denota la semántica de dichos operadores. Técnicamente hablando, estos operadores introducen la noción de máximo punto fijo en la lógica, la cual, de hecho, es un fragmento de μPCTL.

A continuación presentamos la sintaxis y semántica de RPCTL.

Sea AP un conjunto de proposiciones atómicas $\{p_0, p_1, \dots\}$; los conjuntos Φ y Ψ de fórmulas de estado y fórmulas de camino, respectivamente, son definidas recursivamente de la siguiente manera:

$$\begin{array}{lcl} J & ::= & \{>, \geq\} \times [0,1] \\ \Phi & ::= & \top \mid \bot \mid p_i \mid \neg p_i \mid \mathsf{call}_i \mid \Phi_1 \vee \Phi_2 \mid \Phi_1 \wedge \Phi_2 \mid \mathcal{P}_J(\Psi) \mid \mathsf{rec}_i.\Phi \\ \Psi & ::= & \mathsf{X}\Phi \mid \Phi \,\mathcal{U}\, \Phi \mid \Phi \,\mathcal{W}\, \Phi \end{array}$$

Nos interesan fórmulas en las cuales todas sus variables se encuentran bajo el alcanze de algún operador, es decir fórmulas que no tienen variables *libres*. Los índices que aparecen en las sentencias rec y call , cumplen dos propósitos, por un lado ellos sirven para enumerar las variables para la recursión($\mathsf{call}_0, \mathsf{call}_1, \mathsf{call}_2, \dots$); y en segundo lugar, ayudan a visualizar el alcance de cada cuantificador, es decir: call_i esta bajo el alcance de rec_i, para cada i.

En este capítulo estaremos interesados en fórmulas cuyas variables se encuentran cuantficadas, es decir, no consideraremos fórmulas con variables libres.

Describiremos la semántica de RPCTL de la siguiente manera. Intuitivamente, una fórmula RPCTL caracteriza el conjunto de estados de una cadena de *Markov* en los cuales vale dicha fórmula.

Consideremos la siguiente cadena de *Markov* $M = \langle S, P, L, s_0 \rangle$. La semántica de la subfórmula depende de una valuación que asocia un conjunto de estados a cada sentencia call que aparece en la misma. Formalmente, una valuación es una función a la cual denominamos τ, que se encuentra definida de la siguiente manera $\tau : \{\mathsf{call}_0, \mathsf{call}_1, \dots\} \to 2^S$. Denotamos $\tau[S'/\mathsf{call}_i]$ a la valuación tal que $\tau(\mathsf{call}_i) = S'$ y para cada $i \neq j$ se cumple que $\tau[S'/\mathsf{call}_i](\mathsf{call}_j) = \tau(\mathsf{call}_j)$.

La semántica de la fórmula φ, denotada como $[\varphi]^M_\tau$ la definimos de la siguiente manera:

$$\begin{aligned}
[p_i]^M_\tau &= L(p_i) \\
[\neg p_i]^M_\tau &= S \setminus L(p_i) \\
[\mathsf{call}_i]^M_\tau &= \tau(\mathsf{call}_i) \\
[\varphi_1 \wedge \varphi_2]^M_\tau &= [\varphi_1]^M_\tau \cap [\varphi_2]^M_\tau \\
[\varphi_1 \vee \varphi_2]^M_\tau &= [\varphi_1]^M_\tau \cup [\varphi_2]^M_\tau \\
[\mathcal{P}_J(\Psi)]^M_\tau &= \{s \in S \mid \mathsf{measure}_M(s, \Psi) J\} \\
[\mathsf{rec}_i.\Phi]^M_\tau &= \mathsf{gfp}\{S' \subseteq S \mid S' = [\Phi]^M_{\tau[S'/\mathsf{call}_i]}\}
\end{aligned}$$

Ahora presentaremos a través de algunos ejemplos la intuición que hay detrás de estos operadores definidos. Consideremos la siguiente fórmula:

$$\mathsf{rec}.p \wedge \mathcal{P}_{>0}(\mathsf{X}\mathsf{call}), \tag{6.1}$$

Notar que, por cuestiones de simplicidad, no escribimos los índices en cada instancia de los operadores rec y call.

Esta fórmula se cumple en un estado s cuando: *p se cumple en s, y la probabilidad de que dicha fórmula (6.1) se cumpla en los estados siguientes es mayor a* 0.

En otras palabras, p vale en s y s tiene al menos un sucesor que también satisface p, el cual a su vez tiene al menos un sucesor donde también se cumple p, y así sucesivamente. Esta propiedad se puede escribir equivalentemente en CTL con la siguiente fórmula: $\mathsf{EG}p$.

En el caso del operador rec, si tenemos la siguiente fórmula:

$$\mathsf{rec}.\mathcal{P}_{>0.5}(\mathsf{call}\,\mathcal{U}\,p), \tag{6.2}$$

se cumple en el estado s si *recursivamente, hay una probabilidad de más de* 0,5 *de que en los estados sucesores se cumpla esta misma propiedad hasta llegar a un estado en donde p sea verdadera.* .

Es decir, cada estado encontrado o visitado en el camino hasta llegar al estado donde p es cierta, tiene mas de $1/2$ de probabilidad que sus sucesores cumplan con esta propiedad. Como demostraremos a continuación esta propiedad no se puede expresar en CTL ni en PCTL.

Intuitivamente, podemos ver cada variable call_i como una nueva proposición. Cada estado etiquetado por una de las nuevas proposiciones necesita satisfacer la fórmula PCTL obtenida de la llamada recursiva correspondiente, donde cada instancia del operador call son reemplazadas por su correspondiente fórmula. En los ejemplos sucesivos se afianzarán estas primeras intuiciones.

6.2.1. Poder Expresivo

Mostraremos que RPCTL es más expresiva que PCTL e incomparable con CTL.

Teorema 6.1. *RPCTL es más expresiva que PCTL.*

Demostración. Primeros notemos que a nivel sintáctico PCTL se encuentra incluida en RPCTL. Es decir existen propiedades expresadas en RPCTL que no se pueden expresar en CTL. Por ejemplo: La propiedad $\mathsf{rec}.p \wedge \mathcal{P}_{>0.5}(\mathsf{X} \wedge \mathsf{call})$ no se puede expresar en PCTL ni en CTL, en [30] se puede encontrar una demostración más detallada de este resultado. □

Notar que si incluyéramos los cuantificadores existencial y universal en RPCTL, esto no tendría por que incrementar la complejidad de los algoritmos del *model checking*. Esto nos permitiría incluir CTL en RPCTL, si así lo deseamos.

A continuación presentamos algunos otros ejemplos y en ellos aparecen propiedades muy similares a la propiedad presentada en la prueba del Teorema 6.1. Es decir, este tipo de propiedades fuerza a la repetición de cierto patrón de probabilidad, incluso si dicho patrón ocurre en un conjunto para el cual su medida de probabilidad es *cero*. Por lo que sería posible demostrar que estas propiedades no se pueden expresar en CTL ni en PCTL. Vale la pena recalcar que esta capacidad de capturar un patrón de "repetición" que posee RPCTL, permite razonar sobre trazas con medida de probalidad cero y constituye una las características que hacen a esta lógica interesante en la práctica.

6.3. Model Checking

Es esta pequeña sección analizaremos el algoritmo de *model checking* de RPCTL, para cadenas de *Markov* con un número finito de estados, el cual es *polinomial* con

Algoritmo 3 Algoritmo de Model Checking para RPCTL

```
let ∀ i . W_i = ∅;
let ∀ i . S_i = S;
do {
   let ∀ i . W_i = S_i;
   let ∀ i . S_i =
      { s | M(S_1, ..., S_n), s ⊨ rec_i . φ_i( call_j ← c_j | j ∈ [1..n]) };
   }
} while (∃ i . S_i ≠ W_i);
if (M(S_1 , ..., S_n), s ⊨ φ((rec_j . φ_j) ← c_j|j ∈ [1..n])) print ‘‘Yes’’;
else print ‘‘No’’;
```

respecto al tamaño de la cadena de *Markov* y de la fórmula a verificar.

Cabe destacar que este algoritmo se basa en el algoritmo que definimos para μPCTL [31], pero esta es una versión restringida que usa únicamente máximos puntos fijos. A continuación presentaremos dicho algoritmo:

Sea $M = \langle S, P, L, s_0 \rangle$ una cadena de *Markov* y φ una fórmula RPCTL.

Supongamos que $\{\mathsf{call}_1, \ldots, \mathsf{call}_n\}$ es el conjunto de sentencias *call* que aparecen en φ, y sea $\{S_1, \ldots S_n\}$ el conjunto de estados de M. Es decir, para cada $1 \leq i \leq n$ se cumple que $S_i \subseteq S$.

Denominamos como $M(S_1, \ldots, S_n)$ a la estructura definida sobre $AP \cup \{c_1, \ldots, c_n\}$ obtenida a partir de M definiendo $L(c_i) = S_i$.

Para la fórmula $\mathsf{rec}_i.\varphi_i$, sea $\mathsf{rec}_i.\varphi_i(\mathsf{call}_j \leftarrow c_j | j \in [1..n])$ la fórmula obtenida de φ_i, donde cada aparición de call_j es reemplazada por c_j. Finalmente, sea $\varphi(\mathsf{rec}_j.\varphi_j \leftarrow c_j | j \in [1..n])$ la fórmula obtenida de φ, reemplazando cada ocurrencia de $\mathsf{rec}_j.\varphi_j$ por c_j.

Una vez realizados estos reemplazos, el Algoritmo 3 calcula si un estado s de M satisface φ. El mismo invoca al algoritmo de *model checking* de PCTL como una función auxiliar.

Teorema 6.2. *Dada una fórmula RPCTL ϕ, el algoritmo 3 retorna "Yes" ssi $s \in [\phi]^M_\tau$, donde τ es una valuación arbitraria.*

Demostración. La prueba se basa en la aproximación de máximos puntos fijos. El algoritmo computa el máximo punto fijo, partiendo desde el conjunto de todos los estados y progresivamente se van eliminando del conjunto todos aquellos estados

que no satisfacen la propiedad. Una vez que el algoritmo termina de computar, el conjunto de estados obtenido constituyen el punto fijo real. □

Teorema 6.3. *El algoritmo 3 toma tiempo polinomial con respecto al tamaño de M y de la fórmula φ.*

Demostración. Nos basamos en el hecho de que el algoritmo de *model checking* de PCTL es polinomial con respecto al tamaño de la fórmula y del modelo. Notar que las proposiciones c_j aparecen en forma positiva en $\varphi(\mathsf{rec}_j.\varphi_j \leftarrow c_j)$ y también en la fórmula $\mathsf{rec}_i.\varphi_i(\mathsf{call} \leftarrow c_j)$. Sea ψ una de estas fórmulas. Por monotonicidad obtenemos que si $S_i \subseteq S_i'$ entonces $\{s \mid M(S_1, \ldots, S_i, \ldots, S_n), s \models \psi\} \subseteq \{s \mid M(S_1, \ldots, S_i', \ldots, S_n), s \models \psi\}$. Por lo tanto, los conjuntos S_i son monótonamente decrecientes. De este hecho se desprende que si continuamos ejecutando el ciclo, luego de a lo sumo $n \cdot |S|$ iteraciones, el ciclo termina.

Por lo cual podemos concluir que el algoritmo invoca un número polinomial de veces al procedimiento de *model checking* de PCTL y el mismo es también polinomial. □

6.3.1. Ejemplos

Exclusión mutua de dos procesos

Consideremos un ejemplo clásico del área de concurrencia y tolerancia a fallas: el problema de exclusión mutua para dos procesos (llamémoslos P_1 and P_2). Introducimos fallas en este modelo permitiendo que los procesos vayan a un estado de error, en este caso los procesos podrían estar *down* por un período de tiempo indeterminado. La idea básica de este ejemplo fue introducida en [18], la diferencia es que aquí nosotros le agregamos probabilidades para poder analizar algunas propiedades de manera cuantitativa.

En cada estado las probabilidades de moverse a un estado sucesor están distribuidas de manera *uniforme*. El modelo que captura este escenario se encuentra ilustrado en la Figura 6.1. La región M_1 (delimitada por líneas punteadas) contiene aquellos estados que pueden ser alcanzados durante el comportamiento *normal* del sistema, o cuando se detiene P_1 pero con una probabilidad positiva de recuperarse. Para poder simplificar el análisis, consideraremos unicamente las fallas en el proceso P_1, y recién una vez que P_1 falla puede fallar P_2 también. Las transiciones para el caso en que primero falle el proceso P_2 y luego P_1 son similares a las ilustradas en la Figura 6.1. En este modelo la proposición N_i se vuelve verdadera cuando el

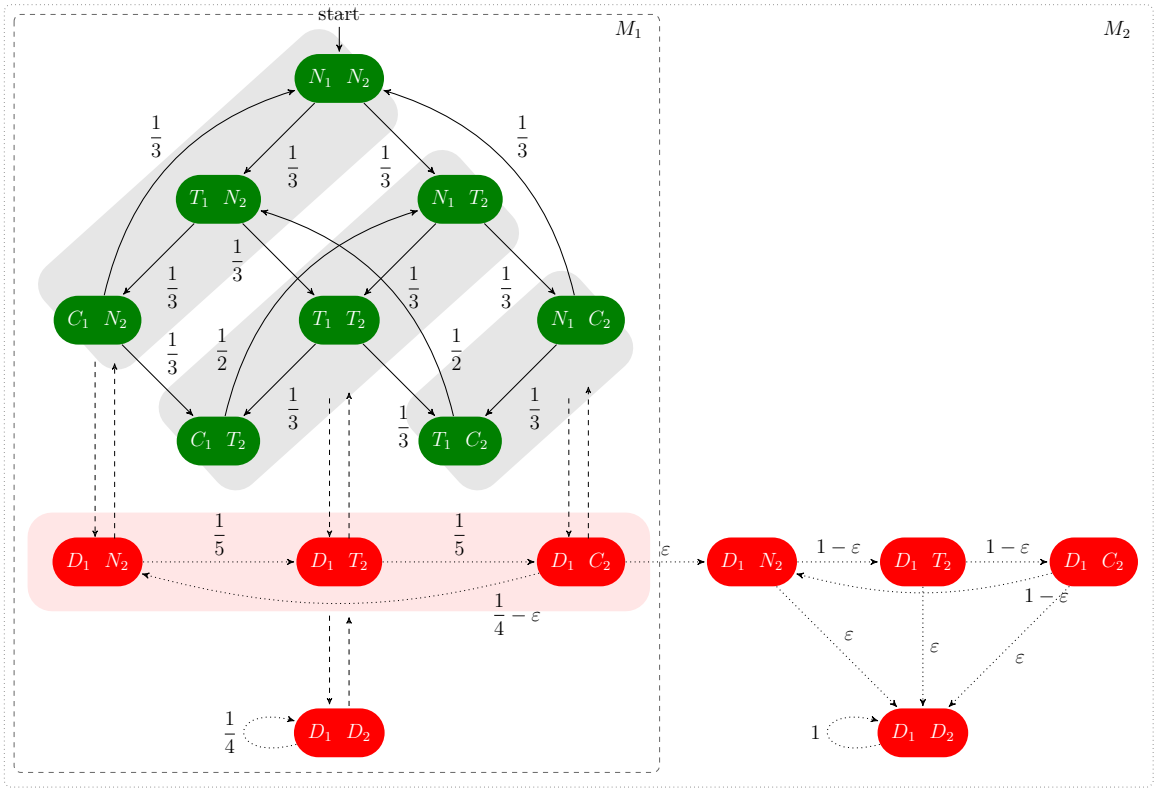


Figura 6.1: Exclusión mutua de dos procesos.

proceso P_i se encuentra en su región *no-critica*, en el caso de las proposiciones T_i y C_i representan la situación en la cual el proceso P_i ingresa a su región *trying* o a su región *crítica*, respectivamente. Por último, la proposición D_i indica que el proceso P_i se encuentra caído o detenido (*down*), lo cual representa la ocurrencia de una falla. Notar que, a los efectos de que la figure resulte mas clara de interpretar, hemos *asociado* estados en regiones, y agregamos transiciones de estos conjuntos de estados a regiones de falla, en lugar de graficar cada transición de manera individual.

Veamos algunas propiedades de este modelo.

Propiedad 1. *Cuando no hay fallas, el proceso* P_1 *permanece en la región segura (*N_1, T_1 *o* C_1 *) con probabilidad mayor o igual a un medio.*

Esta propiedad puede epresarse en νPCTLutilizando la siguiente fórmula:

$$\mathsf{rec.}\left[\neg D_1 \ \wedge \ \mathcal{P}_{\geq \frac{1}{2}}(\mathsf{X}\ \mathsf{call})\right]$$

Red *Token Ring* probabilista

En este segundo ejemplo retomaremos el ejemplo de *token ring* visto en el capítulo anterior, el cual consistía de tres nodos conectados, y donde toda actividad de la red esta regulada a través del protocolo *token ring*. Los tres nodos están conectados entre si en una topología de anillo; y donde un *token* es pasado a través de los nodos de la red de manera tal de garantizar el acceso a un recurso particular al nodo que posea dicho *token*, por ejemplo: permiso para enviar un mensaje a través de la red. La diferencia en este caso es que agregaremos probabilidades para caracterizar el caso en donde el token se pierda.

Recordemos que alguna de las propiedades que se podía requerir a este sistema era la existencia en todo momento de un único token en la red y, que sea quien sea el nodo que posee el *token*, el mismo eventualmente sea pasado al próximo nodo de la red. Además recordemos que una falla que se podía concebir en este sentido, era una en la cual debido a que el medio es inseguro, el *token* podía perderse mientras era enviado entre un nodo y otro. En este caso hemos asignado una probabilidad a la ocurrencia de este evento. Una abstracción probabilística de esta situación, incluída la detección de la falla, es representada en la Figura 6.2. En este modelo, la proposición n_i se vuelve verdadera cuando el *token* es pasado al nodo i. Mientras que ${n_i}'$ representa la situación en la cual el *token* permanece en el nodo i, antes de pasar al próximo nodo.

Es simple de ver que la probabilidad de que el *token* alcance el nodo 1 cuando es enviado por el nodo 0 es $\frac{1}{2}$. Notar que para los otros casos probabilidades similares fueron obtenidad.

Cálculos simples muestran lo siguiente:

$$\mathcal{P}(n_1 \ldots n_2) \ = \ \mathcal{P}(n_2 \ldots n_0) \ = \ \tfrac{1}{2}.$$

Es interesante investigar las propiedades que se cumplen en las regiones sin fallas del sistema.

Propiedad 2. *Cuando no se observan fallas, el token puede permanecer en un estado dado o moverse al estado siguiente, la probabilidad de que ocurra esto es de al menos un medio, y este patrón puede ser repetido una infinidad de veces.*

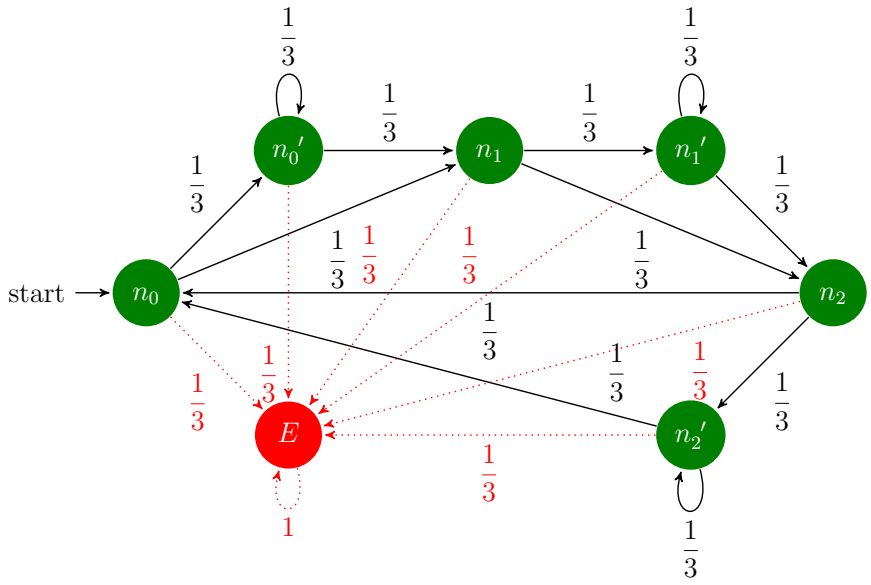


Figura 6.2: Una red *token ring*, donde los *token* pueden perderse.

Notar que en esta propiedad tenemos implícito una noción de *estabilidad*, la cual caracteriza el comportamiento *normal*, o *esperado* del sistema. Un candidato natural para expresar esta propiedad es la siguiente fórmula:

$$\mathsf{rec.}\left[\begin{array}{lll} (n_0 \rightarrow \mathcal{P}_{\geq \frac{1}{2}}(\mathbf{F}(n_1 \ \wedge \mathsf{call}))) & \wedge \\ (n_1 \rightarrow \mathcal{P}_{\geq \frac{1}{2}}(\mathbf{F}(n_2 \ \wedge \mathsf{call}))) & \wedge \\ (n_2 \rightarrow \mathcal{P}_{\geq \frac{1}{2}}(\mathbf{F}(n_0 \ \wedge \mathsf{call}))) & \end{array}\right] \tag{6.3}$$

A groso modo, esta fórmula expresa que, si el *token* se encuentra en el nodo i, entonces la probabilidad de que el *token* alcance el nodo $i+1$, y que este patrón se repita, es de un medio.

Si consideramos el conjunto de estados que satisfacen la primer ocurrencia de call y los etiquetamos como n_1; el conjunto de estados que satisfacen la segunda aparición de call los que están etiquetados como n_2, y sean los estados etiquetados con n_0 el conjunto de estados que satisfacen la última ocurrencia de call, entonces la propiedad PCTL obtenida reemplazando las variables *ligadas* por proposiciones que representan estos conjuntos de estados: n_0, n_1, y n_2.

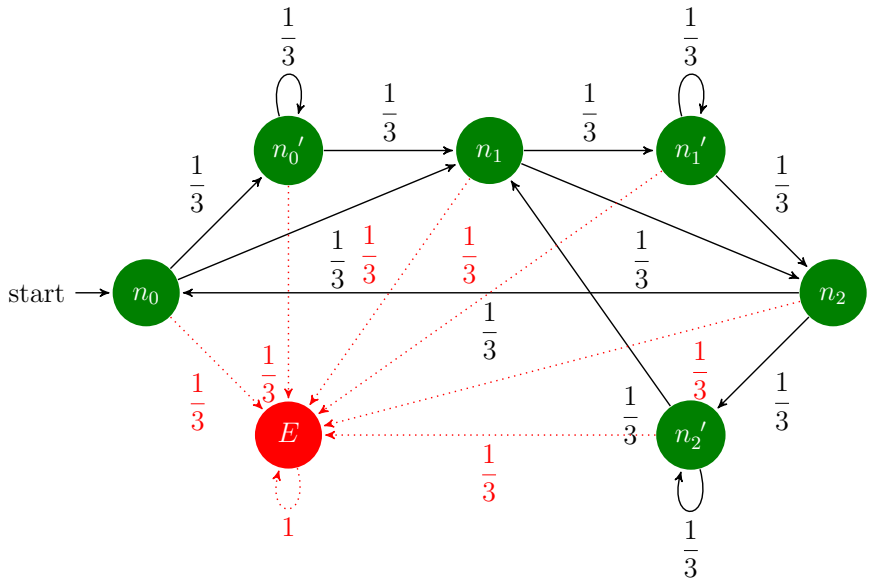


Figura 6.3: Otro modelo de red *token-ring*

Una variante

Consideremos una variante del escenario presentado anteriormente. Ahora, cuando el token se encuentra asignado al nodo 2, podría ocurrir que el mismo permanezca en dicho estado, o se mueva al nodo 1 o al nodo 0, es decir, ahora existe la posibilidad de que el token vuelva a un nodo anterior o pase al nodo siguiente; por ejemplo, este podría ser el caso en un escenario donde el canal que conecta los nodos 2 y nodo 1 sea *corrupto*, y que el token sea retornado a su origen. Esta nueva situación se encuentra representada en la Figura 6.3. La probabilidad de que el token vaya del nodo 2 al nodo 1 es: $\mathcal{P}(n_2 \dots n_0) = 0{,}3636$.

La propiedad anterior no se cumple en este nuevo modelo, notar que el estado n_2 en este modelo no satisface la subórmula $(n_2 \to \mathcal{P}_{\geq \frac{1}{2}}(\mathbf{F}(n_0 \ \wedge \mathsf{call}))$.

Es decir, que RPCTL nos permite distinguir entre estos dos patrones de repetición. En PCTL un podría intentar capturar esta propiedad utilizando la siguiente fórmula:

$$\varphi = \begin{bmatrix} (n_0 \to \mathcal{P}_{\geq \frac{1}{2}}(\mathbf{F}(n_1))) & \wedge \\ (n_1 \to \mathcal{P}_{\geq \frac{1}{2}}(\mathbf{F}(n_2))) & \wedge \\ (n_2 \to \mathcal{P}_{\geq \frac{1}{2}}(\mathbf{F}(n_0))) & \end{bmatrix} \tag{6.4}$$

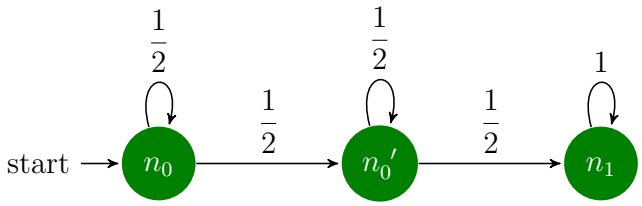


Figura 6.4: Otro modelo que satisface φ.

De hecho φ distinque entre los dos modelos de token ring presentados anteriormente, notar que es verdadera en el modelo de la Figura 6.2, y falsa en el modelo presentado en la Figura 6.3. Sin embargo, notar que esta fórmula PCTL no captura la noción de repetición. Ya que por ejemplo la fórmula 6.4 también es verdadera en estructuras donde el patrón de repetición no está presente, por ejemplo como en la Figura 6.4, donde tenemos que $n_0 \vDash \varphi$.

Otra posible forma de capturar propiedades recursivas como esta utilizando PCTL es através del uso de algun operador global, del estilo de: $\mathcal{P}_{>0}(\mathsf{G}\varphi)$ (donde φ es la propiedad presentada arriba). Pero el conjunto de *paths* que satisfacen esta clase de propiedades tienen medida de probabilidad 0; por lo que es equivalente a $\mathcal{P}_{=0}(\mathsf{G}\varphi)$, lo cual no nos permite realizar ninguna clase de análisis cuantitativo sobre estas trazas.

6.4. Resumen del capítulo

A lo largo de los años han surgido varias variantes de μ-cálculo probabilista. Algunos trabajos destacables son el de Huth y Kwiatkowska [64] y el de McIver y Morgan [103], ambas líneas de trabajo presentan un μ-cálculo cuantitativo que reemplaza la tradicional interpretación booleana de μ-cálculo por una interpretación cuantitativa.

Es decir, la semántica de una fórmula en lugar de estar dada a través de un conjuntos de estados, es una función que asocia un valor a cada estado. Estas lógicas, sin embargo, fallan a la hora de capturar la expresividad de PCTL. Otro punto para hacer notar, es que no poseen un mecanismo que permita volver al obtener fórmulas bajo el dominio booleano.

En el caso del trabajo desarrollado por Mio [107], dicho autor extiende estas lógicas con diferentes interpretaciones de una conjunción cuantitativa. Con el obje-

tivo de poder razonar acerca de diferente tipos de conjunciones, él introduce *parity games* con productos independientes y *tree games*.

En el caso del μ-cálculo cuantitativo presentado por Mio, si captura PCTL y además tiene un fragmento para el cual ciertas subfórmulas se interpretan bajo un dominio *Booleano* y otras en un dominio *cuantitativo* [108]. Desafortunadamente, la complejidad de *model checking* para dicha lógica es muy alta.

En nuestro trabajo [31] introdujimos un subconjunto de la lógica de μ-cálculo probabilista presentada por Mio, denominado μ^p-cálculo. Este fragmento tiene la ventaja que su algoritmo de *model checking* se encuentra en NP $\cap$ co-NP al igual que μ-cálculo. Además presentamos μPCTL, que incorpora puntos fijos en PCTL. Sin embargo esta nueva lógica tiene como desventaja, la cual también lo aprendimos de μ-cálculo, es que al permitir alternar operadores de punto fijo es muy dificil razonar para los usuarios acerca de la semántica de las fórmulas que se escriben. Un fragmento de μPCTL es también considerado en [91].

En este capítulo nosotros presentamos RPCTL, una extensión de PCTL con operadores recursivos. La cual permite especificar propiedades describiendo posible "repeticiones" presentes en la cadena de *Markov* y la probabilidad de la ocurrencia de ciertos eventos dentro de estas repeticiones.

Algo para destacar de este fragmento es que a pesar de agregar poder expresivo, el costo algorítmico de *model checking* es en proporción bajo, ya que termina siendo reiteradas llamadas al *model checker* de PCTL.

Uno de las principales ventajas de RPCTL es que permite capturar propiedades de regiones internas de los modelos.

Esta característica resulta útil en el caso de sistemas donde sea necesario razonar acerca de la repetición de cierto patrón con cierta probabilidad dada. Este es el caso, por ejemplo, de sistemas tolerantes a fallas donde es natural razonar acerca del patrón presente ante la ocurrencia de fallas y capturar la probabilidad de evitar o recuperarse de las mismas. A lo largo de este capítulo presentamos algunos ejemplos que muestran la aplicación de esta lógica en la práctica. Algo que quedó pendiente para trabajos futuros es la implementación del algoritmo de *model checking* y para continuar las investigaciones con casos de estudios mas complejos. Otra línea de investigación para explorar es considerar agregar dentro de la cuantificación de probabilidades expresiones regulares, ya que consideramos que simplificaría la legibilidad de las propiedades.

Capítulo 7

Conclusiones Finales

A lo largo de esta tesis nos centramos principalmente en la definición, y estudio, de formalismos lógicos que resulten adecuados para especificar sistemas tolerantes a fallas. En especial nos enfocamos en intentar capturar, de una manera que resulte lo más intuitiva posible, propiedades características de este tipo de sistemas. Sin embargo, no podemos dejar de mencionar que otro de los objetivos centrales de este trabajo de investigación fué la definición y puesta en práctica de un proceso de verificación automática para los formalismos definidos. Es por ello que, no sólo nos abocamos a definir una lógica que nos ayudara a formalizar dichas propiedades y un lenguaje que nos facilitará la descripción de los modelos sobre los que queremos verificar dichas propiedades, sino que también definimos e implementamos un algoritmo de *model checking* para completar de definir un *framework* de verificación para los mismos.

En particular, nuestro trabajo consistió en el uso de aquellos formalismos que si bien, tradicionalmente fueron utilizados por juristas y filósofos para representar y analizar la estructura lógica de *normas* o *leyes*; conocidos con el nombre de *lógicas deónticas*, estos nos posibilitan, a diferencia de otros enfoques, distinguir entre el comportamiento *normal* y *anormal* de un sistema. Una de la hipótesis de este trabajo es que estos formalismos permiten capturar algunas propiedades de tolerancia a falla de una manera natural e intuitiva, gracias a la analogía que puede establecerse entre la ocurrencia de fallas en los sistemas, que podrían considerarse como *violaciones* de los requerimientos y donde se espera que exista algún mecanismo que permita reestablecer o corregir dicha situación, y lo que ocurre a nivel jurídico cuando se incumple con una ley o norma, donde se requiere la aplicación de medidas correctivas.

En la parte final de esta tesis se presenta uno de los trabajos que hemos hecho en

colaboración con otro grupo de investigadores en donde el objetivo fue la incorporación de nociones de probabilidad para la verificación de propiedades relacionadas con *tolerancia a fallas*. Principalmente hacemos hincapié en el análisis de su utilidad para la verificación de sistemas a través de algunos casos de estudio.

En este capítulo, realizamos un análisis general de nuestro trabajo sobre este problema. En la sección 7.1 presentamos las investigaciones realizadas en el área de especificación y verificación de sistemas tolerantes a fallas, en particular analizamos algunos de los trabajos encontrados en la literatura del uso de lógicas deónticas en procesos formales de verificación de este tipo de sistemas. En 7.1.1, nos enfocaremos en algunos de los enfoques existentes en el área de tolerancia a Fallas los cuales incorporan nociones de probabilidad. Finalmente en las secciones 7.2 y 7.3 nos referiremos a los aportes realizados con nuestras investigaciones y presentaremos algunas de las líneas de trabajo que quisieramos desarrollar en un futuro.

7.1. Trabajos Relacionados

Uno de los puntos centrales de nuestro trabajo se presenta en el capítulo 3, donde presentamos el formalismo lógico denominado dCTL, el cual tiene varios puntos en común con algunos enfoques formales del área. Uno de los primeros trabajos que pudimos encontrar acerca de la utilización de operadores deónticos para realizar razonamientos acerca de tolerancia a fallas es el de *Maibaum* en 1987 [99], donde propoponen una lógica modal que incorpora nociones deónticas para especificar y verificar propiedades sobre sistemas. Como ya hemos mencionado al comienzo de esta tesis, estas ideas continuaron siendo desarrolladas en los trabajos de tesis doctorales de *Khosla* [77] y *Castro* [28].

Otro enfoque que involucra conceptos deónticos es la lógica epistémica denominada *ATLK* [54, 112]. Esta lógica, al igual que en las estructuras de *Kripke* coloreadas que utiliza dCTL, distingue a través de colores los estados *correctos* de los que no lo son. En *ATLK* los estados correctos de los agentes son representados a través del color *verde*, en cambio los estados de color *rojo* representan los estados con fallas. Un model checker que utiliza esta lógica para verificar propiedades sobre sistemas interpretados [54] es MCMAS [96, 95]. La principal aplicación de esta herramienta es la verificación de protocolos donde están involucrados un número dado de agentes (sistemas multi-agentes). Algunas diferencias que podemos notar entre dCTL y *ATLK* es que a pesar de que ambas cuentan con un operador de obligación, las fórmulas condicionales no son consideradas en *ATLK*. Además de que tampoco consideran

ningún operador lógico para poder capturar nociones de *permiso*, *recuperación* ó *reparación*. De hecho, no está claro si se puede lograr expresar propiedades que involucren permisos condicionales en *ATLK*.

También es interesante destacar la lógica introducida en [102], donde se presenta una lógica temporal con operadores deónticos, denominada ROCTL* la cual es utilizada para razonar acerca de sistemas *robustos*. ROCTL* tiene mucho más poder expresivo que dCTL (de hecho es más expresivo que CTL*) y la complejidad de su algoritmo de *model checking* es *PSPACE-hard*. Hasta el momento no conocemos que exista alguna herramienta que implemente los algoritmos de *model checking* para ROCTL*.

Como mencionamos en la primera sección además de definir una lógica que resulte apropiada para capturar y razonar sobre propiedades deseables en los sistemas tolerantes a fallas, este trabajo también implicó la definición de un lenguaje de guardas que nos permitiera la especificación de los sistemas sobre los cuales se iba a realizar la verificación de las propiedades; en el capítulo 4 se puede encontrar una completa descripción del mismo. Este lenguaje desarrollado sigue un estilo similar al introducido por *Arora* y *Gouda* en [15], como así también tiene similitudes con el trabajo de *Gärtner* en [59], donde los programas son descriptos utilizando un lenguaje de guardas al estilo de *Dijkstra*. En estos trabajos, los autores caracterizan a través de conjuntos de estados la tolerancia a falla, es decir que tienen conjuntos de estados los cuales representan las propiedades que se deben garantizar a lo largo de la ejecución del sistema (*invariantes* del sistema) y por otro lado tienen conjuntos de estados para representar aquellos estados a los que se llega luego de la ocurrencia de una falla. Otro punto para analizar de estos trabajos es que a la hora de formalizar las propiedades de tolerancia a fallas como propiedades de *masking*, *nonmasking* ó *failsafe*, estos autores utilizan propiedades de *safety* y *liveness*, de hecho utilizaban lógica de primer orden para escribir dichas fórmulas. A diferencia de dCTL donde se combinan operadores temporales, con operadores deónticos para describir este tipo de propiedades de tolerancia a fallas. Creemos que la utilización de una lógica temporal ramificada contituye un punto beneficioso a la hora de especificar y verificar sistemas concurrentes o reactivos. En particular en lo que se refiere a la verificación automática donde se pueden encontrar muchos artículos que han demostrado su éxito al formalizar y verificar sistemas de *hardware* y otros sistemas críticos [24].

Otro lenguaje de especificación que utiliza un estilo similar al de nuestro trabajo es el lenguaje *Unity* [32], sin embargo nuestro lenguaje está especialmente diseñado para poder distinguir el comportamiento *normal* de los sistemas de aquellas situaciones en las cuales hay fallas presentes.

Por último, cabe destacar que hemos desarrollado una implementación del algoritmo de *model checking* para dCTL utilizando JAVA, el principal objetivo de este desarrollo fue poder evaluar a nivel práctico la *performance* de los algoritmos aquí presentados. Dicha herramienta se implementó utilizando una técnica conocida como *model checking simbólico* [52, 26, 24], la cual a diferencia del enfoque clásico, utiliza una representación abstracta de los estados y las transiciones del sistema, con el fin de mejorar la escalibilidad y performance de la misma; los cuales son dos puntos claves a la hora de verificar sistemas. El uso de fórmulas y *BDD* a la hora de representar programas concurrentes es bien conocido. En nuestro trabajo adoptamos la manera de construir las estructuras que representan el sistema similar a las ideas presentadas en la tesis doctoral de *Kenneth L. McMillan* [105].

7.1.1. Trabajos Relacionados Probabilistas

Años atrás se han desarrollaron varias investigaciones que perseguían el objetivo de lo lograr *standarizar* un lenguaje temporal de especificación de hardware [48, 44]. Estos trabajos fueron precedidos por muchas otras líneas de investigación en las cuales se discutían acerca de que operadores debían incluirse y cuales no, de manera de intentar balancear la facilidad de su utilización a la hora de realizar las especificaciones, su poder expresivo y el costo computacional de sus algoritmos de verificación [13, 49].

En este trabajo nosotros perseguimos un objetivo similar, nuestra idea es colaborar en este proceso de equilibrar estos aspectos pero para lógicas temporales, con el fin de poder obtener variantes lógicas que nos permitan razonar y verificar sistemas probabilísticos.

El lenguaje *standard* para razonar sobre este tipo de sistemas es PCTL. Esta lógica tiene como ventaja que la semántica de sus fórmulas es bastante intuitiva y la complejidad de su algoritmo de *model checking* es razonable. Sin embargo el poder expresivo de la misma es bastante limitado. Las versiones probabilísticas de μ-cálculo de *Mio* ó la definición de los autómatas probabilísticos de *Huth* y *Piterman* y *Wagner* [65] en 2012 son trabajos que contribuyen en este proceso de definición y análisis. Un punto a favor que que tienen estos formalismos definidos es que incrementan notablemente el poder expresivo de las mismas, logrando caracterizar cierto tipos de propiedades o patrones de repetición que antes no era posible capturar, pero tienen su debilidad en lo que se refiere a la facilidad de utilización y legibilidad de las fórmulas.

A través de los años han ido apareciendo diferentes variantes probabilísticas de

μ-cálculo.

Algunos de los trabajos para destacar son el de *Huth* y *Kwiatkowska* [64] y el de *McIver* y *Morgan* [103] ambos trabajos sugieren versiones de μ-cálculo cuantitativo y reemplazan la clásica interpretación *booleana* de μ-cálculo por una interpretación cuantitativa. Es decir, la semántica de una fórmula en lugar de definirse como un conjunto de estados donde dicha fórmula es verdadera, se la define a través de una función que asocia cada estado con un valor posible. Sin embargo estas lógicas fallan en capturar el poder expresivo de PCTL y no tienen un mecanismo definido que les permita volver las fórmulas al dominio *booleano*.

Mio [107] extiende estas lógicas con diferentes interpretaciones de la conjunción cuantitativa, con el objetivo de poder razonar acerca de diferentes versiones de conjunción. El autor introduce *juegos de paridad* (*parity games*) con productos independientes y *tree games*. En el caso particular de la lógica cuantitativa de *Mio* presentada en 2013 [108], esta nueva versión logra capturar la expresividad de PCTL y tiene un fragmento en el cual ciertas subfórmulas están bajo el dominio *booleano* y otras bajos el dominio cuantitativo. Desafortunadamente, la complejidad del *model checking* para la lógica de *Mio* es muy alta.

En nuestro trabajo [31] introdujimos μ^p-cálculo, el cual es un fragmento de la lógica probabilística μ-cálculo desarrollada por *Mio* [108]. Al igual que el algoritmo de *model checking* de μ-cálculo, la complejidad de nuestro procedimento de verificación está en NP $\cap$ co-NP. Sin embargo una de las desventajas que hemos aprendido a través del análisis de μ-cálculo, es que cuando se alternan puntos fijos la semántica de las fórmulas resultantes es bastante dificil de entender para el usuario. En el trabajo de *Liu* [91] analizan un fragmento de μPCTL.

En el capítulo 6 presentamos RPCTL, una extensión de PCTL con operadores recursivos, cuya semántica coincide con la de la lógica μ^p-cálculo [31], es decir está basada en máximos puntos fijos. La cual permite especificar propiedades describiendo posible "repeticiones" presentes en la cadena de *Markov* y la probabilidad de la ocurrencia de ciertos eventos dentro de estas repeticiones.

7.2. Contribuciones

A lo largo de los capítulos 3 y 4 hemos presentado una lógica temporal ramificada especialmente enriquecida para describir propiedades de sistemas tolerantes a fallas y un procedimiento de verificación para la misma. Además en el capítulo 5 analizamos a través de algunos casos de estudio la usabilidad de la lógica a la hora de especificar

propiedades de tolerancia a fallas y la performance del algoritmo de *model checking* implementado.

En particular, los operadores deónticos, son quienes nos ayudan a hacer la distinción entre el comportamiento normal y anormal del sistema, ya que los mismos nos proveen de una expresividad lo suficientemente *rica* para describir muchas de las propiedades de interés en el área de tolerancia a fallas. Demostramos que algunas fórmulas que pueden expresarse en nuestra lógica no pueden ser expresadas en otros fragmentos de CTL^*, incluida ECTL^+ y sus sublógicas. Y a pesar de incrementar el poder expresivo de la misma, el problema de *model checking* de nuestra lógica está en P, cosa que no sucede para ECTL^* y CTL^*, para las cuales este problema es *PSPACE-complete*.

Estos resultados, junto con nuestros argumentos acerca de la utilidad de la misma para la especificación de sistemas tolerantes a fallas, hacen de dCTL un fragmento interesante de CTL^*. Para poder exhibir justamente esta utilidad en el Capítulo 5, desarrollamos varios casos de estudio, que nos permitieron mostrar la expresividad de esta lógica. Antes, expresar propiedades temporales que predicaran acerca de situaciones tolerantes a fallas podía lograrse con enfoques con un nivel de abstracción mas bajo, es decir, directamente se referían a los estados con fallas a través de algunas fórmulas atómicas que justamente "marcaban" dichos estados. Creemos que los operadores deónticos aquí propuestos proveen una manera *indirecta*, o con un mayor nivel de abstracción para referirse a las fallas cuando se especifican propiedades de tolerancia a fallas, ya que capturan algunos *patrones* útiles en este contexto.

Si nos detenemos a analizar nuestra herramienta, y las principales características del algoritmo de *model checking* de dCTL. En primer lugar introdujimos un simple lenguaje de guardas, denominado *Faulty*, el cual es utilizado para especificar los modelos de sistemas tolerantes a fallas de una manera simple, permitiendo discriminar las propiedades que se preservan cuando el programa no presenta fallas.

Luego se detalló la manera adoptada para representar eficientemente la estructura de los programas *Faulty*. Esta tarea se lleva a cabo utilizando una serie de BDDs, los cuales permiten una representación simbólica de las distintas partes que componen la especificación del programa dado. Cabe destacar que en el caso de los cuantificadores existenciales utilizamos una técnica conocida como *Early Quantification* [105], en la cual estos cuantificadores son ubicados lo mas profundo posible dentro de la fórmula, distribuyéndolos con respecto a la *disjunción*. En nuestro caso, la estructura de los programas *Faulty* nos permite distribuirlos dentro de las guardas.

Por último presentamos el algoritmo de *model checking* simbólico para dCTL, el cual es un procedimiento recursivo que calcula para cualquier fórmula φ, su corres-

pondiente BDD $[\![\varphi]\!]$ representando aquellos estados que la satisfacen y luego verifica cuando la conjunción de los estados iniciales I y $[\![\varphi]\!]$ resulta satisfacible. Se dieron las definiciones para calcular cada fórmula de acuerdo a los operadores que involucra, sin embargo nos detuvimos mayormente a explicar la manera de computar aquellos BDDs correspondientes a los operadores Deónticos. Lo cual se basa principalmente en las semánticas de punto fijo de las fórmulas. Por otro lado cabe recalcar que además de dar las definiciones clásicas para calcular dichos operadores, también presentamos algunas variantes con el objetivo de mejorar la performance de cada algoritmo, las cuales se basan principalmente en la manera de inicializar los conjuntos utilizados para calcular los diferentes puntos fijos de acuerdo al operador, como así también siempre que es posible, calcular en ciclos independientes, y bajo demanda (es decir hacerlo sólo cuando se necesitan) aquellas fórmulas que se encuentran libres, es decir fuera del alcance de algún cuantificador.

A través de los ejemplos y casos de estudio que hemos utilizado para comparar la *performance* de nuestro algoritmo de *model checking* con respecto a NuSMV, el cual es un reconocido model checker simbólico para CTL. Hemos podido apreciar que nuestra herramienta ha obtenido mejores o iguales tiempos que NuSMV, en lo que se refiere al tiempo que demanda la traducción de la fórmula y verificación de la misma. Recordemos que en el capítulo 3 explicamos como se pueden traducir algunas de las fórmulas dCTL a CTL. La eficiencia de nuestra herramienta esta dada principalmente por dos razones, por un lado al utilizar una representación simbólica para los modelos y las fallas mejoramos la *performance* de lo algoritmos de punto fijo utilizados por el *model checker*. Por otro lado la estructura de nuestro lenguaje de modelado nos permitió aplicar la técnica de *early quantification*, la cual reduce el costo algorítmico del proceso de verificación.

En lo que se refiere a la incorporación de nociones de probabilidad para la verificación de propiedades relacionadas con *tolerancia a fallas* en el capítulo 6 presentamos RPCTL, la cual extiende a PCTL con llamadas recursivas. Esta lógica es un fragmento de μPCTL [31], la semántica del operador recursivo está dada por un máximo punto fijo. Sin embargo esta clase de abstracción sintáctica tiene el objetivo de aprovechar la familiaridad del concepto de recursión que está naturalizado en los usuarios que utilizan este tipo de formalismos para especificar propiedades de sistemas. Una restricción sintáctica que decidimos imponer es la de no permitir que se aniden mínimos y máximos puntos fijos en una misma fórmula, esta limitación nos asegura mantener la complejidad del algoritmo de *model checking* y a su vez ayuda con la legibilidad de las propiedades que se pueden expresar. Algo para destacar es que a pesar de estas restricciones a nivel de sintaxis el poder expresivo de la misma

supera el brindado por PCTL, ya que RPCTL permite caracterizar patrones de repetición en sistemas probabilísticos a diferencia de PCTL, esta extensión es ortogonal al poder expresivo que adiciona PCTL*, u otros mecanismos utilizados para describir propiedades sobre trazas.

Presentamos un análisis de la expresividad de dicha lógica en [30] y demostramos que RPCTL es más expresiva que PCTL. Demostramos que la complejidad de *model checking* coincide con los procedimientos de verificación de PCTL y es polinomial al tamaño de la cadena de Márkov subyacente. De hecho, el algoritmo invoca un número repetido de veces al procedimiento de *model checking* definido para PCTL.

Consideramos que la principal aplicación de RPCTL es la verificación de propiedades relacionadas a *tolerancia a fallas*. Recordemos que el grado de tolerancia exhibido por un sistema, se puede caracterizar a través de conjuntos de estados "seguros" y de acuerdo a como las ejecuciones del sistema permanecen dentro de esos conjuntos [15]. Por ejemplo un sistema se considera que es *fail-safe* si permanece en ese conjunto de estados seguros aún luego de la ocurrencia de fallas. Por otro lado se lo clasifica como *non-masking* cuando el sistema revisita infinitas oportunidades y con frecuencia esos conjuntos de estados deseables. Lo que ocurre en el caso de sistemas probabilísticos y la utilización de lógicas temporales para especificar propiedades sobre los mismos, es que este tipo de propiedades de tolerancia a fallas no pueden ser caracterizadas tan facilmente ni de manera directa. Debido principalmente a que la probabilidad de que un sistema permanezca siempre en un conjunto de estados puede volverse 0, cuando la ocurrencia de fallas tiene una porbabilidad positiva. Lo cual representa que a lo largo de las ejecuciones de un sistema, las mismas eventualmente se van a escapar de este conjunto de estados "buenos", con 1 de probabilidad. A lo largo del capítulo 6 analizamos este tipo de comportamientos a través de algunos ejemplos.

A pesar que la utilización de operadores recursivos en nuestra lógica constituye simplemente una abstracción sintáctica (lo cual en ingles denominamos comunmente como *syntactic sugar*), creemos que su incorporación mejora notablemente la usabilidad y legibilidad a la hora de especificar y verificar sistemas.

7.3. Conclusiones y Trabajos Futuros

En lo que se refiere a la lógica desarrollada, estamos analizando incluir algunos operadores deónticos alternativos que puedan proveer un poder expresivo equivalente a los operadores incluidos en la versión actual de dCTL, con el objetivo de agregar un

mayor nivel de abstración a nivel sintáctico para que las propiedades de tolerancia a fallas especificadas sean aún más intuitivas a la hora de interpretarlas para los usuarios.

Por otro lado, respecto a la herramienta desarrollada, tenemos planeado extenderla en varios aspectos. Una primera extensión es la de enriquecer el lenguaje de especificación *Faulty*, agregando algunas contrucciones relacionadas a tolerancia a fallas. Una de las ideas a desarrollar consiste en incluir a nivel gramatical contructores sintácticos que permitan capturar algunos conceptos que aparecen con frecuencia a la hora de caracterizar propiedades de tolerancia a fallas, un ejemplo de dichas propiedades es el modelo "*Heard-Of*" utilizado para describir algoritmos de consenso [33]. También estamos interesados en investigar maneras alternativas de mejorar la *performance* del tiempo de ejecución utilizado por nuestro *model checker* a la hora de verificar propiedades de tolerancia a fallas. En este sentido, una línea que planeamos investigar es la utilización de relaciones de simulación para capturar diferentes "niveles" de tolerancia a fallas para acotar el espacio de estados, básicamente la idea es utilizar estas relaciones de simulación como mecanismos de abstracción.

En el capítulo 5, si bien se ha analizado la *performance* de nuestra herramienta *FaultyCheck*, con respecto a otros *model checkers* como NuSMV. Un trabajo pendiente para realizar en el futuro, sería profundizar este análisis comparando el desempeño de *FaultyCheck* con otras herramientas como MCMAS. Cabe aclarar que NuSMV tiene una *performance* similar a la de MCMAS en lo que se refiere a la traducción de los modelos, según consta en [95].

Otra extesión que nos gustaría incorporar en nuestra herramienta es la generación de contraemplos, para poder brindarle al usuario información que resulte útil en aquellos escenarios donde la propiedad no se cumple y permitirle un mejor análisis. Si bien existen diversas técnicas para generar contraejemplos para fórmulas CTL [41] y las mismas se podrían aplicar para un subconjunto de fórmulas de nuestra lógica de manera directa, en el caso de fórmulas que involucran operadores deónticos ya no resulta tan trivial esta generación y es algo que nos gustaría analizar en un futuro.

Por último, en lo que se refiere al área de sistemas probabilistas, algo que quedó pendiente para trabajos futuros es la implementación del algoritmo de *model checking* para RPCTL [30] , con el objetivo de continuar las investigaciones con casos de estudios mas complejos y así poder analizar aún mejor las propiedades de este fragmento lógico. Otra línea de investigación para explorar es considerar agregar dentro de la cuantificación de probabilidades expresiones regulares, ya que consideramos que simplificaría aún mas la legibilidad de las propiedades.

Bibliografía

[1] *Faultycheck model checking for dctl - https://github.com/cl-unrc-lab/faulty.* 4, 4.2, 5.6.3, 5.7

[2] *Nusmv: a new symbolic model checker - http://nusmv.fbk.eu/.* 5.6

[3] Lennart Åqvist, *Good samaritans, contrary-to-duty imperatives, and epistemic obligations*, Noûs **1** (1967), no. 4, 361–379. 2.4.2

[4] Russell J. Abbott, *Resourceful systems for fault tolerance, reliability, and safety*, ACM Comput. Surv. **22** (1990), no. 1, 35–68. 1.1.3

[5] Zair Abdelouahab and Reginaldo Isaias Braga, *An adaptive train traffic controller*, Innovations and Advanced Techniques in Systems, Computing Sciences and Software Engineering (Dordrecht) (Khaled Elleithy, ed.), Springer Netherlands, 2008, pp. 550–555. 1.2

[6] Jean-Raymond Abrial, *Train systems*, Rigorous Development of Complex Fault-Tolerant Systems [FP6 IST-511599 RODIN project], 2006, pp. 1–36. 1.2

[7] ———, *Modeling in event-b - system and software engineering*, Cambridge University Press, 2010. 1.2

[8] C. E. Alchourron and E. Bulygin, *Normative systems*, Springer-Verlag, Wien; New York, 1971. 2.4.1

[9] Layman E. Allen and Gabriel Orechkoff, *Symbolic logic: A razor-edge tool for drafting and interpreting legal documents*, Journal of Symbolic Logic **29** (1964), no. 1, 43–44. 2.4.3

[10] P. E. Ammann and J. C. Knight, *Data diversity: an approach to software fault tolerance*, IEEE Transactions on Computers **37** (1988), no. 4, 418–425. 1

[11] T. Anderson and P. A. Lee, *Fault tolerance: Principles and practice*, Prentice Hall, 1981. 1.1.3

[12] Lennart Aqvist, *Deontic logic*, Handbook of Philosophical Logic: Volume II: Extensions of Classical Logic (D. Gabbay and F. Guenthner, eds.), Reidel, Dordrecht, 1984, pp. 605–714. 2.4.3

[13] R. Armoni, L. Fix, A. Flaisher, R. Gerth, B. Ginsburg, T. Kanza, A. Landver, S. Mador-Haim, E. Singerman, A. Tiemeyer, M.Y. Vardi, and Y. Zbar, *The forspec temporal logic: A new temporal property-specification language*, 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, Lecture Notes in Computer Science, vol. 2280, Springer, 2002, pp. 296–211. 7.1.1

[14] Anish Arora, *A foundation of fault-tolerant computing.*, Ph.D. thesis, The University of Texas at Austin, 1992. 1.2

[15] Anish Arora and Mohamed G. Gouda, *Closure and convergence: A foundation of fault-tolerant computing*, IEEE Trans. Software Eng. **19** (1993), no. 11, 1015–1027. 1.1, 1.4, 4, 5.2, 5.4, 6.1, 7.1, 7.2

[16] Anish Arora and Sandeep S. Kulkarni, *Detectors and correctors: A theory of fault-tolerance components*, Proceedings of the 18th International Conference on Distributed Computing Systems, Amsterdam, The Netherlands, May 26-29, 1998, 1998, pp. 436–443. 1.2

[17] Sanjeev Arora and Boaz Barak, *Computational complexity: A modern approach*, Cambridge University Press, 2009. 4.1.2

[18] Paul C. Attie, Anish Arora, and E. Allen Emerson, *Synthesis of fault-tolerant concurrent programs*, ACM Trans. Program. Lang. Syst. **26** (2004), no. 1, 125–185. 5.4.1, 6.3.1

[19] A. Avizienis, *The methodology of n-version programming*, ch. Ch. 2 in Software Fault Tolerance„ pp. 23–46, John Wiley & Sons Ltd. Editor M.R. Lyu., 1995. 1.1.3

[20] Algirdas Avizienis, *Design of fault-tolerant computers*, American Federation of Information Processing Societies: Proceedings of the AFIPS '67 Fall Joint Computer Conference, November 14-16, 1967, Anaheim, California, USA, 1967, pp. 733–743. 1.1

[21] ———, *The n-version approach to fault-tolerant software*, IEEE Trans. Software Eng. **11** (1985), no. 12, 1491–1501. 1

[22] Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, and Carl E. Landwehr, *Basic concepts and taxonomy of dependable and secure computing*, IEEE Trans. Dependable Sec. Comput. **1** (2004), no. 1, 11–33. 1.1.2

[23] Tolga Ayav, Pascal Fradet, and Alain Girault, *Implementing fault-tolerance in real-time programs by automatic program transformations*, ACM Transactions on Embedded Computing Systems (TECS) **7** (2008). 1.2

[24] Christel Baier and Joost-Pieter Katoen, *Principles of model checking*, MIT Press, 2008. 2.1.1, 2.2.1, 2.2.1, 2.2.2, 2.3.1, 2.3.2, 2.3.3, 2.6, 2.6.3, 3.2, 3.2.4, 4, 7.1

[25] Cinzia Bernardeschi, Alessandro Fantechi, and Stefania Gnesi, *Model checking fault tolerant systems*, Softw. Test., Verif. Reliab. **12** (2002), no. 4, 251–275. 1.2

[26] Jerry R. Burch, Edmund M. Clarke, Kenneth L. McMillan, David L. Dill, and L. J. Hwang, *Symbolic model checking: 10^20 states and beyond*, Proceedings of the Fifth Annual Symposium on Logic in Computer Science (LICS '90), Philadelphia, Pennsylvania, USA, June 4-7, 1990, 1990, pp. 428–439. 2.3.2, 4, 7.1

[27] P. F. Castro and T.S.E. Maibaum, *Deontic action logic, atomic boolean algebra and fault-tolerance*, Journal of Applied Logic **7** (2009), no. 4, 441–466. 1.2

[28] Pablo. Castro, *Deontic action logics for the specification and analysis of fault tolerance. phd thesis.*, Ph.D. thesis, McMaster University, Department of Computing and Software., 2009. 1.4, 7.1

[29] Pablo F. Castro, Cecilia Kilmurray, Araceli Acosta, and Nazareno Aguirre, *dctl: A branching time temporal logic for fault-tolerant system verification*, Software Engineering and Formal Methods - 9th International Conference,

SEFM 2011, Montevideo, Uruguay, November 14-18, 2011. Proceedings, 2011, pp. 106–121. 4.1.4

[30] Pablo F. Castro, Cecilia Kilmurray, and Nir Piterman, *A recursive probabilistic temporal logic*, Formal Methods and Software Engineering - 17th International Conference on Formal Engineering Methods, ICFEM 2015, Paris, France, November 3-5, 2015, Proceedings, 2015, pp. 336–348. 6.2.1, 7.2, 7.3

[31] ———, *Tractable probabilistic mu-calculus that expresses probabilistic temporal logics*, 32nd International Symposium on Theoretical Aspects of Computer Science, STACS 2015, March 4-7, 2015, Garching, Germany, 2015, pp. 211–223. 1.4, 2.6.3, 6.3, 6.4, 7.1.1, 7.2

[32] Jayadev Chandy, K. Mani y Misra, *Parallel program design: A foundation*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1988. 4, 4.1, 7.1

[33] B. Charron-Bost and A. Schiper, *The heard-of model: Computing in distrubuted systems with benign faults*, Distrib. Computing **22** (2009). 7.3

[34] Brian F. Chellas, *Modal logic: An introduction*, Cambridge University Press, 1980. 2.4.1, 2.4.2, 2.4.3

[35] Liming Chen and Algirdas Avizienis, *N-version programminc: A fault-tolerance approach to reliability of software operation*, Twenty-Fifth International Symposium on Fault-Tolerant Computing, 1995, ' Highlights from Twenty-Five Years'., 07 1995, pp. 113–. 1

[36] Timothy C. K. Chou, *Beyond fault tolerance*, IEEE Computer **30** (1997), no. 4, 47–49. 1.1.3

[37] F. Ciesinski and C. Baier, *LiQuor: A tool for qualitative and quantitative linear time analysis of reactive systems*, QEST, IEEE Computer Society, 2006, pp. 131–132. 1.4

[38] E. M. Clarke, E. A. Emerson, and A. P. Sistla, *Automatic verification of finite-state concurrent systems using temporal logic specifications*, ACM Transactions on Programming Languages and Systems **8** (1986), 244–263. 2.2.2, 2.3.1

[39] E. M. Clarke, O. Grumberg, and D.A. Peled, *Model Checking*, The MIT Press, 1999. 2.1, 3.2, 4

[40] Edmund M. Clarke and E. Allen Emerson, *Design and synthesis of synchronization skeletons using branching-time temporal logic*, Logics of Programs, Workshop, Yorktown Heights, New York, May 1981, 1981, pp. 52–71. 2.2.2, 2.3.1

[41] Edmund M. Clarke, Orna Grumberg, Kenneth L. McMillan, and Xudong Zhao, *Efficient generation of counterexamples and witnesses in symbolic model checking*, Proceedings of the 32st Conference on Design Automation, San Francisco, California, USA, Moscone Center, June 12-16, 1995, 1995, pp. 427–432. 7.3

[42] J. Coenen, *Formalisms for program reification and fault tolerance*, Ph.D. thesis, Tenische Universiteit Eindhoven, 1994. 1.2

[43] J.A.A. Coenen, *Specifying fault tolerant programs in deontic logic*, Computing science notes, Technische Universiteit Eindhoven, 1991 (English). 1.2

[44] B. Cohen, S. Venkataramanan, A. Kumari, and L. Piper, *System verilog assertions handbook*, VhdlCohen Publishing, 2010. 1.4, 7.1.1

[45] Flaviu Cristian, *A rigorous approach to fault-tolerant programming*, IEEE Trans. Software Eng. **11** (1985), no. 1, 23–31. 1.2

[46] Peter J. Denning, *Fault tolerant operating systems*, ACM Comput. Surv. **8** (1976), no. 4, 359–389. 1.1.3

[47] A.A. Martino (ed), *Deontic logic, computational linguistics and legal information systems*, First International Conference on Logic, Informatics, Law (A.A. Martino, ed.), vol. 2, North-Holland, Florence, 6-10 April 1981. 2.4.3

[48] C. Eisner and D. Fisman, *A practical introduction to psl*, Springer, 2006. 1.4, 7.1.1

[49] C. Eisner, D. Fisman, J. Havlicek, A. McIsaac, and D. Van Campenhout, *The definition of a temporal clock operator*, 30th International Colloquium on Automata, Languages and Programming, Lecture Notes in Computer Science, vol. 2719, Springer, 2003, pp. 857–870. 7.1.1

[50] E. Allen Emerson, *Temporal and modal logic*, Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics (B) (Jan van Leeuwen, ed.), MIT Press, 1990, pp. 995–1072. 2.2.1

[51] E. Allen Emerson and Joseph Y. Halpern, *"sometimes" and "not never" revisited: on branching versus linear time temporal logic*, J. ACM **33** (1986), no. 1, 151–178. 3.2.3, 3.4, 3.2.3

[52] Ernest Allen Emerson, *The beginning of model checking: A personal perspective*, 25 Years of Model Checking - History, Achievements, Perspectives, 2008, pp. 27–45. 2.3.1, 2.3.2, 2.3.3, 7.1

[53] Jonathan Ezekiel and Alessio Lomuscio, *Combining fault injection and model checking to verify fault tolerance in multi-agent systems*, 8th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2009), Budapest, Hungary, May 10-15, 2009, Volume 1, 2009, pp. 113–120. 1.2

[54] Ronald Fagin, *Reasoning about knowledge*, MIT Press, 1995. 1.4, 7.1

[55] José Luiz Fiadeiro and T. S. E. Maibaum, *Temporal reasoning over deontic specifications*, J. Log. Comput. **1** (1991), no. 3, 357–395. 1.4

[56] Tim French, John Christopher McCabe-Dansted, and Mark Reynolds, *A temporal logic of robustness*, Frontiers of Combining Systems, 6th International Symposium, FroCoS 2007, Liverpool, UK, September 10-12, 2007, Proceedings, 2007, pp. 193–205. 1.2

[57] F. Gärtner, *Specification for fault-tolerance: A comedy of failures*, Tech. report, Darmstadt University of Technology, 1998. 1.2

[58] Felix Gärtner, *Transformational approaches to the specification and verification of fault-tolerant systems: Formal background and classification.*, J. UCS **5** (1999), 668–692. 1.2

[59] Felix C. Gärtner, *Fundamentals of fault-tolerant distributed computing in asynchronous environments*, ACM Comput. Surv. **31** (1999), no. 1, 1–26. 2.5, 7.1

[60] Stefania Gnesi, Gabriele Lenzini, and Fabio Martinelli, *Logical specification and analysis of fault tolerant systems through partial model checking*, Electr. Notes Theor. Comput. Sci. **118** (2005), 57–70. 1.2

[61] Lou Goble, *Murder most gentle: The paradox deepens*, Philosophical Studies **64** (1991), no. 2, 217–227. 2.4.2

[62] Claude Hennebert and Gérard D. Guiho, *SACEM: A fault tolerant system for train speed control*, Digest of Papers: FTCS-23, The Twenty-Third Annual International Symposium on Fault-Tolerant Computing, Toulouse, France, June 22-24, 1993, 1993, pp. 624–628. 1.2

[63] A. Hinton, M.Z. Kwiatkowska, G. Norman, and D. Parker, *PRISM: a tool for automatic verification of probabilistic systems*, TACAS, Lecture Notes in Computer Science, vol. 3920, Springer-Verlag, 2006. 1.4

[64] M. Huth and M.Z. Kwiatkowska, *Quantitative analysis and model checking*, 12th IEEE Symposium on Logic in Computer Science, IEEE Computer Society, 1997, pp. 111–122. 6.4, 7.1.1

[65] Michael Huth, Nir Piterman, and Daniel Wagner, *p-automata: New foundations for discrete-time probabilistic verification*, Perform. Eval. **69** (2012), no. 7-8, 356–378. 1.4, 2.6.4, 7.1.1

[66] Michael Huth and Mark Dermot Ryan, *Logic in computer science: Modelling and reasoning about systems*, Cambridge University Press, 2000. 2.2.2

[67] IEEE.org, *Ieee standard glossary of software engineering terminology*, Dec 1990. 1.2

[68] Jaakko K. Hintikka J., *Quantifiers in deontic logic.*, Societas Scientiarum Fennica, Commentationes humanarum-litterarum, **vol. 23** (Helsingfors 1957), no. no. 4, 23 pp. 2.4.1

[69] Daniel Jackson and Allison Waingold, *Lightweight extraction of object models from bytecode*, IEEE Trans. Software Eng. **27** (2001), no. 2, 156–169. 1.2

[70] Tomasz Janowski, *On bisimulation, fault-monotonicity and provable fault-tolerance*, Algebraic Methodology and Software Technology, 6th International Conference, AMAST '97, Sydney, Australia, December 13-17, 1997, Proceedings, 1997, pp. 292–306. 2.3

[71] Andrew J. I. Jones, *Deontic logic and legal knowledge representation*, Ratio Juris **3** (1990), no. 2, 237–244. 2.4.3

[72] Andrew J. I. Jones and Ingmar Pörn, *Ideality, sub-ideality and deontic logic*, Synthese **65** (1985), no. 2, 275–290. 2.4.2

[73] Andrew J.I. Jones and Marek Sergot, *Deontic logic in the representation of law: Towards a methodology*, Artificial Intelligence and Law **1** (1992), no. 1, 45–64 (English). 2.4.3

[74] H. C. Brearley Jr., *ILLIAC II-A short description and annotated bibliography*, IEEE Trans. Electronic Computers **14** (1965), no. 3, 399–403. 5.4

[75] Eunsuk Kang and Daniel Jackson, *Formal modeling and analysis of a flash filesystem in alloy*, Abstract State Machines, B and Z, First International Conference, ABZ 2008, London, UK, September 16-18, 2008. Proceedings, 2008, pp. 294–308. 1.2

[76] S. Kanger, *New foundations for ethical theory*, New Foundations for Ethical Theory, no. v. 1, Almqvist & Wiksell, 1957 (reprinted 1971). 2.4.1

[77] S. Khosla, *System specification: A deontic approach. phd thesis.*, Ph.D. thesis, Imperial College, 1988. 1.4, 7.1

[78] Panagiotis Kouvaros and Alessio Lomuscio, *Verifying fault-tolerance in parameterised multi-agent systems*, Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI 2017, Melbourne, Australia, August 19-25, 2017, 2017, pp. 288–294. 1.2

[79] Saul A. Kripke, *Semantical considerations on modal logic*, Acta Philosophica Fennica **16** (1963), 83–94. 2.1

[80] Sandeep S. Kulkarni and Anish Arora, *Automating the addition of fault-tolerance*, Formal Techniques in Real-Time and Fault-Tolerant Systems, 6th International Symposium, FTRTFT 2000, Pune, India, September 20-22, 2000, Proceedings, 2000, pp. 82–93. 1.2

[81] Marta Kwiatkowska, Gethin Norman, and David Parker, *Quantitative analysis with the probabilistic model checker prism*, Electron. Notes Theor. Comput. Sci. **153** (2006), no. 2, 5–31. 1.4

[82] Leslie Lamport and Stephan Merz, *Specifying and verifying fault-tolerant systems*, Formal Techniques in Real-Time and Fault-Tolerant Systems, Third International Symposium Organized Jointly with the Working Group Provably Correct Systems - ProCoS, Lübeck, Germany, September 19-23, Proceedings, 1994, pp. 41–76. 1.2

[83] Leslie Lamport, Robert E. Shostak, and Marshall C. Pease, *The byzantine generals problem*, ACM Trans. Program. Lang. Syst. **4** (1982), no. 3, 382–401. 1.2

[84] L. A. Laranjeira, M. Malek, and R. M. Jenevein, *Nest: A nested-predicate scheme for fault tolerance*, IEEE Trans. Computers **42** (1993), 1303–1324. 1.2

[85] François Laroussinie, Nicolas Markey, and Philippe Schnoebelen, *Temporal logic with forgettable past*, 17th IEEE Symposium on Logic in Computer Science (LICS 2002), 22-25 July 2002, Copenhagen, Denmark, Proceedings, 2002, pp. 383–392. 2.2.1

[86] Gabriele Lenzini, Fabio Martinelli, Ilaria Matteucci, and Stefania Gnesi, *A uniform approach to security and fault-tolerance specification and analysis*, Architecting Dependable Systems VI, 2008, pp. 172–201. 1.2

[87] Nancy G. Leveson, *Safeware - system safety and computers: a guide to preventing accidents and losses caused by technology*, Addison-Wesley, 1995. (document), 1.1.3

[88] Orna Lichtenstein, Amir Pnueli, and Lenore D. Zuck, *The glory of the past*, Logics of Programs, Conference, Brooklyn College, June 17-19, 1985, Proceedings, 1985, pp. 196–218. 2.2.1

[89] Yiyan Lin, Borzoo Bonakdarpour, and Sandeep S. Kulkarni, *Automated addition of fault-tolerance under synchronous semantics*, Stabilization, Safety, and Security of Distributed Systems - 15th International Symposium, SSS 2013, Osaka, Japan, November 13-16, 2013. Proceedings, 2013, pp. 266–280. 1.2

[90] Barbara Liskov and John Guttag, *Program development in java: Abstraction, specification, and object-oriented design*, Addison-Wesley Professional, 2000. 4.2

[91] W. Liu, L. Song, J. Wang, and L. Zhang, *A simple probabilistic extension of modal μ-calculus*, 23rd International Joint Conference on Artificial Intelligence (Buenos Aires, Argentina), Lecture Notes in Computer Science, Springer-Verlag, 2015, pp. 882–888. 6.4, 7.1.1

[92] Z. Liu and M. Joseph, *Transformation of programs for fault-tolerance*, Formal Aspects of Computing **4** (1992). 1.2

[93] Z. Liu and M. Joseph, *Specification and verification of fault-tolerance, timing, and scheduling*, ACM Trans. Program. Lang. Syst. **21** (1999), no. 1, 46–89. 1.2

[94] A. Lomuscio and M. J. Sergot, *A formalisation of violation, error recovery, and enforcement in the bit transmission problem*, Journal of Applied Logic **2** (2004), 93–116. 1.2

[95] Alessio Lomuscio, Hongyang Qu, and Franco Raimondi, *MCMAS: an open-source model checker for the verification of multi-agent systems*, STTT **19** (2017), no. 1, 9–30. 1.4, 7.1, 7.3

[96] Alessio Lomuscio and Franco Raimondi, *MCMAS: A model checker for multi-agent systems*, Tools and Algorithms for the Construction and Analysis of Systems, 12th International Conference, TACAS 2006 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2006, Vienna, Austria, March 25 - April 2, 2006, Proceedings, 2006, pp. 450–454. 1.2, 1.4, 7.1

[97] J. Magee and L. Kramer, *Concurrency: State models and java programs*, Wiley, 2006. 5.5

[98] Jeff Magee and Tom Maibaum, *Towards specification, modelling and analysis of fault tolerance in self managed systems*, Proceedings of the 2006 international workshop on Self-adaptation and self-managing systems, SEAMS 2006, Shanghai, China, May 21-22, 2006, 2006, pp. 30–36. 1.2, 1.4

[99] T.S.E. Maibaum, *A logic for the formal requirements specification. technical report.*, Tech. report, Imperial College, 1987. 1.4, 7.1

[100] Ernst Mally, *The basic laws of ought: Elements of the logic of willing*, vol. viii+85pp, Graz: Leuschner und Lubensky, Universitats-Buchhandlung, 1926. 2.4.1

[101] Zohar Manna and Amir Pnueli, *The temporal logic of reactive and concurrent systems - specification*, Springer, 1992. 2.2.1

[102] John McCabe-Dansted, *A temporal logic of robustness*, Ph.D. thesis, The University of Western Australia, 2011. 7.1

[103] A. McIver and C. Morgan, *Results on the quantitative μ-calculus $qM\mu$*, ACM Trans. Comput. Log. **8** (2007), no. 1. 6.4, 7.1.1

[104] Kenneth L. McMillan, *The smv system.*, Tech. report, Technical Report, 1992. 1.2

[105] Kenneth L. McMillan, *Symbolic model checking*, Ph.D. thesis, CarnegieMellonUniversity, 1992. 1.2, 4.1, 4.1.2, 4.3, 7.1, 7.2

[106] John-Jules Ch Meyer, Franciscus Petrus Maria Dignum, and Roelf Johannes Wieringa, *The paradoxes of deontic logic revisited: A computer science perspective*, Tech. Report Technical Report UU-CS-1994-38, University of Utrecht, 1994. 2.4.2

[107] M. Mio, *Game semantics for probabilistic μ-calculi*, Ph.D. thesis, University of Edinburgh, 2012. 1.4, 6.4, 7.1.1

[108] M. Mio and A. Simpson, *Łukasiewicz μ-calculus*, FICS, 2013. 6.4, 7.1.1

[109] Till Mossakowski, Razvan Diaconescu, and Andrzej Tarlecki, *What is a logic translation?*, Logica Universalis **3** (2009), no. 1, 95–124. 3.2, 3.5

[110] Peter L. Mott, *On chisholm's paradox*, Journal of Philosophical Logic **2** (1973), no. 2, 197–211. 2.4.2

[111] Kupferman O. and Grumberg O., *Buy one, get one free!!!*, **6** (1996), 523–523. 3.2.3

[112] Wojciech Penczek and Alessio Lomuscio, *Verifying epistemic properties of multi-agent systems via bounded model checking*, Fundam. Inform. **55** (2003), no. 2, 167–185. 1.4, 7.1

[113] Amir Pnueli, *The temporal logic of programs*, 18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977, 1977, pp. 46–57. 2.2.1

[114] A. N. Prior, *The paradoxes of derived obligation*, Mind **63** (1954), no. 249, 64–65. 2.4.2

[115] Laura L. Pullum, *Software fault tolerance techniques and implementation*, Artech House, Inc., Norwood, MA, USA, 2001. 1.1.2, 1.1.3

[116] P. A. G. Sivilotti: S. M. Pike, *Dining philosophers with crash locality 1*, 24th International Conference on Distributed Computing Systems, IEEE Computer Society, 2004, pp. 22–29. 5.5

[117] Francis Schneider, Steve M. Easterbrook, John R. Callahan, and Gerard J. Holzmann, *Validating requirements for fault tolerant systems using model checking*, 3rd International Conference on Requirements Engineering (ICRE '98), Putting Requirements Engineering to Practice, April 6-10, 1998, Colorado Springs, CO, USA, Proceedings, 1998, pp. 4–13. 1.2

[118] Klaus Schneider, *Verification of reactive systems - formal methods and algorithms*, 1862-4499, Springer-Verlag Berlin Heidelberg, 2004. 3.2.3, 4.1.3

[119] Daniel P. Siewiorek and Robert S. Swarz, *Reliable computer systems - design and evaluation (3. ed.)*, A K Peters, 1998. 1.1.3

[120] Walter Sinnott-Armstrong, *A solution to forrester's paradox of gentle murder*, Journal of Philosophy **82** (1985), no. 3, 162–168. 2.4.2

[121] Ilina Stoilkovska, Igor Konnov, Josef Widder, and Florian Zuleger, *Verifying safety of synchronous fault-tolerant algorithms by bounded model checking*, Tools and Algorithms for the Construction and Analysis of Systems (Cham) (Tomáš Vojnar and Lijun Zhang, eds.), Springer International Publishing, 2019, pp. 357–374. 1.2

[122] Neeraj Suri, Chris J. Walter, and Michelle M. Hugue, *Advances in ultra-dependable distributed systems*, IEEE Computer Society Press, Los Alamitos, CA, USA, 1994. 1.1.1

[123] W. Torres-Pomales, *Software fault tolerance: A tutorial.*, Technical report TM-2000-210616., NASA Technical Memorandum, 2000. 1.1, 1.1.2, 1.1.3, 1.1.3

[124] G. H. von Wright, *I. deontic logic*, Mind **60** (1951), no. 237, 1–15. 2.4.1

[125] R. J. Wieringa and J.-J. Ch. Meyer, *Applications of deontic logic in computer science: A concise overview*, Deontic Logic in Computer Science (John-Jules Ch. Meyer and Roel J. Wieringa, eds.), John Wiley and Sons Ltd., Chichester, UK, 1993, pp. 17–40. 2.4.3

[126] T.-Y. Wuu and Sarma B. K. Vrudhula, *A design of a fast and area efficient multi-input muller c-element*, IEEE Trans. VLSI Syst. **1** (1993), no. 2, 215–219. 5.4

[127] Tomoyuki Yokogawa, Tatsuhiro Tsuchiya, and Tsuchiya Kikuno, *Automatic verification of fault tolerance using model checking*, 8th Pacific Rim International Symposium on Dependable Computing (PRDC 2001), 17-19 December 2001, Seoul, Korea, 2001, pp. 95–102. 1.2

[128] K. Saluja Z. Xie, H. Sun, *A survey of software fault tolerance techniques*, Madison, WI, USA **vol. 1415, 2006.** (2006). 1.1.3