UNIVERSIDAD
DE LA REPÚBLICA
URUGUAY

FACULTAD DE
INGENIERÍA
UDELAR

PEDECIBA INFORMÁTICA

Instituto de Computación, Facultad de Ingeniería
Universidad de la República
Montevideo, Uruguay

TESIS DE DOCTORADO EN INFORMÁTICA

Metodología de Verificación, Detección de Errores y Generación de Casos de Prueba en Modelos RT-DEVS con Requisitos Temporales Cuantitativos

Ariel Gonzalez
agonzalez@dc.exa.unrc.edu.ar

Mayo de 2025

Directores de Tesis: Maximiliano Cristiá[1] y Carlos Luna [2]

[1] Universidad Nacional de Rosario y CIFASIS
cristia@cifasis-conicet.gov.ar

[2] InCo, Facultad de Ingeniería, Universidad de la República
cluna@fing.edu.uy

Resumen

Los sistemas de tiempo real suelen presentar errores vinculados a la temporización que son difíciles de detectar. Los formalismos utilizados para modelar este tipo de sistemas permiten especificar comportamientos a diferentes niveles de abstracción, descomponiendo los modelos en submodelos (componentes), los cuales interactúan entre sí para describir correctamente el comportamiento global del sistema. Los requisitos temporales cuantitativos añaden una complejidad adicional a estos modelos, que pueden inducir a los diseñadores a cometer errores, especialmente en aspectos temporales durante la construcción.

En sistemas grandes, el número de componentes suele ser considerable, por lo que es necesario contar con herramientas que automaticen la tarea de verificar si el modelo cumple con los requisitos del problema.

DEVS (*Discrete Event System Specification*) es un formalismo matemático utilizado para modelar sistemas dinámicos discretos, especialmente aquellos que evolucionan a lo largo del tiempo debido a eventos que ocurren en momentos específicos. Estos modelos, y en particular los de la variante RealTime-DEVS (RT-DEVS), permiten especificar sistemas con requisitos temporales cuantitativos, pero carecen de un mecanismo de verificación formal (*model checkers, SAT Solvers, etc.*) que posibilite la comprobación de este tipo de requisitos.

En esta tesis se propone un mecanismo que, por un lado, permite verificar con un *model checker* si los modelos RT-DEVS cumplen con un conjunto de propiedades recurrentes (patrones) en los sistemas, y por otro lado, basado en mutantes de propiedades temporales, ayuda a descubrir errores en los sistemas cuando no se satisfacen tales propiedades. En este trabajo, utilizamos el *model checker* Uppaal tanto para verificar un conjunto de patrones de propiedades temporales (Time-Bounded Response, Precedence with Delay, Time-Restricted Precedence, Conditional Security, Time-Bounded Frequency, entre otros), como para detectar errores de temporización mediante mutantes de dichos patrones en modelos RT-DEVS. Aunque Uppaal no admite directamente el análisis de estas propiedades, proponemos un proceso de transformaciones y empleamos la técnica del autómata observador para habilitar dicho análisis en Uppaal.

Además, se define un mecanismo para generar casos de prueba a partir de los patrones y sus mutantes, que complementa las estrategias convencionales en la ingeniería de software.

A lo largo del trabajo, se introduce y analiza un caso de estudio sobre un sistema de cruce de trenes.

Palabras clave: Sistemas de Tiempo Real, Real-Time DEVS, Uppaal, Chequeo de Modelos, Testing por Mutación, Mutación de Propiedades.

A Methodology for Verification, Error Detection, and Test Case Generation in RT-DEVS Models with Quantitative Temporal Requirements

Abstract

Real-time systems often have timing-related errors that are hard to detect. The formalisms used to model these systems allow for the specification of behaviors at different levels of abstraction by decomposing models into submodels (components), which interact with one another to accurately describe the overall system behavior. Quantitative temporal requirements introduce additional complexity into these models, potentially causing engineers to introduce errors, particularly regarding timing aspects during development.

In large-scale systems, the number of components tends to be substantial, hence the need for tools that automate the verification of models with respect to the system requirements.

DEVS (Discrete Event System Specification) is a mathematical formalism used to model discrete dynamic systems, especially those that evolve over time due to events occurring at specific moments. These models, and in particular those of the Real-Time DEVS (RT-DEVS) variant, allow for the specification of systems with quantitative temporal requirements, but they lack a formal verification mechanism (*model checkers, SAT solvers, etc.*) to enable the validation of such requirements.

In this thesis, a mechanism is proposed that, on one hand, allows verifying with a *model checker* whether RT-DEVS models satisfy a set of recurrent properties (patterns) in systems. On the other hand, based on temporal property mutants, helps to uncover errors in the systems when such properties are not satisfied.

In this work, we use the *model checker* Uppaal both to verify a set of temporal property patterns (Time-Bounded Response, Precedence with Delay, Time-Restricted Precedence, Conditional Security, Time-Bounded Frequency, among others), and to detect timing errors through mutants of these patterns in RT-DEVS models. Even though, in general, Uppaal cannot handle this kind of properties, we propose a transformation process and employ the observer automaton technique to enable such analysis.

Additionally, a mechanism is defined to generate test cases from the patterns and their mutants, complementing conventional strategies in software engineering.

Throughout the work, a case study from the railway domain is introduced and analyzed.

Keywords: Real-Time System, Real-Time DEVS, Uppaal, Model Checking, Mutation Testing, Property Mutation.

Agradecimientos

MUCHAS GRACIAS a Maxi y Carlos (directores de esta tesis), por su invaluable aporte de conocimientos, el tiempo dedicado a lo largo de todo este proceso y su constante compromiso para asegurar que el trabajo realizado ofreciera contribuciones significativas en las áreas de mi interés.

De la misma forma, agradezco profundamente a mis supervisores María Urquhart (Marita) y Héctor Cancela por su confianza, acompañamiento y seguimiento a lo largo de todo el proceso.

A Carlos (Luna), quien, al igual que en la Maestría, fue el motor de toda mi carrera doctoral. Desde los 18 años nos une una valiosa relación académica y una entrañable amistad. ¡Enormes gracias!

Y, principalmente, a mi hermosa y gran familia – y en especial a mi esposa – por su apoyo incondicional en cada uno de estos objetivos. Al igual que en ocasiones anteriores, este logro no es solo personal, sino un verdadero triunfo compartido por toda la familia.

Contenido

List of Figures

Capítulo 1

Introducción

La industria es generadora de productos que embeben sistemas críticos y dinámicos: el sistema de frenado de un auto, los sistemas de control inteligente de ascensores, las cajas de cambio automáticas, la robotización en la industria automotriz y una gran variedad de dispositivos utilizados en los centros de salud son algunos ejemplos. Las pruebas de software constituyen una parte muy importante del proceso de desarrollo de sistemas. Se estima que el 50% del esfuerzo total está dedicado a pruebas y depuración [1, 2, 3]. En la industria, las pruebas a menudo se realizan de manera *ad hoc*, sin una base teórica sólida. En particular, [4, 5] reportan casos con consecuencias significativas debido a errores encontrados en los sistemas de software. La disponibilidad de una especificación formal permite convertir las pruebas en un proceso bien definido, respaldado por una teoría sólida.

Los sistemas críticos, y en especial los de tiempo real [6], deben validarse muy minuciosamente. Las técnicas de validación que no están basadas en métodos sistemáticos de generación de casos de prueba suelen dar malos resultados en términos de detección de errores y cobertura. Dado que las técnicas de validación basadas en métodos formales permiten la generación sistemática de casos de prueba, constituyen una buena alternativa. En particular, una especificación formal proporciona una descripción precisa de los requerimientos de un sistema, que facilita la aplicación e implementación de técnicas de verificación automática. Como ejemplos podemos citar:

- El chequeo de modelos (*model checking* - MC[1]) [7], en donde un sistema se comprueba algorítmicamente frente a requisitos codificados en una lógica temporal (Uppaal [8], SPIN [9], NuSMV [10]).
- El testing basado en modelos (*Model-Based Testing* - MBT), en donde la especificación formal se usa para obtener casos de prueba con los cuales testear una implementación [11, 12].
- El problema de satisfacibilidad booleana, es un algoritmo (Satisfiability solver - SAT solver) [13] que se enfoca en resolver el problema de determinar si existe una interpretación que satisfaga una fórmula booleana dada (Alloy [14], MiniSat [15]).
- El probador de teoremas, es un programa que permite demostrar teoremas, de manera en general asistida, según un lenguaje y conjunto de reglas predefinidas (COQ [16], Isabelle [17]).

En esta tesis nos enfocamos en sistemas que no sólo deben producir resultados o reacciones correctas, sino que deben hacerlo en el momento correcto; ni demasiado pronto ni demasiado tarde. La reacción oportuna es tan importante como el tipo de reacción. Este tipo de sistemas

[1]En adelante, usaremos también *MC* para referirnos a un *model checker*.

son conocidos como sistemas de tiempo real. En la ultima década el crecimiento de los sistemas embebidos de tiempo real ha sido vertiginoso. En algunos casos son aplicados en contextos en donde los riesgos y costos son críticos y, en consecuencia, han causado nuevos y profundos estudios en las actividades de verificación y testing de software [18, 19, 20, 21].

En algunos casos, el incumplimiento de una restricción temporal (vencimiento de una actividad) suele tener consecuencias menores (*restricciones temporales flexibles*[2]). Por ejemplo, si el requisito temporal "cuando las puertas del automóvil se cierran los vidrios deben cerrarse en a lo sumo 10 segundos" se cumple unos segundos más tarde, la falla no es grave. Mientras que en otros sistemas se necesita que el mismo responda dentro de los límites de tiempo definidos, pues de lo contrario las consecuencias podrían ser muy costosas (*restricciones temporales rígidas*[3]). Dentro de la industria existe un conjunto importante de sistemas cuyos requisitos temporales son rígidos. Por ejemplo, si el sistema de frenado de un auto moderno reacciona demasiado tarde debido a un error en el software de control podría ser causa de accidentes; de la misma manera un error de software que hace que un dispositivo utilizado en las salas de terapia intensiva reaccione segundos más tarde de lo debido podría conducir a situaciones que ponen en riesgo la vida de un paciente.

Los métodos tradicionales de testing a veces no alcanzan para garantizar la ausencia de ciertas situaciones no deseables que tienen que ver no solo con la lógica del sistema sino también con los tiempos de respuesta. En [22] el autor describe los veinticinco errores más comunes en los sistemas de tiempo real.

Los procesos de desarrollo de software incluyen varias etapas cruciales para asegurar un buen desarrollo. Una de estas etapas es el modelado de los sistemas, que permite construir una representación abstracta del sistema, incorporando los aspectos más relevantes. Posteriormente, se utilizan diversas herramientas para analizar y descubrir fallas en las primeras etapas del desarrollo.

Existen diversos lenguajes para modelar (especificar) formalmente este tipo sistemas—redes de petri temporizadas [23], el formalismo DEVS [24], autómatas temporizados (Timed Automata - TA) [25], etc. Además, existen herramientas de software que facilitan la especificación y verificación de los modelos escritos en esos lenguajes. El formalismo RealTime-DEVS (RT-DEVS) propuesto por Hong [26], es una variante de los DEVS clásicos que permite la especificación de sistemas de tiempo real. Para especificar ciertos sistemas, la función avance del tiempo en RT-DEVS retorna un intervalo de números reales dado que el tiempo de ocurrencia de un evento suele estar mejor representado por un intervalo más que por un instante específico. Esta es una característica recurrente en los sistemas de tiempo real. Incluso en sistemas de tiempo real con restricciones temporales rígidas, se espera que las respuestas ocurran dentro de un intervalo de tiempo, en lugar de hacerlo en un instante específico [27]. Esto se debe a que, el hardware podría no ser capaz de generar una determinada respuesta en un instante de tiempo específico porque puede estar ocupado recibiendo entradas o produciendo otras salidas. Las restricciones temporales expresadas como eventos que ocurren en intervalos de tiempo precisos se denominan *requisitos temporales cuantitativos* (RTC) o *propiedades temporales cuantitativas* (PTC).

Por otro lado, también existen diversos lenguajes para especificar las propiedades temporales que debe satisfacer un sistema. Los lenguajes lógicos tales como la *Linear Temporal Logic (LTL)* [28] y *Computation Tree Logic* (CTL) [28, 29] son muy útiles para describir propiedades de sistemas de tiempo real. Una de las principales ventajas de estas lógicas es que son soportadas por modernos MC [30, 9].

Estas lógicas no admiten la especificación literal del tiempo, es decir, no es posible especificar que ciertos sucesos deben ocurrir dentro de lapsos o intervalos de tiempo concretos; dicho de otro

[2] soft timing constraints
[3] hard timing constraints

modo, no permiten especificar RTC. En cambio, las *Timed Computational Tree Logic* (TCTL) [31] y *Metric Temporal Logic (MTL)* [32] están diseñadas para la especificación de PTC, y en el mejor de los casos, están parcialmente soportadas por *model checkers*.

En general, la comunidad DEVS valida los modelos mediante simulación. En la actualidad existen algunas herramientan de simulación que soportan diversas variantes del formalismo DEVS [33]. Las simulaciones son adecuadas para recrear y comprender el comportamiento de un sistema, e incluso para verificar ciertas propiedades, pero no son suficientes para asegurar la corrección de un modelo respecto a una propiedad. Para garantizar que un sistema no viola una propiedad deberíamos simular todas las ejecuciones posibles, algo que en general no es posible. Por el contrario, los MC permiten verificar exhaustivamente ciertas propiedades utilizando resultados de la lógica temporal. Sin embargo, los MC pueden enfrentarse al problema de explosión de estados [34], momento en el cual la simulación es una herramienta alternativa.

Por este motivo, en este trabajo proponemos una técnica para la verificación de algunas clases de RTC expresadas en RT-DEVS y MTL mediante el uso del MC Uppaal, el cual soporta parcialmente TCTL. Dado que Uppaal no admite directamente el análisis de RTC, proponemos un proceso en el que un modelo RT-DEVS se transforma en un autómata que es controlado por un autómata observador [35, 36]. Además, cuando el modelo RT-DEVS no cumple con un RTC, proponemos un método que ayuda al ingeniero a detectar el error en el modelo (o en el RTC) mediante mutaciones de propiedades, las cuales buscan capturar posibles errores que un ingeniero podría cometer.

Todavía existe una brecha entre un modelo que se verifica como una entidad abstracta y el código real que se ejecuta en una plataforma de destino. Los errores aún podrían aparecer en la implementación final a medida que el programador implementa los requisitos capturados y modelados. Varias etapas y actividades del proceso de testing de software pueden automatizarse y no requieren recursos humanos del más alto nivel técnico. Sin embargo, el testing solo puede aumentar la confianza en la corrección del programa; no sirve para *demostrar* su corrección. Las pruebas de software garantizarían la corrección si se hiciera de manera exhaustiva, pero las pruebas exhaustivas no son factibles en general. El MBT combina las bondades de los métodos formales con las del *testing* tradicional. Usando MBT se busca determinar (automáticamente) si una implementación dada cumple con su especificación, donde la noción de "cumplimiento" no garantiza la corrección pero es mucho más rigurosa que en los contextos clásicos de testing.

Por esta razón, esta tesis también describe cómo generar casos de prueba usando el modelo RT-DEVS para testear la implementación. La generación de casos de prueba se realiza de forma automática usando Uppaal.

1.1 Contribuciones

La principal contribución de esta tesis es la verificación de propiedades temporales en modelos RT-DEVS. Nos basamos en traducciones existentes de modelos RT-DEVS a TA [37, 38, 21] y aprovechamos la expresividad de Uppaal para implementar dichas propiedades. Los métodos propuestos para traducir RT-DEVS a TA, que luego son utilizados por Uppaal, no permiten verificar PTC. Por esta razón, en este trabajo utilizamos los TA definidos por Alur [39, 25], que Uppaal extiende con nuevos elementos que nos permiten modelar las PTC. Estos TA permiten describir RTC que pueden ser codificadas en Uppaal.

Los aportes de este trabajo pueden entenderse mejor al observar la Figura 1.1. Los ingenieros escriben un modelo RT-DEVS que debe verificar ciertas PTC. Proponemos transformar ambos en TA [39, 25], los cuales se introducen en el MC Uppaal, que puede probar automáticamente si el modelo verifica o no las PTC. Sin embargo, dado que las PTC pueden ser difíciles de formalizar, se proporciona un conjunto de patrones de fórmulas temporales (recuadro de línea

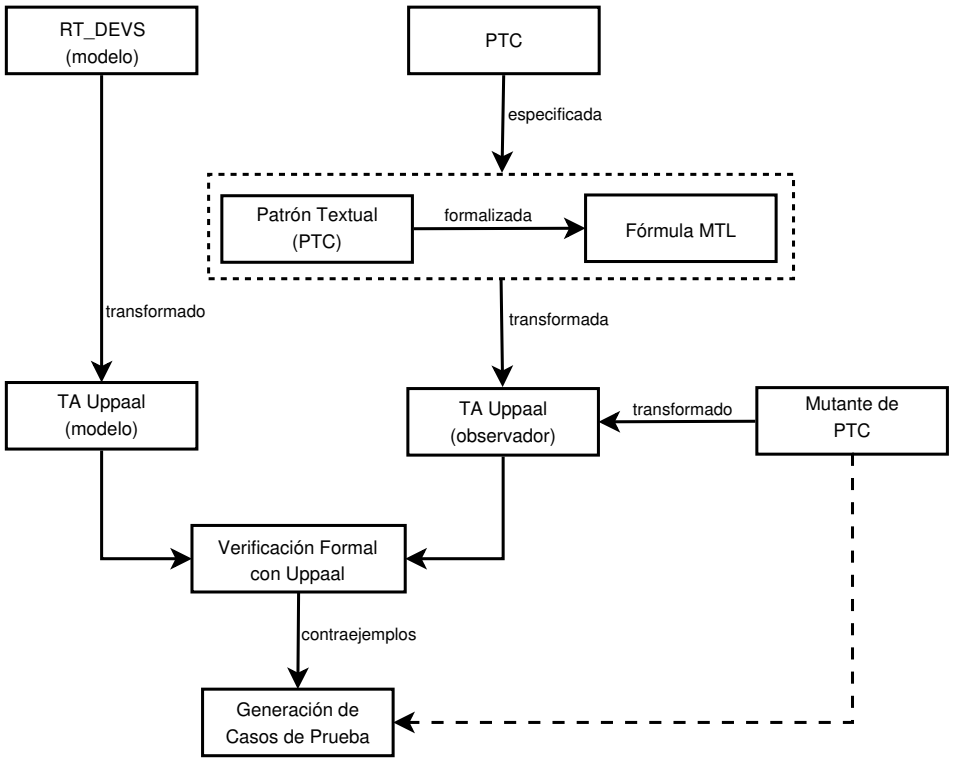


Fig. 1.1: Flujo de Actividades del Proceso de Verificación Formal

discontinua). De esta manera, el ingeniero puede comenzar con una descripción informal y familiar de la propiedad temporal y luego formalizarla con la ayuda de la documentación de patrones. Finalmente, los mutantes de PTC y los contraejemplos generados por Uppaal ayudan a los ingenieros a detectar errores temporales en sus modelos, los cuales también pueden utilizarse para generar casos de prueba para una implementación del modelo.

Por lo tanto, las contribuciones son las siguientes:

- Una técnica para la verificación de PTC expresadas mediante RT-DEVS y MTL utilizando el verificador de modelos Uppaal. No tenemos conocimiento de métodos que partan de modelos RT-DEVS con PTC y proporcionen información sobre cómo estas propiedades temporales pueden verificarse formalmente con un MC. En particular, nuestro enfoque hace uso de un *autómata observador* [35, 36] y de algunos elementos avanzados de Uppaal. Es decir, dado un modelo RT-DEVS (M) y una PTC que debe verificar (P), se describe un método que propone utilizar el MC Uppaal para verificar M respecto a P, basándose en la estrategia de los autómatas observadores.

- Cuando se trata de sistemas de tiempo real, los ingenieros se enfrentan a la desafiante tarea de verificar comportamientos complejos, incluyendo restricciones temporales difíciles de manipular. Por lo tanto, una segunda contribución de esta tesis es una técnica que utiliza mutantes de propiedades temporales y Uppaal para ayudar a los ingenieros a detectar errores temporales en modelos RT-DEVS. Esta técnica es complementaria a un proceso de verificación basado en simulación.

  Los mutantes de P (P') consisten en cambios semánticos que representan posibles interpretaciones erróneas de P hechas por un diseñador o desarrollador. Estos mutantes también se implementan utilizando los TA de Uppaal. Es decir, se verifica si M satisface P' y, en

caso afirmativo, se habrá descubierto el error.

- Una tercera contribución es la extensión de la técnica anterior para generar casos de prueba para una implementación[4] M de los modelos RT-DEVS, con el fin de descubrir si un error en el modelo fue propagado a la implementación.

Para ilustrar el trabajo propuesto, se analiza un caso de estudio que especifica el comportamiento de un cruce de trenes. En particular, el caso de estudio incluye el análisis de propiedades temporales cuantitativas ampliamente usadas [40] tales como *Time-Bounded Response*, *Precedence with Delay*, *Time-Restricted Precedence* y *Conditional Security*.

1.2 Resumen de Publicaciones

Esta tesis se basa y amplía los siguientes artículos publicados:

- [41] A. Gonzalez, C. Luna, M. Daniele, R. Cuello, M. Perez, Towards an automatic model transformation mechanism from UML state machines to DEVS models, CLEI Electron. J. 18 (2) (2015). doi:10.19153/cleiej.18.2.3.
- [42] A. Gonzalez, C. D. Luna, R. Abella. UML state machine as modeling language for DEVS formalism. In *XLII Latin American Computing Conference, CLEI 2016, Valparaíso, Chile, October 10-14, 2016, IEEE, 2016, pp. 1–12*. doi:10.1109/CLEI.2016.7833350.
- [43] A. Gonzalez, M. Cristiá, C. Luna. Mutants for metric temporal logic formulas. In *Proceedings of the XXII Iberoamerican Conference on Software Engineering, CIbSE 2019, La Habana, Cuba, April 22-26, 2019*. Curran Associates, pp. 349–362. https://api.semanticscholar.org/CorpusID:199465702
- [44] A. Gonzalez, M. Cristiá, C. Luna. Error finding in real-time systems using mutants of temporal properties. In *2021 40th International Conference of the Chilean Computer Science Society (SCCC)*. pp. 1–8. doi:10.1109/SCCC54552.2021.9650361.
- [45] A. Gonzalez, M. Cristiá, C. Luna. Verification of Quantitative Temporal Properties in RealTime-DEVS. Simulation: Transactions of the Society for Modeling and Simulation International (2025), pp. 1-20. doi:10.1177/00375497251340070.

En todos estos trabajos he participado como autor principal, apotando tanto en las contribuciones como en la redacción. El objetivo principal ha sido fortalecer los aspectos de modelado y verificación del formalismo DEVS, con el fin de proporcionar a la comunidad nuevas herramientas que amplíen el uso de los modelos. Considero al formalismo DEVS (y sus variantes) una herramienta valiosa para modelar y analizar sistemas de tiempo real. Aunque existen trabajos previos, aún queda mucho por aportar en cuanto al uso de técnicas de verificación formal de sus modelos. Actualmente, existen varias herramientas de simulación de modelos DEVS (PowerDevs [46], SuiteDevs [47], DEVSim++ [48], entre otras) que ayudan a comprender y analizar el comportamiento de los sistemas. Sin embargo, las simulaciones no garantizan que un modelo sea correcto respecto a alguna propiedad. Este es precisamente el objetivo principal de esta tesis.

Los artículos [41] y [42] son trabajos complementarios que contribuyen significativamente al fortalecimiento de las capacidades de modelado. Actualmente, no existe un lenguaje estándar ampliamente aceptado y utilizado por toda la comunidad de diseñadores DEVS, a diferencia del caso de UML establecido por el OMG [49]. No obstante, en los últimos años han surgido diversas

[4] Código fuente de un programa que implementa el modelo.

propuestas orientadas a facilitar la especificación. Algunas de estas propuestas utilizan una representación textual, como CML-DEVS [50], mientras que otras emplean una representación gráfica, como la presentada en [51].

En [41] se presenta un proceso de transformaciones que parte de modelos de máquinas de estado UML (Statecharts UML - SC-UML) y culmina en la generación de código C++ para la herramienta PowerDevs. Específicamente, se define un mapeo de modelos SC-UML a modelos DEVS, seguido de la transformación de estos últimos a código C++. Este proceso se enmarca dentro del paradigma de desarrollo guiado por modelos (Model-Driven Development - MDD) [52]. Posteriormente, [42] amplía y automatiza las transformaciones propuestas en [41], combinando las ventajas de ambos dominios: la simplicidad y amplia adopción del lenguaje SC-UML junto con el poder y versatilidad de las herramientas de simulación basadas en modelos DEVS.

Cabe destacar, que ambos trabajos han sido citados en recientes trabajos [53, 54, 55].

Por otro lado, en [43, 44, 45] las contribuciones se centran en aspectos relacionados con la verificación formal de modelos DEVS.

En [43] se propone un mecanismo para la detección de errores en sistemas de tiempo real modelados con DEVS clásicos. Las propiedades del sistema se especifican utilizando la lógica MTL, y se definen operadores de mutación en MTL con el objetivo de identificar posibles errores en los modelos DEVS. Sin embargo, las limitaciones relacionadas con la decidibilidad de MTL y la ausencia de *model checkers* que soporten esta lógica motivaron el desarrollo de un nuevo enfoque presentado en [44]. En este trabajo, independientemente del formalismo DEVS, se propone un mecanismo para la identificación de errores de temporización basado en PTC que debe cumplir un sistema, junto con mutantes de dichas propiedades. Las PTC representan situaciones recurrentes en sistemas de tiempo real, mientras que los mutantes reflejan posibles interpretaciones erróneas por parte de diseñadores o desarrolladores. Tanto las propiedades como sus mutaciones se modelan mediante TA siguiendo el enfoque de [25], y la verificación se basa en la estrategia de los autómatas observadores.

Finalmente, en [45], basado en la propuesta presentada en [44], se implementa un método para la verificación formal de PTC de modelos RT-DEVS utilizando el MC Uppaal. Además, se desarrolla en Uppaal un mecanismo para la detección de errores de temporización en los modelos RT-DEVS, así como la generación de casos de prueba para una implementación, empleando los mutantes de las PTC.

Esta tesis representa una extensión significativa del trabajo desarrollado en [45].

1.3 Organización

En el Capítulo 2 se presentan los trabajos preliminares que sirvieron de base para los aportes principales en esta tesis. El Capítulo 3 describe y analiza trabajos relacionados con la verificación formal de los modelos DEVS y sus variantes. En el Capítulo 4 se revisan conceptos fundamentales, incluyendo el formalismo RT-DEVS, los TA, la lógica MTL, los principios básicos del *model checking* y el testeo por mutación.

El Capítulo 5 aborda un caso de estudio sobre un cruce ferroviario (Train-Gate), utilizado para validar los resultados obtenidos en este trabajo. En el Capítulo 6 se introducen los patrones de PTC, sus mutantes, y se detalla el mecanismo de verificación con Uppaal. En el Capítulo 7 se define una estrategia de búsqueda de errores basada en mutaciones de PTC.

Posteriormente, en el Capítulo 8 se presenta la generación de casos de prueba para una implementación. Finalmente, las conclusiones y posibles líneas de trabajo futuro se exponen en el Capítulo 9.

Capítulo 2

Trabajos Previos

Las principales contribuciones de esta tesis surgen a partir de estudios y aportes previos orientados a optimizar el uso del formalismo DEVS. Los primeros trabajos se concentran en aspectos de modelado, mediante representaciones gráficas y su integración con otros lenguajes, mientras que los restantes se enfocan específicamente en la verificación formal de propiedades temporales en modelos DEVS y RT-DEVS.

2.1 Representación gráfica de modelos DEVS

El formalismo DEVS utiliza la teoría de conjuntos y funciones matemáticas para describir el comportamiento de los sistemas. La versión clásica es la definida por Zeigler en [56]. Esta versión permite que el conjunto de estados de un sistema sea infinito, lo que imposibilita la creación de una representación visual de estos modelos. No obstante, existen variantes que limitan este formalismo, permitiendo así definir una representación visual. En [41] y [42], utilizamos estas variantes y explicamos cómo emplear un lenguaje con representación gráfica ampliamente adoptado por la comunidad de ingenieros.

2.1.1 Artículo: *Towards an automatic model transformation mechanism from UML state machines to DEVS models*

En [41] se describe una representación gráfica de los modelos DEVS basada en las máquinas de estados de UML, dado que es ampliamente utilizada por la comunidad de diseñadores. En primer lugar, se define una representación XML (metamodelo) de los modelos atómicos del formalismo clásico DEVS (DEVS-XML), asumiendo ciertas restricciones, como considerar el conjunto de estados finito.

Luego, en el contexto de MDD, se plantean dos transformaciones. La primera es una transformación *model-to-model* que mapea modelos SC-UML hacia modelos DEVS-XML mediante reglas de transformación implementadas en el lenguaje QVT Relations (QVT-R) [57]. La segunda es una transformación *model-to-code* que convierte un modelo DEVS-XML a código C++ compatible con la herramienta de simulación PowerDevs [46], utilizando la librería Saxon XSLT y el procesador XQuery [58]. El proceso completo de estas transformaciones se muestra en la Figura 2.1.

De esta forma, la comunidad DEVS dispone de una alternativa de modelado con representación gráfica basada en UML. A su vez, la comunidad de diseñadores UML accede a una herramienta de simulación que les permite analizar el comportamiento de sus modelos, cubriendo así una necesidad importante.

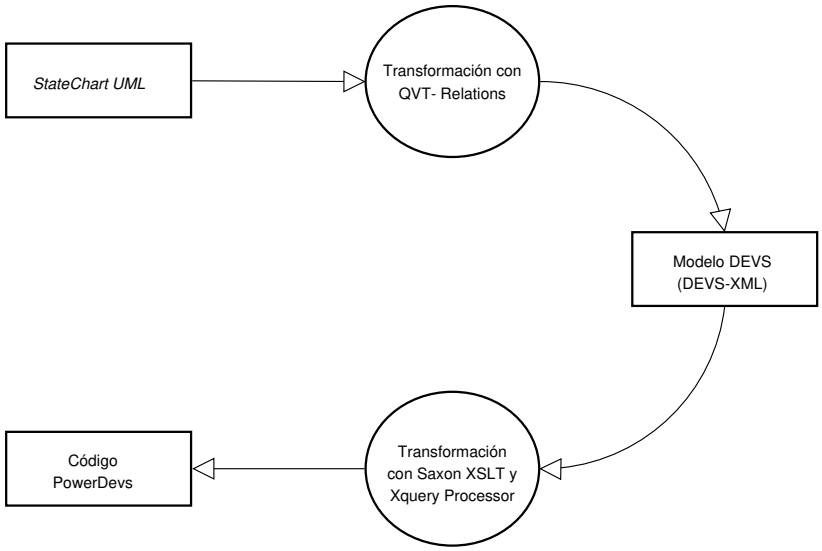


Fig. 2.1: Transformaciones de SC-UML a DEVS-XML y de DEVS-XML a PowerDEVS

La principal ventaja de definir una secuencia de dos transformaciones, en lugar de realizar directamente la conversión de SC-UML a código C++, radica en que la traducción intermedia a DEVS-XML permite ampliar el proceso en el futuro. Por un lado, esta intermediación posibilita la generación de código compatible con otras herramientas de simulación, como Devs-Suite [47]. Por otro lado, facilita el uso de otras representaciones gráficas, que solo necesitarían traducirse a DEVS-XML, sin preocuparse por generar código específico para una herramienta en particular, aprovechando así los aportes de este trabajo.

2.1.2 Artículo: *UML State Machine as Modeling Language for DEVS Formalism*

En la misma línea, [42] amplía la propuesta anterior. En este caso, se desarrolla un *plugin* para el entorno de desarrollo Eclipse [59] que implementa una secuencia de actividades que abarcan desde SC-UML con estados compuestos hasta la generación de código C++ para PowerDevs.

Las contribuciones de este trabajo amplían las de [41] y se centran en los siguientes aspectos:

- La definición de una transformación para mapear máquinas de estados compuestos[1] de UML a máquinas de estados simples[2] de UML, un proceso denominado "aplanado", mediante reglas de transformación en QVT-R. Este enfoque permite modelar sistemas con diferentes niveles de abstracción.
- La creación de una transformación de modelo a código (*Meta-Object Facility to Text* - M2T), utilizando Acceleo [60], que convierte modelos DEVS-XML (en formato Ecore) en código C++ ejecutable con el motor de simulación PowerDevs.
- La implementación de un *plugin* para Eclipse que automatiza el proceso de transformación, el cual incluye:
  1. Un editor de máquinas de estados UML *ad hoc* basado en el proyecto Papyrus [61].
  2. Una implementación en Java para ejecutar la transformación del "aplanado".

[1] Estados que pueden descomponerse en varios subestados
[2] Estados que no se descomponen en subestados

3. Una implementación en Java para transformar máquinas de estados simples en modelos DEVS.

4. Una implementación en Java para ejecutar la transformación M2T y generar código C++ a partir de modelos DEVS.

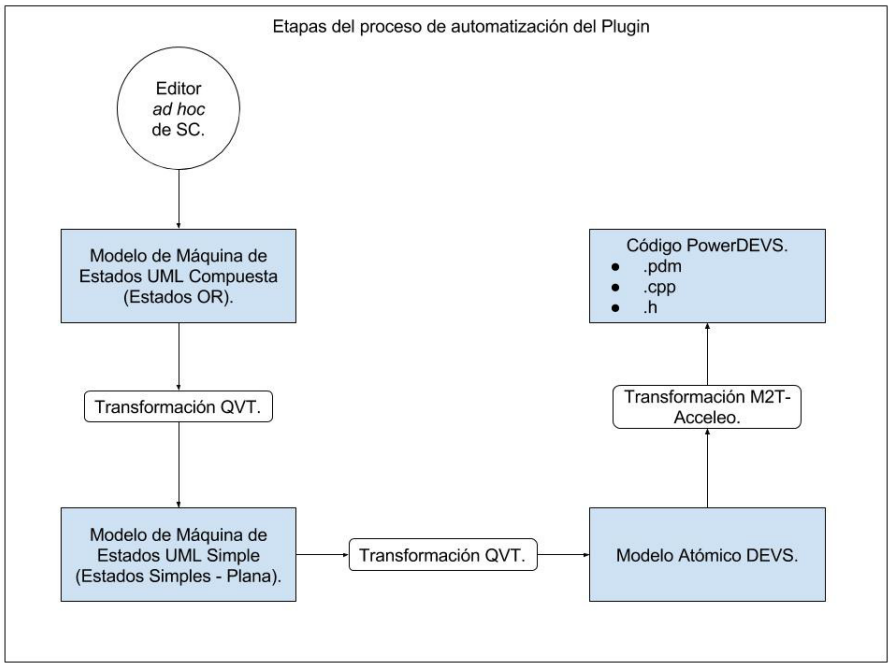


Fig. 2.2: De DEVS-XML a PowerDevs: proceso de automatización.

De esta manera, [42] representa un avance significativo en la construcción de un sistema que permita automatizar por completo el proceso de análisis, diseño y ejecución de máquinas de estados UML mediante un motor de simulación de modelos DEVS, tal como se ilustra en la Figura 2.2. Por otro lado, como se mencionó anteriormente, la comunidad DEVS puede utilizar los SC-UML como una alternativa para representar gráficamente los modelos DEVS con estados finitos.

2.2 Verificación formal de modelos DEVS

La verificación formal de modelos DEVS ha sido una necesidad no resuelta hasta ahora. Algunas aproximaciones han intentado verificar variantes del formalismo, aunque sin obtener resultados significativos. En este contexto, hemos trabajado en los últimos años para lograr resultados prácticos, novedosos y convincentes. Los trabajos que se presentan a continuación reflejan la evolución de estas investigaciones, que no solo se enfocan en la verificación de modelos DEVS, sino también en el desarrollo de una metodología para la búsqueda de errores en los modelos y la generación de casos de prueba. Las contribuciones iniciales fueron publicadas en [43], y los aportes presentados en [44] surgen de extensiones y mejoras del trabajo previo. Finalmente, en [45], se amplían, refinan e implementan las metodologías de verificación, búsqueda de errores y generación de casos de prueba desarrolladas en los trabajos anteriores.

2.2.1 Artículo: *Mutants for Metric Temporal Logic Formulas*

Los primeros resultados fueron publicados en [43], donde se propone una metodología para verificar, con un MC, PTC en sistemas de tiempo real modelados con el formalismo clásico de DEVS. En primer lugar, se selecciona un conjunto de PTC recurrentes en los sistemas (patrones), que se especifican mediante MTL. Las especificaciones en MTL permiten no solo definir el orden en que ciertos estados deben aparecer en un sistema, sino también determinar cuándo deben ocurrir, lo que convierte a esta lógica en una herramienta adecuada para especificar PTC.

Las propiedades de precedencia son comunes en las especificaciones de sistemas reactivos y se refieren a situaciones en las que un predicado activa la ocurrencia de otro.

Ejemplo 2.1. Si consideramos un semáforo clásico que alterna entre tres estados, pasando de verde a amarillo, de amarillo a rojo y de rojo a verde, una propiedad del sistema sería que el color amarillo debe seguir inmediatamente después de que el semáforo haya permanecido 20 segundos en verde.

En MTL, esta propiedad se puede especificar como: $\Box(verde \rightarrow (verde \cup_{[20,20]} amarillo))$. Esta fórmula se puede leer como: *siempre* ($\Box$), cuando el semáforo se pone en *verde*, *entonces* ($\rightarrow$) se mantiene así *hasta* ($\cup$) que transcurren 20 segundos, momento en el cual cambia a *amarillo*.

De manera general, la propiedad puede expresarse como: $\Box(P \rightarrow (P \cup_{[x,y]} Q))$, donde P y Q son predicados del problema. □

También, proponemos mutantes de estos patrones mediante cambios sintácticos en las fórmulas MTL, los cuales representan posibles errores en los modelos DEVS. Algunos cambios se realizan sobre los intervalos de los operadores de MTL, mientras que otros afectan a los operadores modales ($\Box$, $\Diamond$ y $\cup$). Un mutante del Ejemplo 2.1 podría ser: $\Box(verde \rightarrow (verde \cup_{(k,20]} amarillo))$, con $0 < k < 20$. Este cambio representa un comportamiento del semáforo que permite un cambio de color antes de lo previsto.

Finalmente, se describe cómo podríamos utilizar un MC para generar trazas que permitan testear una implementación a partir de los patrones y sus mutantes, aunque no se presenta un ejemplo concreto en este punto. Los aportes más importantes en esta etapa son:

- La selección de un conjunto de propiedades recurrentes en los sistemas de tiempo real.
- La especificación de estas propiedades mediante fórmulas MTL.
- La definición de mutantes de estas fórmulas, basados en cambios sintácticos.
- Una propuesta para la generación de trazas utilizando un MC.

La propuesta de este trabajo se ejemplifica con un sistema de alarmas de un automóvil.

2.2.2 Artículo: *Buscando Errores en Sistemas de Tiempo Real con Uppaal*

En [44] presentamos avances que nos acercan a una posible verificación formal de PTC mediante un MC. En este trabajo, por un lado, ampliamos tanto el número de patrones como el de mutantes, y por otro, modelamos estos elementos mediante TA cercanos a los definidos por Alur y Dill en [25]. De este modo, dejamos abierta la posibilidad de utilizar un MC que trabaje con alguna variante de estos TA, como Kronos [62], Uppaal [8], HYTECH [63], entre otros.

Estos TA actúan como TA observadores, es decir, se ejecutan en paralelo con el modelo y transicionan según lo haga el modelo del sistema. Posteriormente, podemos verificar si algún estado del TA observador, que representa la violación de una propiedad, es alcanzado.

Ejemplo 2.2. Las propiedades de precedencia hacen referencia a situaciones en las que un predicado P permite que otro predicado Q se satisfaga dentro de un cierto intervalo de tiempo. El patrón *Time-Restricted Precedence*, que pertenece a esta categoría de propiedades, es representado por el TA de la Figura 2.3. Los círculos representan los estados del autómata, mientras que las flechas que los conectan representan las transiciones; el círculo gris indica el estado inicial; P y Q expresiones booleanas que representan los predicados del patrón y forman parte de la guarda (condición) de una transición; x:=0 significa que el reloj x se reinicia a cero; $x > k$ es una expresión que forma parte de la guarda de la transición; y Urgent indica que una transición activa debe dispararse inmediatamente. El estado Error es utilizado para determinar si la propiedad se satisface o no, esto es explicado en detalle en el Capítulo 6. Otro elemento sintáctico que se puede observar en las Figuras 2.5(f) y (g), es el estado marcado con una letra C, el cual indica que se trata de un estado *commit*, y permite representar una secuencia atómica de cambios de estado. □

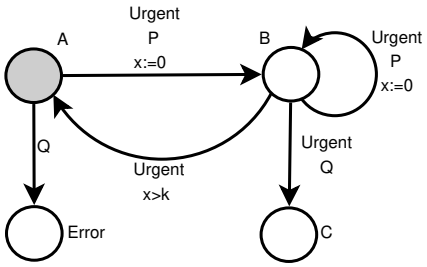


Fig. 2.3: TA del patrón *Time-Restricted Precedence*.

Las mutaciones propuestas en este trabajo no se centran en cambios sintácticos de las fórmulas, sino en capturar posibles interpretaciones erróneas del diseñador. Por ejemplo, el diseñador podría interpretar que una nueva ocurrencia de P no extiende el tiempo permitido para Q; en este caso, esta interpretación es capturada por el TA mutante de la Figura 2.4, el cual se obtiene eliminando la transición cíclica del estado B en el TA de la Figura 2.3.

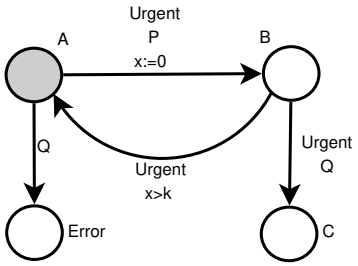


Fig. 2.4: TA mutante del patrón *Time-Restricted Precedence*.

En este trabajo se utiliza una versión cercana a las definidas por Alur; sin embargo, estas pueden adaptarse a versiones compatibles con algún MC. Por último, se describe un análisis de casos para la generación de entradas para una implementación que refina y extiende la propuesta presentada en [43].

Los principales aportes de este trabajo son:

- Aumentar el número de patrones y mutantes para ampliar la cobertura de situaciones recurrentes en sistemas de tiempo real. Para ello, nos basamos en los patrones definidos

en [40], descartando aquellos que no pueden ser verificados con un MC. Este análisis se explica en el Capítulo 6.

Aquí, los patrones se presentan en un formato textual que facilita su interpretación por parte de ingenieros no familiarizados con las lógicas temporales. Para más detalles sobre su formalización, se puede consultar la Sección 6.2. A continuación, se enumeran los patrones considerados en este trabajo:

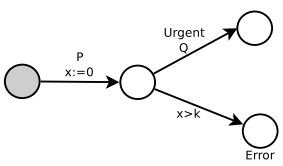
- *Time-Bounded Response.* Patrón: *P must be followed by Q within k time-units.* Q ocurre en respuesta a P, en un plazo de a lo sumo k unidades de tiempo.
- *Time-Bounded Existence.* Patrón: *Starting from the current point of time, P must occur within k time-units.* Una propiedad P debe ocurrir en a lo sumo k unidades de tiempo.
- *Precedence with Delay.* Patrón: *Q enables P after a delay k.* P es activada solo si Q ocurrió al menos k unidades de tiempo antes.
- *Time-Restricted Precedence.* Patrón: *P enables Q for k time-units.* Q se activa solo si P ocurrió no más de k unidades de tiempo antes; es decir, P activa a Q durante un lapso de tiempo de k unidades.
- *Time-Restricted Disable.* Patrón: *Q disables P within k time units.* Si ocurre P, esta se mantiene válida hasta que, en algún momento dentro de un lapso de tiempo k, ocurre Q.
- *Security/Absence.* Patrón: *Starting from the current point of time, P does not occur for k time units.* P nunca ocurre durante un lapso de tiempo k.
- *Conditional Security.* Patrón: *If Q then P occurs for k time units.* P debe cumplirse y mantenerse así durante un lapso de tiempo k a partir del momento en que ocurre Q.
- *Time-Bounded Frequency.* Patrón: *P occurs frequently before k time-units.* P siempre vuelve a ocurrir antes que pase un determinado tiempo.
- *Time-Bounded Frequency with time-out.* Patrón: *P occurs frequently before k time units or occurs Q.* Si P no vuelve a ocurrir en un lapso de tiempo k, entonces Q ocurre tras k unidades de tiempo; es decir, Q desactiva la frecuencia de P.

- Definir una representación de estos patrones y mutantes mediante TA. La Figura 2.5 presenta los TA de los patrones considerados en este trabajo.
- Extender y experimentar con la metodología de generación de casos de prueba. En [43] se describe brevemente un marco teórico sobre cómo se podrían generar casos de prueba utilizando los mutantes de las PTC, y se plantea como trabajo futuro. Sin embargo, en este trabajo se realizó una experimentación parcial con la generación de un caso de prueba concreto ejemplificado con un sistema de alarmas de automóviles. Para este caso, utilizamos el patrón *Time-Bounded Response* y un mutante del mismo.

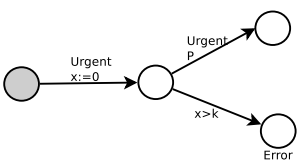
Todas las contribuciones se ejemplificaron con una versión extendida del sistema de alarmas de un automóvil descrito en [43].

2.2.3 Artículo: *Verification of Quantitative Temporal Properties in Real-Time-DEVS*

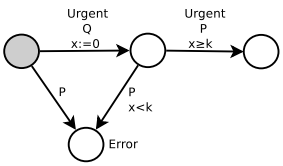
Esta tesis se basa y focaliza esencialmente en [45], trabajo en el que se describen las contribuciones más relevantes, y se extienden, implementan y refinan los aportes realizados en [43, 44]. En este trabajo mostramos cómo utilizar y aprovechar las capacidades del MC Uppaal para verificar,



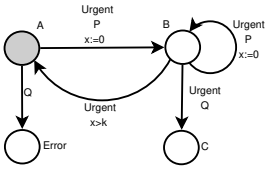
(a) TA *Time-Bounded Response.*



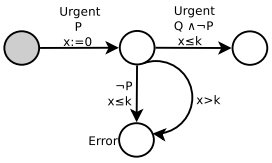
(b) TA *Time-Bounded Existence.*



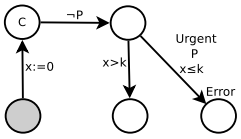
(c) TA *Precedence with Delay.*



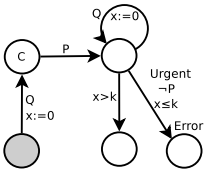
(d) TA *Time-Restricted Precedence.*



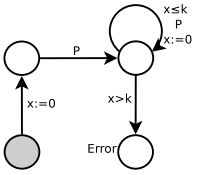
(e) TA *Time-Restricted Disable.*



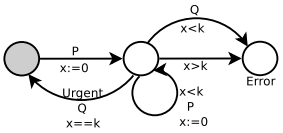
(f) TA *Ausence.*



(g) TA *Conditional Security.*



(h) TA *Time-Bounded Frequency.*



(i) TA *Time-Bounded Frequency with time-out.*

Fig. 2.5: TA de propiedades temporales recurrentes (patrones).

detectar errores y generar casos de prueba. Adaptamos los TA de [44] a una versión compatible con esta herramienta y utilizamos una variante del formalismo DEVS, RT-DEVS, ya que observamos que se adapta mejor para modelar sistemas de tiempo real.

Los principales aportes de este trabajo son:

- Adaptación de los patrones y mutantes a TA admitidos por Uppaal.
- Definición de una metodología de verificación y detección de errores con Uppaal.
- Definición de dos estrategias de generación de casos de prueba para una implementación.

Las contribuciones son validadas sobre un sistema de control ferroviario. No describimos acá más detalles de este trabajo, ya que se abordan detalladamente en los Capítulos 4-8 de esta tesis.

Capítulo 3

Trabajos Relacionados

Existen diversos estudios que presentan lenguajes y métodos para modelar sistemas de tiempo real y verificar propiedades de interés. Algunos se enfocan exclusivamente en la modelización, otros en técnicas para la verificación formal o por simulación de modelos DEVS, y algunos abarcan ambos aspectos.

En [64], los autores proponen un lenguaje de modelado con representación gráfica, denominado *High Level Language for Systems Specification* (HiLLS), para la simulación y verificación formal de modelos. Definen dos transformaciones de modelos utilizando el lenguaje ATL [65]: una que convierte modelos HiLLS en modelos DEVS clásicos, permitiendo su simulación con herramientas existentes, y otra que transforma modelos HiLLS en TA para la verificación de propiedades mediante Uppaal. Además, muestran cómo especificar los patrones temporales de Dwyer con HiLLS [66], para luego traducirlos a *queries* de Uppaal. Como trabajo futuro, plantean la automatización de algunas etapas de la transformación y la integración de herramientas de distintos dominios. Cabe destacar que los patrones definidos por Dwyer describen propiedades temporales no cuantitativas.

Otra transformación hacia modelos DEVS se presenta en [67], donde se propone una traducción de procesos de negocio modelados con BPMN, incluyendo notaciones que representan restricciones temporales, hacia modelos DEVS. El objetivo es detectar violaciones de estas restricciones mediante simulaciones. Aunque no realizan traducciones para la verificación formal, podrían aprovechar los aportes de esta tesis para ese propósito.

[68] muestra cómo el uso de formalismos combinados permite modelar, verificar y validar software complejo, como los videojuegos. En este trabajo, se extiende el lenguaje PROMELA [9] (DEv-PROMELA) para representar también modelos DEVS. Luego, se describe un framework de verificación y simulación transparente para el usuario, basado en dos transformaciones: (1) convertir el modelo DEv-PROMELA en un modelo PROMELA para su verificación con SPIN [9], aunque sin considerar aspectos temporales; y (2) transformar el mismo modelo DEv-PROMELA en un modelo de simulación DEVS, permitiendo su ejecución mediante un algoritmo existente. Sin embargo, esta estrategia impide el uso del model checker SPIN para la verificación formal de propiedades temporales.

En [69], Zeigler, Nutaro y Seo analizan y describen diversos métodos que combinan las ventajas del formalismo DEVS con *model checking*. Principalmente, presentan traducciones entre formalismos, algunas de las cuales ya han sido mencionadas previamente, como la conversión de modelos DEVS a TA para su verificación con Uppaal. Además, describen cómo traducir modelos de Finite Deterministic DEVS [70] a PROMELA para su verificación con SPIN y, como principal aporte, cómo analizar la variante *Finite Probabilistic DEVS* (FP-DEVS) utilizando Me MS4 [71]. En este trabajo no consideran verificar formalmente PTC.

En [19], los autores proponen un mecanismo para verificar algunas propiedades de seguridad

(*safety*) sin utilizar un MC. Usan la variante RT-DEVS y validan la propuesta mediante un ejemplo del dominio ferroviario. El trabajo define la noción de matriz de relojes para la comunicación entre los modelos RT-DEVS. Luego, basado en la matriz de relojes, definen un algoritmo que construye un árbol de alcanzabilidad temporizado el cual es usado para analizar propiedades de seguridad. Las propiedades que se verifican son formuladas con una lógica temporal similar a LTL, la cual no admite la especificación literal del tiempo.

En [20], los autores describen una traducción de RT-DEVS a los TA de Uppaal. Se basan en una traducción informal y mencionan que puede ser automatizada dada la cercanía entre ambos formalismos. Realizan simulaciones de los modelos RT-DEVS usando la herramienta de simulación ActorDEVS y verifican algunas propiedades temporales con Uppaal para validar los modelos.

En [37] se presenta una traducción de modelos DEVS clásicos a TA con el fin de describir las propiedades de alto nivel con los TA y las propiedades de bajo nivel con modelos DEVS. El método se ejemplifica con un sistema de llenado de barril (*barrel filling*) en una planta de producción. Aquí, definen una traducción del formalismo DEVS clásico a la teoría de TA mediante la técnica de *relación de simulación*. Más precisamente, definen una relación de simulación (*forward simulation relation*) que vincula el comportamiento del DEVS con el del TA. Sin embargo, no verifican ninguna propiedad temporal sobre el TA, simplemente aluden a que los resultados y herramientas sobre los TA pueden ser aplicados en ese contexto específico.

Otros trabajos que traducen una variante del formalismo DEVS (Rational-TimeAdvance DEVS - RTA DEVS) a TA son [38] y [21]. La variante RTA-DEVS se caracteriza por aceptar en la función de avance del tiempo solo constantes racionales. En ambos trabajos se muestra cómo traducir estos RTA-DEVS a TA para luego verificar propiedades con Uppaal. En particular, en [21] se muestra un caso de estudio de un controlador de un robot tipo E-Puck, y se verifica si hay *deadlock* y otras propiedades en base a posibles entradas que son modeladas con autómatas.

En [72], los autores definen una variante propia del formalismo DEVS que considera estados, entradas y salidas finitas, y además, fuerzan a que el número de transiciones internas y externas sea finito dentro de algún período definido de tiempo. En ese trabajo se implementa un algoritmo para verificar los modelos DEVS admitidos. Para expresar las propiedades no usan ninguna lógica temporal, estas son expresadas en el mismo lenguaje de programación (JAVA). Aducen que de esta forma podrían expresar propiedades complejas acerca de las variables de estado, incluso medidas de desempeño (tiempos de espera máximos, promedios, etc.). Ejemplifican la propuesta con una arquitectura de interconexión utilizada en sistemas en chip (*Network-on-Chip*), diseñada para facilitar la comunicación eficiente entre múltiples núcleos de procesamiento, memorias y otros componentes dentro de un único chip. No obstante, únicamente abordan la verificación de ciertos requisitos no cuantitativos que el sistema debe cumplir, los cuales se definen en Java.

Otro trabajo que verifica modelos DEVS con Uppaal es [73]. Aquí, se define una bisimulación entre DEVS clásicos y TA. Por un lado, experimenta con un modelo de la plataforma Apache Storm mediante simulaciones para descubrir ciertos comportamientos, y por otro, define una transformación para verificar formalmente propiedades temporales que estos sistemas deben cumplir. Transforma la especificación del DEVS Apache Storm a su correspondiente TA mediante la técnica de bisimulación y verifica propiedades que pueden ser expresadas por la lógica de Uppaal. Por lo tanto, con el método propuesto por estos autores no se puede verificar propiedades temporales cuantitativas tal como lo hacemos en nuestro trabajo. Es importante aclarar que dicho trabajo solo muestra cómo traducir los componentes del DEVS Apache Storm pero no desde un DEVS genérico.

Los trabajos analizados previamente muestran cómo verificar modelos DEVS. Algunos de ellos lo realizan a través de una traducción de los modelos DEVS a TA de Uppaal. Por lo tanto, estos métodos se encuentran limitados a expresar propiedades con la lógica que acepta Uppaal,

la cual no admite la especificación de requisitos temporales cuantitativos. Por ejemplo, con estos métodos no es posible expresar propiedades como *P enables Q for k time-units*. Si bien existen artículos (por ejemplo [38, 21]) que muestran cómo se podría verificar la propiedad de respuesta con límite de tiempo (*P must be followed by Q within k time-units*) añadiendo una variable booleana y un reloj, quedan limitados a unas pocas propiedades.

En nuestro trabajo, no solo demostramos cómo verificar este tipo de propiedades con Uppaal, sino que también presentamos una estrategia sistemática para la detección de errores mediante mutantes de las propiedades temporales. De este modo, la comunidad de diseñadores DEVS y sus variantes dispone de una herramienta de verificación que permite comprobar si sus modelos cumplen con PTC recurrentes en los sistemas de tiempo real.

Capítulo 4

Nociones Preliminares

En esta sección recordamos brevemente los formalismos que utilizamos en el resto del trabajo: el formalismo RT-DEVS, la teoría de TA, conceptos básicos de *model checking*, la Lógica Temporal Métrica (MTL) y testeo por mutación. Estos temas constituyen la base teórica para la metodología propuesta en este artículo. El lector que esté familiarizado con alguno de ellos puede saltear la sección correspondiente.

4.1 Formalismo Real-Time DEVS (RT-DEVS)

El vertiginoso avance tecnológico ha dado lugar a la aparición de sistemas dinámicos cada vez más complejos. Ejemplos de estos incluyen las cajas de cambio automáticas en vehículos, los sistemas de control ferroviario y la robotización en plantas de producción, entre otros. Las actividades en estos sistemas están vinculadas a la ocurrencia asincrónica de eventos discretos, algunos provocados por el entorno (como la presión sobre el freno de un vehículo) y otros originados por el propio estado interno del sistema (como la falla espontánea de un dispositivo). Esta característica da lugar a la definición del término *Sistemas de Eventos Discretos* (Discrete Event System - DES).

En la teoría de DES [74] no solo se encuentran herramientas específicas para abordar problemas de modelización, simulación y análisis de sistemas, sino también un campo fértil para el desarrollo de nuevas técnicas y teorías, debido a la gran cantidad de problemas aún abiertos en el área, entre ellos, la aplicación de técnicas de verificación formal. Entre los formalismos más utilizados para representar DES se encuentran las Redes de Petri [75], Statecharts [76], Grafos de Eventos [77], así como diversas generalizaciones y especializaciones de estos.

Orientado a los problemas de modelado y simulación de sistemas discretos, en la década del 70', Bernard Zeigler propuso un marco teórico y una metodología para la modelización y simulación de sistemas, dando origen al formalismo DEVS [78]. DEVS es un formalismo de modelado con una sólida semántica, basado en la teoría de sistemas, que permite describir sistemas dinámicos orientados a eventos discretos con restricciones de tiempo.

En la actualidad, existe un conjunto de variantes de DEVS que permiten modelar problemas en dominios específicos, entre ellas: FD-DEVS [70], Cell-DEVS [79], RTA-DEVS [38], FP-DEVS [69], entre otras.

En este trabajo, empleamos la variante RT-DEVS [26, 19], que se adapta mejor a la modelización de sistemas de tiempo real. La principal diferencia con los DEVS clásicos radica en que permite especificar la ocurrencia de un evento interno dentro de un intervalo de tiempo, en lugar de un instante preciso, mediante un tiempo de ejecución asociado a una *actividad* de estado.

Definición 4.1.1. *Un modelo atómico RT-DEVS se define como la tupla*

$$\langle X, Y, S, ta, \delta_{ext}, \delta_{int}, \lambda, A, \omega, ti \rangle$$

, donde:

(i) X *es el conjunto de eventos de entrada;*

(ii) Y *el conjunto de eventos de salida;*

(iii) S *el conjunto de estados;*

(iv) $ta : S \to \mathbb{R}^{+}_{0,\infty}$ *la función de avance del tiempo;*

(v) $\delta_{ext} : Q \times X \to S$ *la función de transición externa, con* $Q = \{(s, e) \,|s \in S \wedge 0 \leq e \leq ta\,(s)\}$ *el conjunto de estados total, e el tiempo transcurrido desde la última transición y* $ta(s)$ *el tiempo de permanencia en el estado;*

(vi) $\delta_{int} : S \to S$ *es la función de transición interna;*

(vii) $\lambda : S \to Y$ *la función de salida;*

(viii) $A = \{a \mid t(a) \in \mathbb{R}^{+}_{0,\infty} \wedge a \notin \{X?, Y!, S =\}\}$*, es el conjunto de actividades, donde* $t(a)$ *es el tiempo de ejecución de una actividad* a*,* $X?$ *es la acción de recibir datos desde* X*,* $Y!$ *la acción de enviar datos a través de* Y*, y* $S =$ *es la acción de modificar un estado en* S*;*

(ix) $\omega : S \to A$ *la función que mapea un estado en una actividad;*

(x) $ti : S \to \mathbb{R}^{+}_{0,\infty} \times \mathbb{R}^{+}_{0,\infty}$ *la función que define el intervalo de tiempo de un estado tal que* $ti(s) = [li, ls]$*, siempre y cuando* $li \leq ta(s) \leq lb$*,* $li \leq t(a) \leq lb$ *con* $a = \omega(s)$*.*

Los estados representan actividades concretas definidas por la función w. La trasmisión de un mensaje, el procesamiento de una tarea, etc., son ejemplos de una actividad. Para modelar el tiempo de ejecución de una actividad, $t(a)$, los estados son vinculados a un intervalo de tiempo de números reales, dado que podría no ser un valor exacto, por consiguiente, ahora en una transición interna el estado destino podría determinarse según el tiempo transcurrido desde el último cambio de estado. Para el propósito de análisis de sistemas con requisitos temporales usaremos una forma reducida de RT-DEVS asumiendo que $ta(s) = t(a)$, con $a = \omega(s)$, y también ignorando términos dependientes de la representación de una actividad tal como la función ω. De esta forma el modelo atómico RT-DEVS queda definido con la siguiente tupla: $\langle X, Y, S, \delta_{int}, \delta_{ext}, \lambda, ti \rangle$. Dada esta definición, es importante notar que un RT-DEVS es lo mismo que un DEVS excepto que la función de avance del tiempo es reemplazada por la función de intervalo de tiempo.

La semántica es similar a los DEVS clásicos excepto para el tiempo de permanencia en un estado. Un sistema que está en un estado $s \in S$, permanecerá en dicho estado por un tiempo $t \in ti(s) = [li, ls]$, dando lugar a una *transición interna*, excepto que, habiendo transcurrido un tiempo $e \leq ls$ ocurra un evento de entrada con valor $x \in X$. En tal caso, el sistema experimentará una *transición externa* y cambia al estado $s' = \delta_{ext}\,(s, e, x)$. Cuando una *transición interna* se dispara primero producirá un evento de salida con valor $y = \lambda\,(s)$ y luego un cambio a un nuevo estado $s' = \delta_{int}\,(s)$.

Como en los modelos DEVS, los modelos RT-DEVS pueden acoplarse mediante puertos para construir modelos complejos. Dado un modelo acoplado RT-DEVS, es posible obtener un modelo atómico RT-DEVS equivalente, como en DEVS.

En este caso, X (inputs) y Y (outputs) tienen la siguiente estructura:

(*i*) $X = \{(p, v) \mid p \in \mathit{InPorts} \wedge v \in X_p\}$ donde *InPorts* es el conjunto de puertos de entrada y X_p es el conjunto de valores de entrada que pueden recibirse en el puerto p;

(*ii*) $Y = \{(p, v) \mid p \in \mathit{OutPorts} \wedge v \in Y_p\}$ donde *OutPorts* es el conjunto de puertos de salida y Y_p es el conjunto de valores que pueden enviarse a través del puerto p.

Cuando los modelos RT-DEVS se acoplan, los puertos de entrada de un componente se conectan a los puertos de salida de otros componentes y viceversa [80].

4.2 Autómata Temporizado

Los TA constituyen una teoría para la modelización y verificación de sistemas de tiempo real. Siguiendo el trabajo de Alur y Dill [25], se han desarrollado diversos *model checkers* cuyos lenguajes de entrada son variantes de estos TA, como Kronos [62], Uppaal [8] y HYTECH [63].

Un TA es, en esencia, un autómata finito extendido con variables denominadas relojes, las cuales pueden tomar valores reales. Estas variables representan los relojes lógicos del sistema. Al iniciar la ejecución, se inicializan en cero y, posteriormente, se incrementan de manera sincronizada a una misma velocidad.

Formalmente un TA es una tupla [25, 81]: $A = \langle N, l_0, E, I\rangle$ donde N es un conjunto finito de nodos, $l_0 \in N$ es el nodo inicial, $E \subseteq N \times \beta(C) \times \Sigma \times 2^c \times N$ es el conjunto de transiciones, y $I : N \to \beta(C)$ asigna invariantes a los nodos. C es un conjunto de variables de reloj, Σ es un conjunto de acciones y 2^c es el conjunto de relojes a ser reseteados a cero. En las transiciones y los nodos se pueden usar restricciones sobre los relojes, donde $\beta(C)$ denota el conjunto de todas ellas. Cuando se usan en las transiciones son llamadas *guardas*, mientras que en los nodos son llamadas *invariantes*. Una restricción de reloj es una conjunción de fórmulas atómicas de la forma $x \sim n$ o $x - y \sim n$ con $x, y \in C$, $\sim \in \{<, >, =, \leq, \geq\}$ y $n \in \mathbb{N}$, donde $\mathbb{N}$ es el conjunto de los números naturales.

Las guardas son restricciones que deben satisfacerse para que una transición se pueda disparar. Los invariantes restringen el tiempo que el TA puede permanecer en un nodo. Una expresión de la forma $l \xrightarrow{g,a,r} l'$ con $(l, g, a, r, l') \in E$ es una transición desde el nodo l al nodo l' donde g es una restricción de reloj, a es una acción y r es un conjunto de relojes a ser reseteados a cero.

Semántica Operacional

El estado de un TA es un par de la forma (L, u), donde L es el nodo corriente y $u : C \to \mathbb{N}$ (llamada valuación) es una función que determina el valor de cada reloj en L. Un TA puede producir un retardo por algún tiempo (*transición de retardo*) o seguir por una transición activa (*transición de acción*). En una *transición de retardo* de la forma $(L, u) \xrightarrow{d} (L, u + d)$ el avance del tiempo d dispara una transición en donde el nodo origen y destino es el mismo y $u + d = \{(c, y) \mid c \in C \wedge y = u(c) + d\}$ representa el avance del tiempo mediante una actualización de relojes. En una *transición de acción* $(L, u) \xrightarrow{a} (L', u')$ una acción a dispara una transición $L \xrightarrow{g,a,r} L'$ tal que la valuación u satisface la guarda g (lo que se abrevia con $u \in g$), se resetean a cero los relojes de r ($u' = \{(c, y) \mid c \in C \wedge$ **if** $c \in r$ **then** $y = 0$ **else** $y = u(c)\}$), y u' satisface el invariante de L' ($u' \in I(L')$).

4.3 Lógica Temporal Métrica (Metric Temporal Logic - MTL)

MTL es una extensión de LTL, una lógica ampliamente utilizada para la especificación y verificación de sistemas concurrentes y reactivos. La mayoría de los enfoques basados en LTL

adoptan un modelo de tiempo discreto, en el cual la ejecución de un sistema se representa como una secuencia de observaciones. Sin embargo, este enfoque resulta inadecuado para sistemas de tiempo real, donde la ejecución se modela ya sea como una secuencia de eventos con marcas de tiempo en números reales o como una trayectoria con dominio en los números reales no negativos. A pesar de esta limitación, LTL permite la especificación de un amplio espectro de propiedades temporales, incluidas aquellas de *security* (garantizando que nada malo suceda en el futuro) y de *liveness* (asegurando que algo bueno eventualmente ocurrirá). Por ejemplo, la propiedad "si el sensor detecta algún movimiento, entonces el sistema de alerta debe activarse" se expresa en LTL mediante la fórmula $\Box(detecta\_movim \rightarrow \Diamond activar\_alerta)$, donde el operador modal $\Box$ significa "siempre" y el operador modal $\Diamond$ indica "eventualmente en el futuro" ($\rightarrow$ denota "implica").

Sin embargo, LTL no puede expresar que el sistema de alerta debe activarse dentro de un intervalo de tiempo específico, por ejemplo, en un máximo de 10 segundos tras la detección de movimiento. Para abordar esta limitación, MTL extiende LTL mediante la incorporación de restricciones temporales en los operadores modales, lo que permite la especificación de PTC [32, 82]. De este modo, el requisito anterior puede modelarse mediante la siguiente fórmula MTL: $\Box(detecta\_movim \rightarrow \Diamond_{[0,10]} activar\_alerta)$, lo que significa que siempre que *detecta_movim* sea verdadero, entonces *activar_alerta* será verdadero en un máximo de 10 segundos después de que *detecta_movim* se haya vuelto verdadero.

Un ejemplo ligeramente más complejo es el siguiente: "si el sensor detecta algún movimiento durante al menos 2 segundos, entonces el sistema de alerta debe activarse a más tardar 10 segundos después de eso". En este caso, la fórmula MTL es la siguiente: $\Box(\Box_{[0,2]} detecta\_movim \rightarrow \Diamond_{(2,12]} activar\_alarma)$. Aquí, $\Diamond_{(2,12]}$ significa que *activar_alarma* será verdadero en no más de 10 segundos después de que el sensor haya detectado movimiento durante al menos 2 segundos.

MTL se define de la siguiente manera.

Definición 4.3.1. *Dado un conjunto de proposiciones atómicas P, las fórmulas MTL son construidas desde P usando conectores lógicos y versiones de restricción de tiempo del operador $\cup$ (*until*):*

$$\alpha ::= p \mid \neg\alpha \mid \alpha \wedge \alpha \mid \alpha \vee \alpha \mid \alpha \cup_I \alpha$$

donde $p \in P$ e $I \subseteq [0, \infty]$ es un intervalo de enteros[1].

El operador $\phi \cup_I \alpha$ significa: ϕ debe cumplirse desde ahora hasta algún punto dentro de I unidades de tiempo, momento en el cual α comienza a satisfacerse.

Podemos definir el operador $\Diamond_I$ (*finally*) como: $\Diamond_I \alpha \equiv true \cup_I \alpha$; y el operador $\Box_I$ (*globally*) de la siguiente forma: $\Box_I \alpha \equiv \neg\Diamond_I \neg\alpha$. Por simplicidad, la ausencia del intervalo asociado a un operador modal representa el intervalo $[0, \infty]$ ($\cup \equiv \cup_{[0,\infty]}$).

4.4 Model Checking

La verificación de modelos (*model checking*) es un método de verificación formal que explora sistemáticamente el espacio de estados de un modelo del sistema para comprobar si dichos estados satisfacen ciertas propiedades.

En el contexto de sistemas críticos, constituye una herramienta ampliamente utilizada por diseñadores y desarrolladores de software para garantizar que un modelo de sistema cumple con un comportamiento esperado.

[1] Es decir, I tiene una de las siguientes formas: (a, b), $(a, b]$, $[a, b)$ y $[a, b]$, con $I \subseteq \mathbb{N} \cup \infty$, de modo que a no puede ser ∞ y, cuando b es ∞, el intervalo debe ser abierto por la derecha.

Las técnicas de verificación se basan en modelos que describen el posible comportamiento del sistema de manera matemática, precisa y sin ambigüedades. Las herramientas de *model checking* verifican modelos especificados en un lenguaje formal (e.g., autómatas, redes de Petri, máquinas de estado, Promela, TLA+ [27]). Asimismo, las propiedades que el modelo debe satisfacer se expresan en un lenguaje formal, como la lógica proposicional, de primer orden, temporal, entre otras.

Cuando un MC concluye que un modelo no satisface una propiedad (formalmente, $M \not\models P$), retorna un contraejemplo que muestra cómo se puede alcanzar un estado que no cumple con P. El contraejemplo se representa como una secuencia de cambios de estado del modelo cuyo último estado satisface la negación de la propiedad ($\neg P$). En términos formales, un MC verifica si el lenguaje aceptado por el modelo está incluido en el lenguaje descripto por la propiedad ($\mathcal{L}(M) \subseteq \mathcal{L}(P)$). Es decir, todo el comportamiento de M debe estar incluido en el comportamiento de P. Además, los contraejemplos suelen ser utilizados para construir casos de prueba para testear una implementación. En MBT los MC han sido ampliamente usados, entre los más conocidos podemos citar: SPIN [9], Uppaal [8], Java Pathfinder [83], NuSMV [10], TLC Model Checker [84], entre otros.

En particular, para modelar sistemas de tiempo real, existen lenguajes que permiten la especificación de restricciones temporales, como los TA, RT-DEVS y Redes de Petri temporizadas, entre otros. Por otro lado, para expresar las propiedades temporales que estos modelos deben cumplir, se emplean lógicas como LTL, TCTL y MTL, que figuran entre las más utilizadas.

Sin embargo, en algunas situaciones, los lenguajes de modelado se emplean también para representar propiedades cuando no existe un *model checker* que soporte una lógica temporal determinada o cuando una propiedad compleja no puede expresarse mediante una lógica temporal convencional. En estos casos, se aplican técnicas específicas, como el uso de un autómata observador (testigo) [35, 36] para verificar la propiedad en cuestión.

Entre las herramientas para la verificación de modelos de sistemas de tiempo real, Uppaal es uno de los *model checkers* más utilizados tanto en la industria como en la academia. Se trata de un entorno de herramientas integradas para el modelado, validación y verificación de modelos basados en TA, junto con propiedades expresadas en un subconjunto de la lógica TCTL [31].

El subconjunto de TCTL admitido por Uppaal no permite versiones temporizadas de los operadores modales $\Box$ y $\Diamond$, como ocurre en MTL. Además, los TA utilizados en Uppaal son una extensión de los TA de Alur [39, 25], diseñada para simplificar y clarificar el modelado de sistemas. Algunas de estas extensiones incluyen el uso de variables enteras compartidas, canales *urgents*, nodos *commit*, invariantes de nodos y redes de TA.

El motor de verificación de modelos de Uppaal está diseñado para verificar fórmulas que tienen las siguientes formas[2] : $\mathcal{A}\Box P$ (P siempre se mantiene invariante), $\mathcal{E}\Diamond P$ (es posible alcanzar un estado en el cual P se satisface), $\mathcal{A}\Diamond P$ (siempre eventualmente el autómata alcanza un estado donde P se satisface), $\mathcal{E}\Box P$ (P es potencialmente siempre verdadera), donde P es un predicado. Estas fórmulas temporales pueden llamarse *fórmulas de caminos* porque cuantifican sobre caminos o trazas de estados del modelo. La Figura 4.1 ilustra las diferentes *fórmulas* soportadas por Uppaal. Los círculos grises (blancos) representan estados donde el predicado P (no) se cumple, mientras que los bordes enlazan estados en el mismo camino. Entonces, por ejemplo en el cuarto gráfico, si P se cumple en al menos un camino, entonces $\mathcal{E}\Box P$ se cumple.

Como podemos observar, Uppaal trabaja sobre un subconjunto de la lógica (TCTL) que difiere de MTL. MTL permite especificar RTC pero no permite distinguir entre caminos de ejecución; por el otro lado, el subconjunto de TCTL implementado por Uppaal no permite la especificación de requisitos temporales cuantitativos pero permite predicar sobre lo que ocurre

[2] $\mathcal{A}$, $\Box$, $\mathcal{E}$ and $\Diamond$ son modalidades temporales; P es un predicado. La sintaxis equivalente en Uppaal es: $\mathcal{A} \equiv$ `A`, $\mathcal{E} \equiv$ `E`, $\Box \equiv$ `[]`, $\Diamond \equiv$ `<>`.

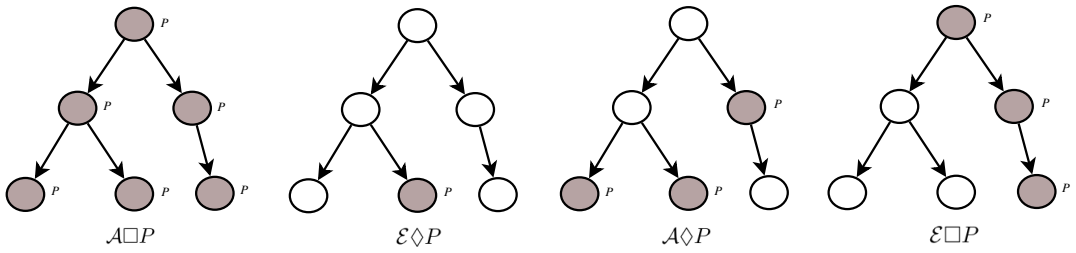


Fig. 4.1: Fórmulas TCTL de Uppaal

en distintos caminos de ejecución (como se ve en la Figura 4.1). En este trabajo mostramos cómo TCTL combinado con los TA de Uppaal pueden ser usados para verificar alguna clase importante de PTC descriptas con MTL. Para alcanzar nuestro objetivo, nos basamos en una técnica conocida como *automata observador* [35, 36], que es explicada en las Secciones 6.2 y 6.4.

4.4.1 Extensiones Uppaal a los TA

Los TA que admite Uppaal extienden los TA definidos más arriba para facilitar la tarea de modelado y verificación. Las siguientes extensiones son las utilizadas en esta tesis:

- Funciones boolenas en las guardas: permite definir funciones booleanas que pueden ser asociadas a las guardas de las transiciones. Estas funciones pueden escribirse en un estilo similar al lenguaje C.
- Canales y canales urgentes (*urgent chan*): los canales implementan la comunicación y sincronización entre múltiples TA. En particular, los canales urgentes sirven para forzar a una transición activa a dispararse sin que pase el tiempo.
- Nodos *commit*: Cuando un proceso alcanza un nodo *commit*, debe ejecutar una transición saliente de manera inmediata. Esto permite modelar secuencias atómicas de acciones.
- Variables compartidas: sirven para la comunicación asincrónica entre los TA. Se pueden usar variables enteras (también arreglos de números enteros), cada una con un dominio acotado y un valor inicial. Las variables pueden ser de acceso local a un TA o de acceso compartido a varios TA. Los predicados sobre las variables enteras se pueden usar en las guardas de las transiciones y además se pueden resetear al igual que los relojes. La sintaxis relacionada con estas variables es similar al estándar del lenguaje C.

4.5 Testeo por Mutación

El objetivo principal de la actividad de testeo es identificar errores aún no descubiertos en el software y/o en la especificación. Para garantizar un software confiable, el testeo de una especificación es tan importante como el testeo del programa, ya que detectar errores en las primeras etapas del ciclo de vida facilita su corrección y reduce la complejidad del proceso de eliminación.

Las pruebas basadas en mutación [85] constituyen un enfoque, propuesto inicialmente en [86], para evaluar la calidad de un conjunto de casos de prueba y detectar errores en una implementación. El éxito de la actividad de testeo depende de la calidad del conjunto de casos de prueba. Básicamente, este enfoque consiste en generar mutantes de un programa o especificación a partir de un conjunto de operadores de mutación diseñados para modelar errores comunes y

típicos que ocurren durante el desarrollo. Además, las pruebas basadas en mutación también se han explorado para respaldar las pruebas de integración. Delamaro et al. [87], proponen la mutación de interces para detectar errores en las conexiones entre unidades de programa, utilizando un enfoque basado en pares. Por otro lado, Ghosh y Mathur [88] propusieron la mutación de interfaz como un método para detectar errores en la interacción entre componentes de una aplicación distribuida.

Existen diversas formas de aplicar mutaciones, siendo las más comunes las basadas en código y en modelos. Sin embargo, en esta tesis, la mutación se aplica sobre las propiedades que los sistemas deben cumplir.

4.5.1 Mutación basada en código fuente

Las mutaciones basadas en código fuente constituyen una técnica empleada para medir la efectividad de un conjunto de casos de prueba, permitiendo identificar su capacidad para detectar errores en el software.

Un mutante es una copia de un programa en la que se ha introducido un pequeño cambio en el código fuente, generalmente en una sola sentencia. Estos cambios pueden incluir la sustitución de un operador por otro, la inserción o eliminación de una instrucción, o la modificación de una constante, entre otros.

Estas modificaciones permiten identificar partes del código que no están completamente cubiertas por los casos de prueba actuales, simulando errores típicos que pueden surgir durante la implementación. Las pruebas de mutación consisten en volver a ejecutar el conjunto de casos de prueba con el código modificado para determinar si las modificaciones son detectadas. Un "mutante sobreviviente" es aquel que no es identificado por ninguna prueba, mientras que un mutante "muerto" es aquel cuyas modificaciones generan un resultado distinto al del programa original.

Los mutantes generados al introducir un cambio en un programa se conocen como mutantes de primer orden [89]. Los mutantes con más de un cambio se denominan mutantes de orden superior. Debido al efecto de acoplamiento, es probable que estos mutantes sean detectados por los casos de prueba diseñados para los mutantes de primer orden. Por esta razón, en general, solo se generan mutantes de primer orden en las pruebas de mutación. Posteriormente, se ejecuta cada caso de prueba y se determina cuántos de ellos *matan* uno o más mutantes, es decir, cuántos generan un resultado distinto al del programa original.

4.5.2 Mutación basada en modelos

En el testing por mutación, y particularmente en los métodos basados en modelos, los casos de prueba se generan a partir del modelo o especificación del comportamiento de un sistema. Al introducir modificaciones en los operadores del lenguaje de modelado, se obtienen versiones erróneas del modelo con el propósito de generar trazas que distingan el comportamiento esperado del real, facilitando así la detección de errores en la implementación.

Un mutante de la especificación representa la versión de la especificación que el desarrollador interpretó como la correcta. Posteriormente, se generan trazas para diferenciar la especificación original de la especificación mutada. Estas trazas permiten definir casos de prueba que determinan si la implementación sigue el modelo mutado o el original. En el primer caso, la salida producida corresponderá al modelo mutado en lugar del original, lo que indica una posible desviación en la implementación.

Diversos estudios han aplicado el testing por mutación para validar especificaciones basadas en técnicas formales como *Statecharts* [90], Redes de Petri [91], *Estelle* [92], y TA [93, 94],

Specification and Description Language (SDL) [12], *Design by Contract* [95], entre otros. A continuación, se describen algunos de los trabajos mencionados.

En [12], los autores definen un conjunto de operadores de mutación para SDL, diseñados para modelar errores relacionados con el comportamiento de los procesos, la comunicación entre ellos, la estructura de la especificación y algunas características intrínsecas de SDL. Posteriormente, proponen una estrategia que verifica si una implementación corresponde al modelo mutado o no.

En [93], se propone un *framework* de mutación para aplicaciones de tiempo real, en el que tanto el modelo del sistema como sus mutantes se expresan mediante una variante de los TA. Además, los autores desarrollan un algoritmo basado en estos mutantes, el cual emplea técnicas de model checking para la generación de casos de prueba destinados a la validación de sistemas de tiempo real.

Khalilov et al. [95] proponen una nueva técnica para aplicar mutaciones en contratos basados en especificaciones. Aquí, presenta un conjunto de operadores de mutación diseñados para modificar contratos (*Design by Contract*) representados mediante tablas de decisión. Posteriormente, generan entradas a partir de las tablas modificadas para evaluar el impacto de las mutaciones.

4.5.3 Mutación basada en propiedades

El testing por mutación de propiedades de un sistema es una propuesta de los últimos años [96, 97, 98]. En este contexto, los mutantes son generados por:

- cambios en los modelos pero guiados por las propiedades que el sistema debe cumplir,
- cambios en las propiedades que el sistema debe verificar, en lugar de cambios en el modelo.

Las propiedades se especifican con alguna lógica. En particular, para los sistemas de tiempo real se utilizan fórmulas escritas con alguna lógica temporal como LTL, TCTL, MTL, entre otras. De esta forma, se testea si la implementación verifica o no la propiedad que el sistema debe cumplir. Luego, a través de técnicas como MC se generan trazas de modelos basadas en las mutaciones, las cuales son convertidas en entradas para la implementación.

Si bien tiene varias similitudes técnicas con la mutación de modelos, los trabajos mencionados muestran resultados novedosos.

A continuación citamos y describimos algunos trabajos relevantes.

En [96], los autores describen cómo generar casos de testeo para verificar una propiedad de seguridad específica en protocolos de comunicación, considerando posibles vulnerabilidades en la implementación. De manera intuitiva, aplican mutaciones al modelo de forma que la propiedad de seguridad sea violada. Estas mutaciones son altamente específicas y están relacionadas con errores comunes a nivel de implementación. Para la verificación, emplean la herramienta Avispa [99], cuyos contraejemplos generados representan trazas de ataque. Una traza de ataque describe una secuencia de mensajes que un intruso podría ejecutar para identificar la presencia de una falla real en la implementación. Si una traza de ataque puede reproducirse en la implementación, entonces se confirma la existencia de una vulnerabilidad de seguridad.

En [97], se presenta un mecanismo de generación selectiva de casos de prueba basado en la mutación de los requerimientos de un sistema formulados en lógica LTL. En este trabajo, se define formalmente una métrica de cobertura de propiedades que mide qué tan bien pueden excluirse las diferentes mutaciones de una propiedad LTL al generar casos de prueba. Los mutantes de las fórmulas LTL seleccionados mediante esta métrica son utilizados para construir casos de prueba con el apoyo de un MC, aunque no se menciona uno específico.

Trakhtenbrot en [98] realiza un análisis de mutaciones de propiedades temporales escritas en LTL que especifican comportamientos de sistemas reactivos. Un aspecto clave de este trabajo

es que las mutaciones introducen cambios semánticos en las fórmulas LTL, reflejando posibles interpretaciones erróneas de los diseñadores y desarrolladores. Estas mutaciones, en lugar de generarse arbitrariamente, surgen de errores conceptuales que pueden trasladarse al modelo y/o implementación. En este contexto, las fórmulas LTL analizadas describen los patrones de comportamiento definidos por Dwyer [66], que no corresponden a especificaciones de PTC. Finalmente, los autores sugieren la posibilidad de emplear un MC para generar casos de prueba basados en estos mutantes.

Mutaciones basadas en la semántica

Los métodos de mutación basados en sintaxis suelen generar un número considerable de mutantes irrelevantes, mientras que los métodos basados en semántica son más específicos y están directamente vinculados con posibles errores en el modelo o la implementación. Un mutante basado en la semántica suele introducir múltiples cambios sintácticos en la fórmula original. Sin embargo, en muchos casos, el efecto de estas mutaciones no puede ser replicado mediante mutaciones basadas en sintaxis, ya que las fórmulas resultantes de los mutantes semánticos pueden diferir esencialmente de las del requisito original. En esta tesis, presentamos un método basado en mutaciones semánticas de PTC para detectar errores en el modelo y la implementación. Estos errores están relacionados con problemas de temporización.

Capítulo 5

Caso de Estudio: Sistema de Control Ferroviario

El proceso de verificación, detección de errores y generación de casos de prueba se ejemplifica en esta tesis mediante un caso de estudio basado en un sistema ferroviario. En la Sección 5.1, describimos el problema; luego, en la Sección 5.2, presentamos su especificación utilizando el formalismo RT-DEVS. Finalmente, en la Sección 5.3, proporcionamos una especificación con TA, obtenida mediante una traducción de RT-DEVS a los TA de Uppaal. Esta traducción se emplea en la metodología de verificación de modelos RT-DEVS, desarrollada en la Sección 6.1.

5.1 Descripción del problema

El sistema controla el acceso de N trenes a un puente mediante un intercambiador de vías (en adelante, Sistema de Control Ferroviario - SCF). El puente cuenta con una única vía y un controlador automático (`Railroad-Controller`) que garantiza que, como máximo, solo un tren puede acceder a la vez. Además, a cierta distancia del puente, hay un semáforo ferroviario encargado de regular la entrada de trenes a la zona de cruce cuando el número de trenes en la zona es excesivo. Este semáforo es controlado por el mismo software que gestiona el intercambiador de vías. Cuando un tren se aproxima (`Appr`), se le puede ordenar detenerse antes de 10 unidades de tiempo (en adelante, consideraremos la unidad de tiempo como segundos). Si no recibe la orden a tiempo, debe cruzar el puente (`Cross`). En caso de detenerse (`Stop`), debe reiniciarse (`Start`) antes de cruzar. El proceso de detener y arrancar un tren requiere de un cierto tiempo.

Cuando un tren se aproxima, el controlador le ordena detenerse y lo dirige a un sector de espera dentro de la zona si otro tren está cruzando el puente; de lo contrario, le permite atravesarlo. Los trenes en el sector de espera forman una cola hasta que el controlador les conceda el paso. Si hay N trenes dentro de la zona de cruce, la entrada se deshabilita (semáforo en rojo) hasta que todos hayan cruzado el puente. Una vez que la zona queda despejada, el sistema vuelve a habilitar la entrada (semáforo en verde).

Por otro lado, el sistema cuenta con una alarma que se activa cuando el número de trenes dentro de la zona de cruce es elevado. Al activarse, la alarma emite un sonido de alerta durante un cierto tiempo y luego continúa con una luz intermitente durante algunos segundos más. Cuando un tren ingresa a la zona de cruce y el número de trenes en ella es mayor o igual a 3, la alarma se prepara para activarse (`Warning`). Si después de 2 segundos el número de trenes no ha disminuido, la alarma comenzará a sonar (`On`) en algún instante dentro de los próximos 5 segundos; en caso contrario, se desactiva (`Off`).

Cuando la alarma se activa, el sistema pone el semáforo en rojo y no permite el ingreso de nuevos trenes a la zona de cruce hasta que se desactive. Es importante tener en cuenta que,

mientras la alarma se encuentra en estado de preparación (`Warning`), los trenes aún pueden seguir ingresando a la zona. La alarma puede sonar por un máximo de 7 segundos, tras lo cual cambia al modo de luz intermitente (`Flash`), que durará hasta 3 segundos, a menos que el número de trenes disminuya por debajo de 3, en cuyo caso se desactiva antes. Al desactivarse, el sistema pone el semáforo en verde para habilitar nuevamente el ingreso de trenes a la zona de cruce.

Considerando $N = 6$ las siguientes propiedades temporales deben cumplirse:

(P1) Si los 6 trenes están dentro de la zona de cruce, ya sea aproximándose al puente, en cola o cruzando, los mismos deben atravesarlo antes de 122 segundos.

(P2) Cuando en la zona de cruce hay al menos 3 trenes la alarma debe activarse en a lo sumo 10 segundos.

(P3) Si se activa la alarma, durante 9 segundos no se permite la entrada de trenes al cruce ferroviario.

5.2 Especificación con RT-DEVS

Los modelos RT-DEVS (atómicos o acoplados) pueden ser representados con una notación gráfica muy intuitiva [19], como se puede apreciar en la Figura 5.1—el Apéndice A presenta la definición matemática del modelo RT-DEVS. Un estado `s` se representa con un círculo y el intervalo asociado `[li,ls]` describe el valor de *ti(s)*. El estado inicial se identifica con una flecha que no tiene origen. Una transición interna se representa con una flecha de línea punteada que va desde el estado origen hasta el estado destino. La etiqueta `p!v` sobre esta transición representa la ejecución de la función de salida λ, donde `v` es el valor de salida (se puede omitir si no es relevante) y `p` el puerto por el cual se envía.

Para representar una transición externa, se utilizan flechas de línea continua que conectan el estado de origen con el estado de destino. La etiqueta `p?v` indica que el valor `v` ingresa a través del puerto `p`. Tanto las transiciones internas como las externas pueden llevar etiquetas con expresiones booleanas (guardas) de la forma `[expr]`. Estas expresiones deben cumplirse para que la transición pueda activarse.

Para graficar el acoplamiento mediante puertos, los puertos salientes se representan uniendo a los bordes de los modelos RT-DEVS un triángulo con el interior relleno, mientras que los puertos de entrada se indican mediante un triángulo sin relleno. Ambos tipos de puertos se conectan utilizando líneas continuas. Para mejorar la claridad, los modelos gráficos pueden enriquecerse mostrando variables relevantes para el comportamiento. Por ejemplo, la variable booleana *input_enabled* se utiliza para modelar el funcionamiento del semáforo, que determina si se permite o no el ingreso de los trenes a la zona de cruce.

La Figura 5.1 muestra los modelos RT-DEVS que especifican el SCF. El modelo `Train` (a) representa el entorno, mientras que los modelos `Alarm` (b) y `Railroad-Controller` (c) corresponden a los componentes encargados del control del sistema. No se muestra la especificación del `Railroad-Controller`, ya que, al no contener restricciones temporales, no es relevante para el análisis de requisitos temporales cuantitativos, que es el objetivo de este trabajo.

5.3 Especificación con TA: Desde RT-DEVS a TA

Varios trabajos [100, 38, 21, 20] presentan traducciones de distintas variantes del formalismo DEVS a TA, las cuales resultan equivalentes, ya que preservan el mismo comportamiento y propiedades.

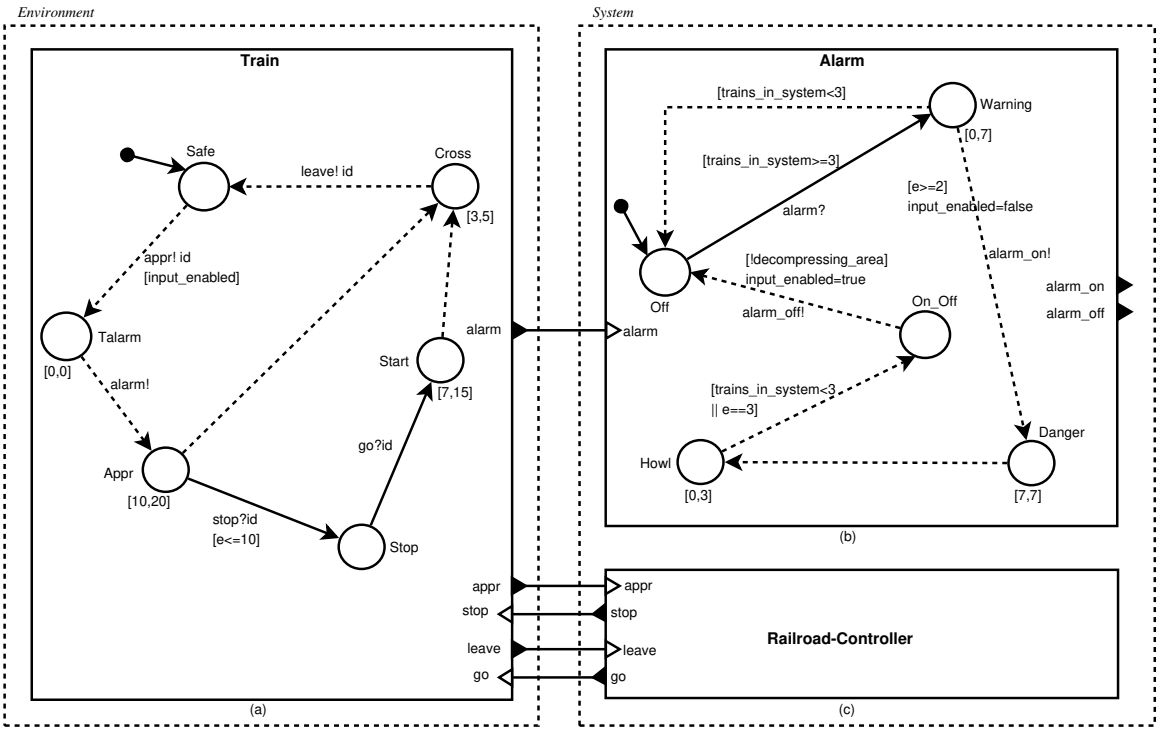


Fig. 5.1: RT-DEVS del Sistema de Control Ferroviario (SCF)

Utilizamos exactamente la misma traducción propuesta por Furfano y Nigro [20]. A continuación, se describe esta traducción aplicándola a los modelos atómicos RT-DEVS `Train` y `Alarm`, representados en la Figura 5.1, obteniendo como resultado el TA de la Figura 5.2.

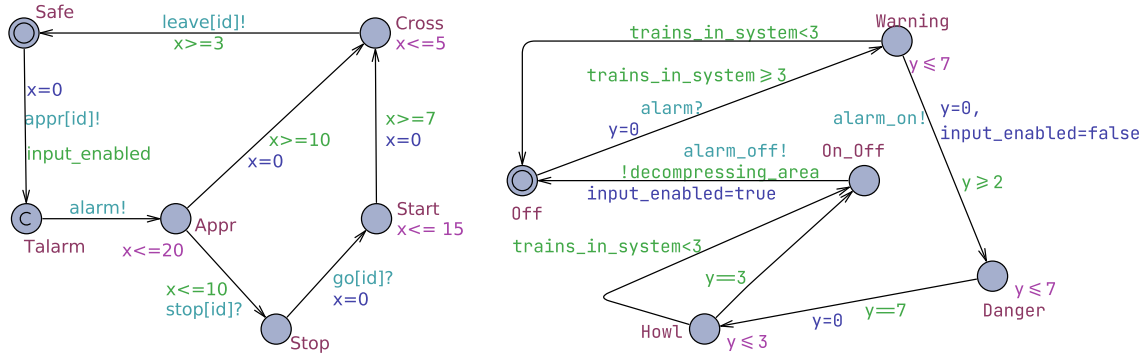
Actualmente, la traducción se realiza manualmente, siguiendo el enfoque propuesto por Furfano y Nigro. No obstante, esta tarea podría automatizarse mediante una herramienta de transformación de modelos dentro de un enfoque de Desarrollo Dirigido por Modelos [52]. De hecho, en [41, 42], hemos desarrollado herramientas de transformación de modelos capaces de tomar modelos DEVS y generar automáticamente código PowerDevs [46], lo que demuestra la viabilidad de este enfoque para optimizar el proceso de traducción.

Cada modelo atómico RT-DEVS se mapea a una plantilla (*template*) de Uppaal. En Uppaal, un TA es una instancia de una plantilla. En adelante, utilizaremos los términos TA y plantilla de manera indistinta.

El envío y recepción de mensajes a través de los puertos en RT-DEVS se modela mediante la sincronización de canales en las transiciones correspondientes del TA, mientras que los valores de los mensajes se representan mediante variables compartidas en Uppaal. Cada estado en el modelo RT-DEVS se mapea a un nodo en el TA con el mismo nombre (en adelante, lo denominaremos estado). Además, un único reloj en cada TA es suficiente para modelar el tiempo transcurrido en un estado del modelo RT-DEVS.

Sea $[inf, sup]$ el intervalo asociado a un estado en un modelo RT-DEVS. La traducción se realiza de la siguiente manera: el estado RT-DEVS se mapea a un nodo en el TA, donde el invariante es $x \leq sup$, mientras que la condición $inf \leq x$ se asocia a las guardas de las transiciones de salida de dicho nodo. Por ejemplo, en la Figura 5.1(a), el nodo `Cross` y su transición de salida hacia `Safe` ilustran esta correspondencia.

Una transición interna con salida `p!v` en el modelo RT-DEVS se representa con una transición en el TA asociada al canal de sincronización de salida `p!` que asigna el valor de salida `v` a



(a) TA Train (b) TA Alarm

Fig. 5.2: TA correspondientes a la traducción del RT-DEVS de la Figura 5.1

una variable compartida del mismo nombre en el TA. Es decir, si en el modelo RT-DEVS se transmite un valor desde un canal de salida al respectivo canal de entrada, se debe crear una variable compartida en el TA correspondiente. Un ejemplo es la traducción de la transición `leave!id` del RT-DEVS de la Figura 5.1(a), allí la variable `id` en el RT-DEVS se corresponde con la variable compartida `id` en el TA.

Una transición externa con entrada `p?v` en el modelo RT-DEVS se representa en el TA mediante una transición asociada al canal de sincronización `p?`. El valor `v` se puede leer de la variable compartida `v` en el TA (creada para el canal `p`), la cual es actualizada por el respectivo canal de salida `p!`. Es importante notar que, en las transiciones del modelo RT-DEVS, el valor `v` puede omitirse cuando no es necesario. En estos casos, no es necesario crear una variable compartida en el TA. Esta situación puede observarse comparando la transición que lleva del estado `Off` al estado `Warning` en el RT-DEVS de la Figura 5.1(b) con la transición correspondiente entre los nodos de la Figura 5.2(b).

Para modelar un estado s en el modelo RT-DEVS con $ti(s) = [0, 0]$, utilizamos un estado *committed* en Uppaal. Por ejemplo en la Figura 5.2(a), el estado `Talarm` es un estado *committed*.

La Tabla 5.1 muestra un resumen de la traducción descrita en esta sección. La segunda columna presenta los componentes que deben crearse en el TA en relación con los componentes del RT-DEVS.

Componente RT-DEVS	Componente TA
estado S	estado S
reloj e	Reloj en el TA (ej. clock x)
ta(S)=$[inf, sup]$	Invariante de estado: Inv(S) = $x \leq sup$
S ----> S'	Guarda de Transición: S $\xrightarrow{[x \geq inf]}$ S'
S $\xrightarrow{p!v}$ S'	Variable global v
S $\xrightarrow{p?v}$ S'	Canal p
	S $\xrightarrow{p!}$ S'
	S $\xrightarrow{p?}$ S'
Guarda de Transición (interna o externa)	Guarda de Transición
S $\xrightarrow{[expr]}$ S'	S $\xrightarrow{[expr]}$ S'
ta(S)=$[0, 0]$	S es un estado *committed*

TABLA 5.1: Traducción de RT-DEVS a TA

La especificación completa de RT-DEVS y todo el código de Uppaal utilizado en esta tesis se encuentran *online* en un repositorio de GitHub [101].

Capítulo 6

Verificación de Propiedades Temporales Cuantitativas

Las PTC definen restricciones de comportamiento y temporización en sistemas de tiempo real. En estos sistemas, las actividades, eventos y reacciones están sujetas a vencimientos, por lo que deben especificarse mediante un lenguaje formal.

Los siguientes son algunos ejemplos de PTC aplicados en entornos industriales:

- "Cuando la línea de comandos está operativa (modo normal), si la palanca de mando del tren de aterrizaje se ha accionado hacia la posición ABAJO y permanece en dicha posición, el tren de aterrizaje se bloqueará en posición desplegada, y las puertas se cerrarán en un tiempo inferior a 15 segundos después de la activación de la palanca." (Sistema de tren de aterrizaje de un avión [102]).

- "Si el circuito hidráulico mantiene la presión 10 segundos después de la detención de la electroválvula general, se detecta una anomalía en el sistema hidráulico." (Sistema de un tren de aterrizaje de un avión [102]).

- "El cambio de marcha debe completarse en un tiempo máximo de 1.5 segundos." (Sistema de una caja de cambios automática [103]).

- "En condiciones normales de operación, un cambio de marcha debe realizarse en un tiempo máximo de 1 segundo." (Sistema de una caja de cambios automática [103]).

Como mencionamos en capítulos previos, las lógicas temporales como TCTL o MTL son adecuadas para describir restricciones mediante fórmulas lógicas, ya que permiten la especificación explícita del tiempo en los operadores modales. Luego, a través de una herramienta de verificación formal, es posible determinar si un sistema satisface las PTC que el modelo debe cumplir. En algunos casos, estas fórmulas no se verifican formalmente debido a la falta de herramientas que las soporten; sin embargo, siguen siendo útiles para especificar restricciones temporales de manera precisa y sin ambigüedades, proporcionando una referencia clara que los desarrolladores deben respetar en la implementación.

Existen trabajos que, tras un análisis exhaustivo, identifican y definen propiedades temporales recurrentes en los sistemas de tiempo real, siendo [40] uno de los más relevantes. En este estudio, Konrad presenta un conjunto de propiedades recurrentes o patrones en estos sistemas, la mayoría de ellos como extensiones de los definidos por Dwyer. Además, distingue entre propiedades temporales cuantitativas y no cuantitativas, proporcionando la semántica de cada una mediante MTL y otras lógicas temporales.

Sin embargo, los problemas de decidibilidad de MTL dificultan la existencia de herramientas de verificación que las soporten completamente. Aun así, en [104] se menciona que las PTC más comunes, como la invariancia, la respuesta acotada y otras, pueden expresarse en fragmentos de MTL que sí son verificables mediante un model checker[1]. En esta tesis, consideramos patrones de sistemas de tiempo real verificables con un MC. Utilizamos el MC Uppaal y los patrones se especifican con TA que funcionan como TA observadores o testigos; es decir, su comportamiento en las transiciones permite determinar si el sistema satisface o no una propiedad.

A continuación, se describe una metodología para la verificación de un conjunto de PTC recurrentes en sistemas modelados con RT-DEVS. Además, esta metodología incluye un mecanismo para detectar errores por incumplimiento temporal de las propiedades y para generar casos de prueba en la implementación.

Dicha metodología contempla las contribuciones más relevantes de esta tesis y, además, es aplicable a otros formalismos de modelado de sistemas de tiempo real.

6.1 Metodología de Verificación de RT-DEVS

En esta sección, presentamos una metodología para la verificación de propiedades temporales en modelos RT-DEVS. Esta metodología se basa en:

1. Traducir los modelos RT-DEVS a TA (Sección 5.3).
2. Expresar las propiedades temporales cuantitativas usando patrones de fórmulas de lógica temporal (Sección 6.2 y 6.3).
3. Utilizar el modelchecker Uppaal para determinar si los TA del punto 1 verifican o no las propiedades temporales (Sección 6.4).
4. Generar mutaciones de los patrones del punto 2 (Capítulo 7).
5. Utilizar Uppaal para generar casos de prueba (Capítulo 8).

Elegimos Uppaal como *model checker* por dos razones: (a) en la actualidad, es uno de los MC más utilizados en la industria; y (b) los TA admitidos por Uppaal permiten expresar requisitos temporales cuantitativos que, junto con la lógica temporal que soporta (subconjusnto de TCTL), son suficientes para verificar propiedades descritas en MTL, es decir, aquellas mencionadas en el punto 2 de la metodología descrita anteriormente.

El punto 1 fue abordado en la Sección 5.3, donde mostramos cómo obtener una especificación con TA a partir de un modelo RT-DEVS, ejemplificando la transformación con el problema SCF. En el resto del capítulo, se detallan los pasos 2 y 3 de la metodología, mientras que los puntos 4 y 5 se desarrollan en los Capítulos 7 y 8, respectivamente. En todos los casos, utilizamos el caso de estudio del SCF introducido en el Capítulo 5 como ejemplo guía.

6.2 Patrones de propiedades temporales cuantitativas

Un problema que limita la adopción de tecnologías de verificación de modelos en la industria es la dificultad que enfrentan los no expertos para expresar sus requisitos en los lenguajes de especificación compatibles con las herramientas de verificación. De hecho, a menudo existe una

[1]Para una presentación detallada de MTL y una discusión sobre la decidibilidad de varios de sus fragmentos, remitimos al lector a [82].

brecha significativa entre los formatos estándar empleados en la documentación de requisitos y los formalismos utilizados por las estas herramientas, los cuales suelen basarse en lógica temporal.

Para reducir esta brecha, hemos definido un conjunto de PTC con representación textual, capaces de especificar con precisión requisitos frecuentes en sistemas de tiempo real. Luego, el ingeniero podrá verificar el cumplimiento de estos patrones recurrentes mediante la metodología presentada en la Sección 6.1.

Dado que la formalización de propiedades temporales es, en general, una tarea compleja, los patrones de propiedades temporales contribuyen a simplificar este proceso al vincularse con descripciones informales generales que representan situaciones recurrentes.

Entonces, cada patrón describe informalmente una propiedad temporal recurrente que está asociada a una fórmula temporal donde los predicados atómicos se presentan como parámetros. Los ingenieros pueden consultar una descripción informal y luego reemplazar los parámatros con predicados atómicos específicos. El patrón proporciona la estructura temporal de la fórmula; los usuarios no necesitan entender en profundidad la relación entre, por ejemplo, "*siempre*" y el operador $\Box$, ni de descripciones informales más complejas y combinaciones de las modalidades temporales y conectivos lógicos.

En consecuencia, si el desarrollador tiene una idea informal de la propiedad que su sistema debe verificar puede encontrar la fórmula temporal correspondiente buscando entre unos pocos patrones. Por ejemplo, *P must be followed by Q within k time-units* describe el patrón denominado *Time-Bounded Response* (Q responde a P), donde P, Q y k son parámetros que el diseñador debe instanciar con predicados y tiempos de su problema.

Es importante destacar que, si bien no somos los primeros en proponer patrones para propiedades en tiempo real, hasta donde sabemos, somos los primeros en definir una clase de patrones de propiedades cuantitativas en tiempo real que pueden ser verificadas mediante un model checker. Por ejemplo, los trabajos de Konrad y Cheng [40], Gruhn y Laue [105], y Abid et al. [106] presentan patrones para sistemas en tiempo real basados en los introducidos por Dwyer et al. [66]. Sin embargo, algunos de estos patrones generan fórmulas que no pueden ser verificadas mediante un model checker. A partir de un análisis basado en diversas investigaciones [107, 82, 104], hemos identificado y formalizado en MTL un subconjunto de patrones cuyas fórmulas sí pueden ser verificadas automáticamente. En esta tesis, utilizamos estos patrones para demostrar la viabilidad del enfoque de verificación formal en modelos RT-DEVS. Presentamos estos patrones, descritos en lenguaje natural y formalizados mediante la lógica MTL. Además, mostramos cómo implementarlos en Uppaal para verificar su cumplimiento.

La estrategia utilizada es la de construir un TA que representa una PTC y otro TA que representa el modelo que se quiere verificar. Al primer TA lo denominamos *TA observador* y al segundo *TA modelo*. El TA observador se comunica con el TA modelo y ambos se ejecutan en paralelo. De esta forma, a medida que el TA modelo transiciona, el TA observador controla si esas transiciones verifican o no la propiedad en cuestión. En consecuencia la verificación de la propiedad se reduce a algún problema de alcanzabilidad de un estado *bad* (cuando el sistema no satisface la propiedad) o de un estado *good* (cuando el sistema cumple con la propiedad) [108, 105] tal como se describe más adelante en la Sección 6.4. La estrategia de usar *observadores* para capturar propiedades temporales es ampliamente usada [35, 36, 106].

En la Sección 6.2.1 describimos aspectos importantes sobre la documentación de los patrones. A continuación, desde la Sección 6.2.2 hasta la Sección 6.2.4, presentamos aquellos que utilizaremos para especificar las propiedades (P1), (P2) y (P3) del SCF. Finalmente, en la Sección 6.3, se introducen otras alternativas de patrones que aparecen con frecuencia en sistemas con RTC.

6.2.1 Documentación de los Patrones

Documentamos los patrones mediante cajas denominadas *Pattern*, en las cuales se incluye la siguiente información correspondiente a cada uno:

- La representación textual del patrón.
- Una descripción informal de su semántica.
- La semántica formal expresada en MTL.
- El TA asociado al patrón, utilizado para verificar si un modelo lo satisface (ver Sección 6.4).
- Trazas características que ilustran el comportamiento del patrón.

Las trazas muestran gráficamente las ejecuciones del TA que no alcanzan el estado Error, según los instantes en que se satisfacen los predicados involucrados; es decir, trazas que verifican el patrón. Estas trazas pueden ayudar al usuario a determinar si su problema corresponde a una instancia del patrón. La notación utilizada es la siguiente:

✓ $\overset{P}{-\!\bullet\!-}$, el predicado P se cumple en el punto $\bullet$;

✓ $\overset{P...}{-\!\bullet\!-}$, el predicado P se cumple en el punto $\bullet$ y se mantiene;

✓ $\overset{k}{\vdash\!\!\dashv}$, representa un intervalo de tiempo de longitud `k`;

La caja *Pattern* 1 muestra un ejemplo de un patrón genérico.

6.2.2 Time-Bounded Response

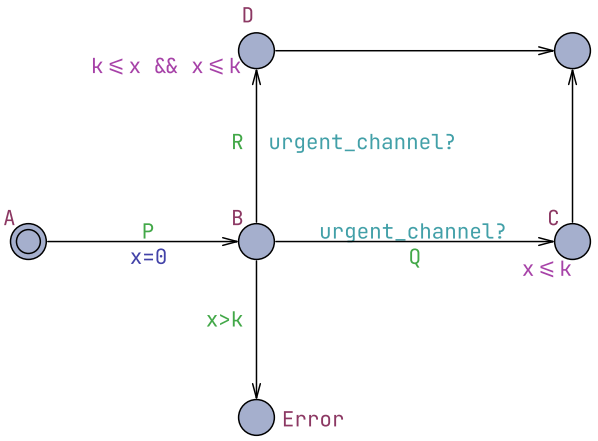
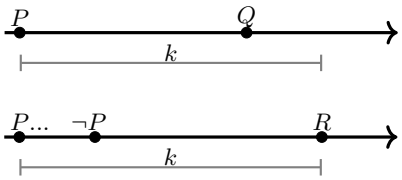
La propiedad de respuesta con límite de tiempo (*time-bounded response)* es la más recurrente en los sistemas de tiempo real. Son usadas para expresar una relación de causa-efecto, tal como el hecho de que un predicado Q debe eventualmente cumplirse luego de otro predicado P se haya cumplido. No solo debe respetar un orden sino además un límite de tiempo para Q. Por ejemplo, si el sensor detecta movimiento, la alarma debe activarse antes de los 10 segundos. El patrón es documentado en el *Pattern* 2.

Como se puede ver, en el TA observador los predicados P y Q se implementan como funciones booleanas, aunque los usuarios pueden aprovechar otras funcionalidades de Uppaal como los canales, tal como mostramos en la Sección 6.2.3.

El reloj x es utilizado para controlar que el predicado Q se satisface dentro del intervalo de tiempo $[0, k]$ desde que se cumple P. Observar que x se pone a cero cuando se satisface el predicado P (transición de A a A_B). De aquí en más los estados *bad* son etiquetados con el prefijo `Error`⟨*número*⟩ o solo `Error` si es el único estado *bad*.

La variable urg_chan es un canal urgente que fuerza la transición inmediata de A_B a B cuando la función $Q()$ se satisface.

En consecuencia, si el TA modelo transiciona de tal manera que el TA observador alcanza el estado Error significa que la propiedad ha sido violada porque, después de k unidades de tiempo (u.t.) de que P se cumple, Q permanece falso (transición directa de A_B a Error); o bien P y Q se cumplen simultáneamente (transición curva de A_B a Error).

Patrón	P must be followed by Q within k time-units or by R in exact k time-units
Descripción	Q or R se cumplen en respuesta a P. Si P se cumple en un instante $t(P)$, entonces la respuesta Q debe cumplirse antes de $t(P)+k$, o en exactamente $t(P)+k$ debe cumplirse R.
Fórmula MTL	$\Box(P \rightarrow (\Diamond_{(0,k]} Q \vee \Diamond_{[k,k]} R)$
TA Observador	
Trazas características	

Pattern 1: *Patrón Genérico*

Aplicación al SCF. La propiedad (P1) del SCF puede expresarse con este patrón instanciando los predicados P y Q y el lapso de tiempo k de la siguiente forma:

- trains_in_system == N must be followed by trains_in_system == 0 within 122 time units
- MTL: $\Box$(trains_in_system == N $\rightarrow \Diamond_{(0,122]}$ trains_in_system == 0)

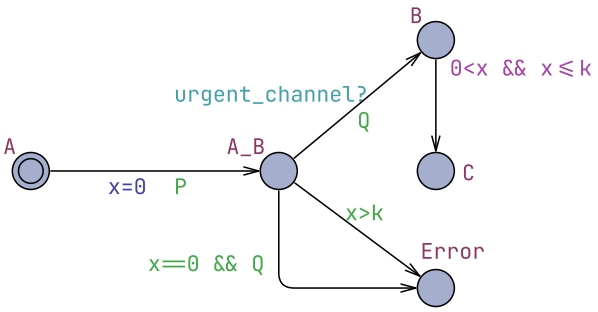
donde train_in_system coincide con la variable usada en la Figura 5.2 (b). Además, el TA observador es instanciado sustituyendo k con 122 y proporcionando las siguientes implementaciones de Uppaal para P y Q:

```
bool P() {return train_in_system == N;}
bool Q() {return train_in_system == 0;}
```

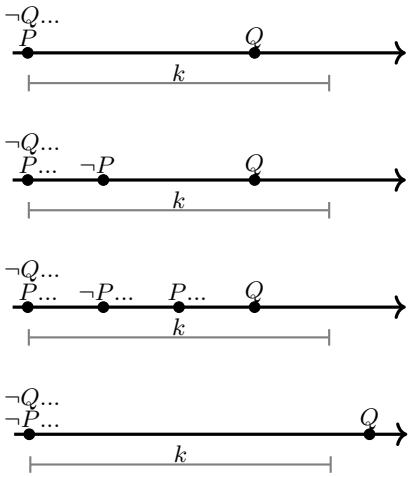
6.2.3 Time-Restricted Precedence

Las propiedades de precedencia refieren a situaciones en donde un predicado P permite que otro predicado Q se satisfaga dentro de un cierto tiempo. Es decir, Q no puede satisfacerse

Patrón	P must be followed by Q within k time-units
Descripción	Q se cumple en respuesta a P. La respuesta debe cumplirse antes de $t(P) + k$, donde $t(P)$ es el momento en que se satisface P.
Fórmula MTL	$\square(P \rightarrow \Diamond_{(0,k]} Q)$
TA Observador	



Trazas características



Pattern 2: *Time-Bounded Response*

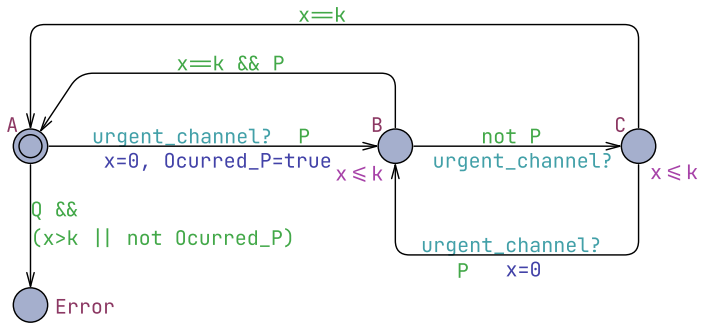
si k u.t. antes no se cumplió P. Por ejemplo, una puerta de seguridad puede abrirse en a lo sumo 20 segundos después de que una tarjeta autorizada haya sido insertada en el lector de tarjetas. Notar que Q podría no cumplirse después que P se cumplió, tal caso no invalida la propiedad. Por lo tanto, si la puerta de seguridad no se abre, la propiedad de precedencia sigue cumpliéndose.

El patrón es documentado en el *Pattern* 3.

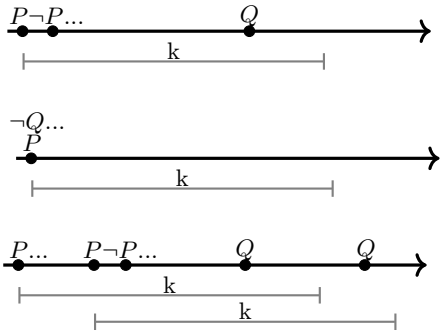
Aplicación al SCF. La propiedad (P2) del SCF se puede expresar con este patrón instanciando los predicados y la variable k de la siguiente forma:

- trains_in_system ≥ 3 enables alarm_on? for 10 time-units
- MTL: $\Diamond_{(0,10]}$alarm_on? $\rightarrow (\neg(\text{alarm\_on?}) \cup_{[0,10)} \text{trains\_in\_system} \geq 3)$

Patrón	P enables Q for k time-units
Descripción	Q se puede cumplir solo si P se cumple y el tiempo es menor o igual a $t(P) + k$; es decir, que P permite que se cumpla Q por un lapso de tiempo k. Si Q se satisface luego de k u.t. desde que se cumplió P, la propiedad es violada.
Fórmula MTL	$\Diamond_{(0,k]} Q \rightarrow (\neg Q \cup_{[0,k)} P)$
TA Observador	



Trazas características



Pattern 3: *Time-Restricted Precedence*

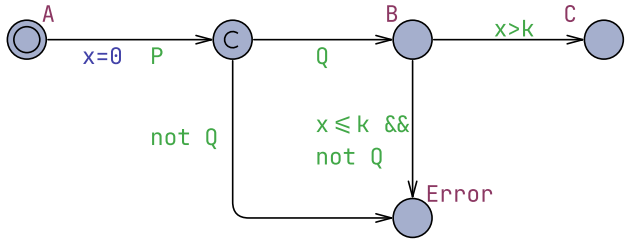
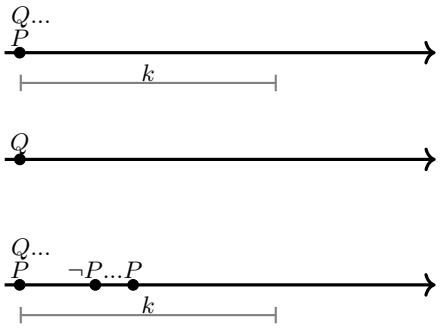
En este caso, en lugar de implementar Q mediante una función booleana, se implementa con el canal de entrada alarm_on?. Q se satisface cuando este canal recibe una señal del canal de salida correspondiente, alarm_on!, presente en el TA Alarm.

6.2.4 Conditional Security

Esta propiedad pertenece a la familia de las propiedades de *safety*. Se refiere a una situación en donde un predicado Q se cumple y se mantiene verdadero durante un período de tiempo desde el momento en que se cumple otro predicado P. Por ejemplo, si un sensor detecta algún movimiento (P), una luz indicadora debe encenderse (Q) y mantenerse encendida durante 10 segundos.

El patrón es documentado en el *Pattern* 4.

Application to the SCF. La propiedad (P3) del SCF se puede expresar instanciando este patrón de la siguiente forma:

Patrón	If P then Q holds for k time-units
Descripción	El predicado Q debe satisfacerse durante un lapso de tiempo k desde que P se cumple.
Fórmula MTL	$\Box(P \rightarrow \Box_{[0,k]}Q)$
TA Observador	
Trazas características	

Pattern 4: *Conditional Security*

- If alarm_on? then input_enabled == false holds for 9 time-units
- MTL: $\Box$(alarm_on? $\rightarrow \Box_{[0,9]}$input_enabled == false)

donde input_enabled es la variable usada en la Figura 5.2(a) and (b).

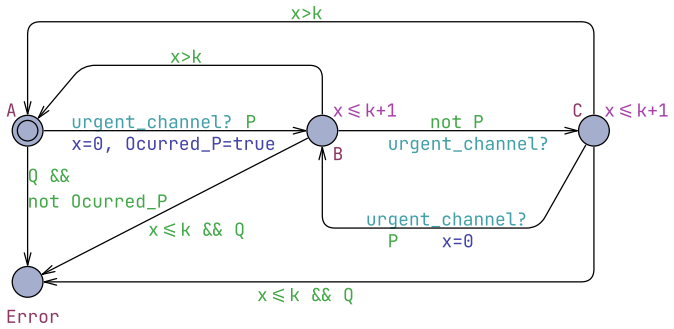
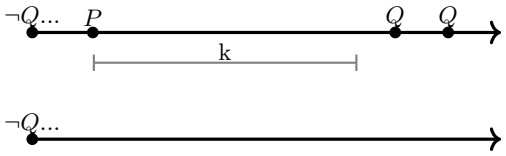
6.3 Más Patrones de PTC

A continuación, se presentan más patrones de PTC que pueden formar parte de los requisitos temporales de otros sistemas de tiempo real y, por lo tanto, pueden verificarse mediante la estrategia definida en esta tesis.

6.3.1 Precedence with Delay

Este patrón describe situaciones en las que un predicado P habilita el cumplimiento de otro predicado Q, pero solo después de un cierto período de tiempo. Es decir, si Q se satisface, implica que P ocurrió previamente en algún instante dentro de ese intervalo temporal. Por ejemplo, en un sistema de seguridad para el hogar, la activación del sistema (Q) ocurre 10 segundos después de que el usuario haya ingresado el código de seguridad (P), permitiéndole salir de la casa.

El patrón es documentado en el *Pattern* 5.

Patrón	P enables Q after k time-units
Descripción	Q se puede cumplir solo si P se cumple y el tiempo es mayor a $t(P)+k$; es decir, que P permite que se cumpla Q luego de un lapso de tiempo k. Si Q se satisface antes de k u.t. desde que se cumplió P, la propiedad es invalidada.
Fórmula MTL	$\Diamond Q \rightarrow \neg Q \cup (P \wedge P \rightarrow (\Box_{[0,k]} \neg Q))$
TA Observador	
Trazas características	

Pattern 5: *Precedence with Delay*

6.3.2 Time-Bounded Frequency

Este patrón captura situaciones en las que algo debe cumplirse nuevamente en el futuro, pero no demasiado tarde. Por ejemplo, una copia de seguridad de datos debe ejecutarse nuevamente en un plazo máximo de 30 días.

El patrón es documentado en el *Pattern* 6.

Ejemplo 6.1. Consideremos la siguiente variante del SCF (SCF_v):

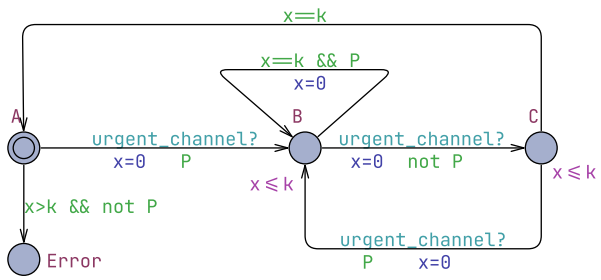
1. Se sabe que un tren volverá a ingresar a la zona de cruce antes de que transcurra un determinado tiempo (d).

2. El semáforo no cambia a rojo cuando todos los trenes se encuentran dentro de la zona de cruce. En este caso, la propiedad (P1) queda desestimada.

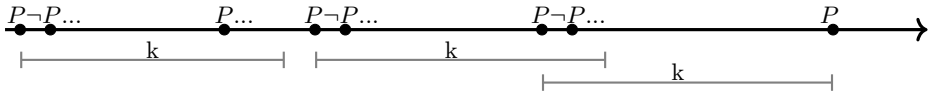
La Figura 6.1 muestra el modelo RT-DEVS de Train que especifica el cambio 1, basado en la Figura 5.1(a). En esencia, la función ahora es $ti(Safe) = [0, 50]$ (asumiendo $d = 50$). El punto 2 es implementado simplificando la especificación del RT-DEVS Railroad-Controller ignorando el caso en que configura el semáforo en rojo en el momento que todos los trenes se encuentran en la zona de cruce. Como dijimos en la Sección 5.2, no mostramos la especificación del Railroad-Controller dado que no contempla ninguna restricción temporal. □

Queremos verificar si SCF_v cumple con la siguiente propiedad:

Patrón	P occurs frequently before k time-units
Descripción	El predicado P debe cumplirse de nuevo antes de $t(P)+k$ u.t.
Fórmula MTL	$\Box \Diamond_{[0,k]} P$
TA Observador	



Trazas características



Pattern 6: *Time-Bounded Frequency*

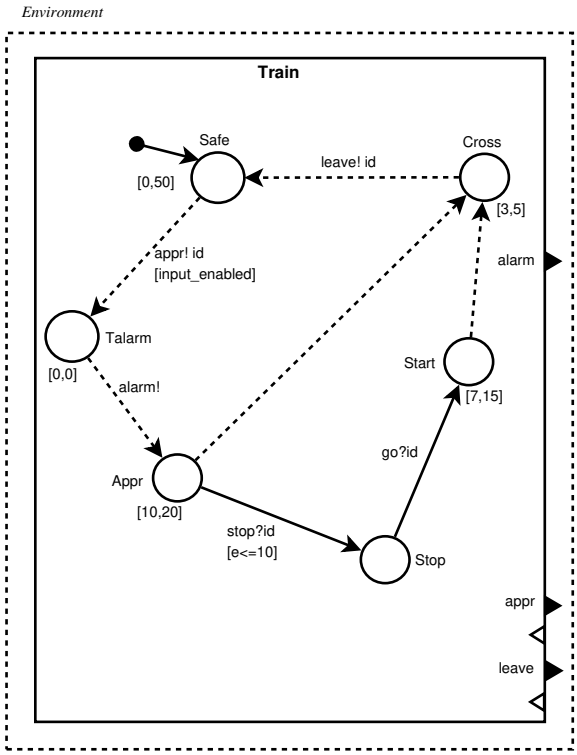


Fig. 6.1: RT-DEVS Train correspondiete al SCF_v

(P4) Dentro de la zona de cruce siempre hay al menos un tren cada 10 segundos.

La propiedad (P4) puede escribirse en términos del patrón *Time-Bounded Frequency* instanciando P and k como sigue:

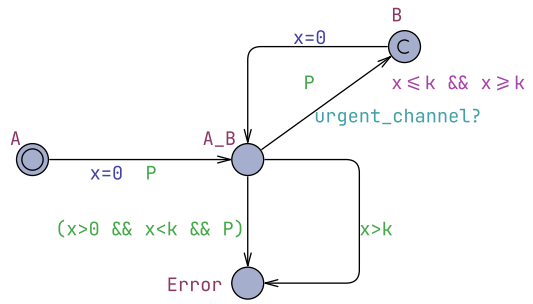
- train_in_system $>$ 0 occurs frequently before 10 time-units
- MTL: $\square \Diamond_{[0,10]}$`train_in_system > 0`

6.3.3 Time-Constant Frequency

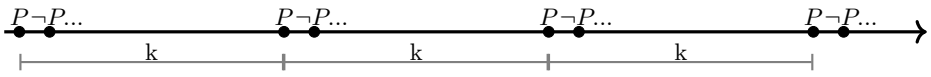
Una propiedad se satisface de manera periódica con un período constante. Por ejemplo, un sensor debe ser leído cada 100 milisegundos.

El patrón es documentado en el *Pattern* 7.

Patrón	P occurs every k time-units
Descripción	El predicado P siempre debe cumplirse en $t(P) + k$ u.t.
Fórmula MTL	$P \wedge \square_{(0,k]}(\neg P \cup_{[k,k]} P)$
TA Observador	



Trazas características



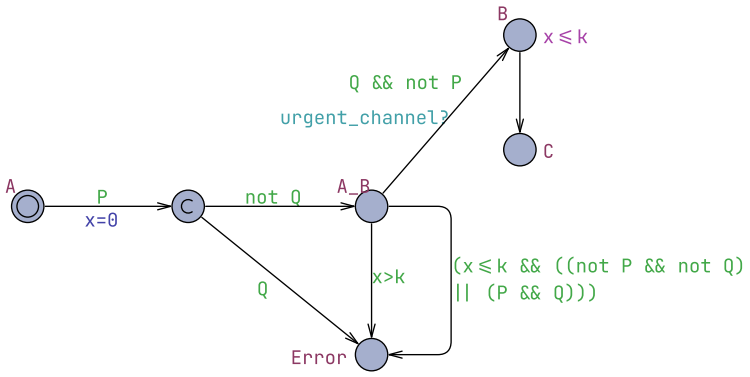
Pattern 7: *Time-Constant Frequency*

6.3.4 Time-Restricted Disable

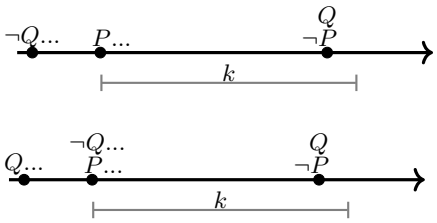
Si P se cumple, Q debe cumplirse antes de k unidades de tiempo desde que P se volvió verdadero, y cuando Q se cumple, P deja de ser verdadero; es decir, Q desactiva P en no más de k unidades de tiempo. Por lo tanto, si Q desactiva P después de k unidades de tiempo o si P deja de ser verdadero dentro del intervalo de k unidades de tiempo sin que Q sea verdadero, la propiedad es inválida. Por ejemplo, una vez que un semáforo se vuelve verde, debe pasar a amarillo en no más de 20 segundos. En este caso el cambio de estado a amarillo desactiva el verde.

El patrón es documentado en el *Pattern* 8.

Patrón	Q disables P within k time-units
Descripción	Cuando P se cumple, Q ser cumplirá en $t(P) + k'$, con $k' \leq k$, y en ese momento P deja de cumplirse.
Fórmula MTL	$\Box\,(P \rightarrow ((P \wedge \neg Q) \cup_{(0,k]} (Q \wedge \neg P)))$
TA Observador	



Trazas características



Pattern 8: *Time-Restricted Disable*

6.4 Verificación de propiedades temporales cuantitativas con Uppaal

En esta sección se presenta una técnica para determinar si un modelo RT-DEVS (M) verifica o no una propiedad temporal cuantitativa (P) instanciada a partir de uno de los patrones descritos anteriormente. La técnica es la siguiente. M y P se traducen a TA, TA_M y TA_P respectivamente, como se muestra en la Sección 5.3. TA_M corresponde al TA modelo mientras que TA_P corresponde al TA observador (ver Sección 6.2). Como dijimos más arriba, TA_M y TA_P se ejecutan concurrentemente de manera tal que este último cambia de estado según la evolución del primero. Por otro lado, se expresa en TCTL una fórmula que determina si TA_P alcanza o no un estado *bad*. Si se alcanza un estado *bad* significa que M no verifica P, caso contrario M verifica P. Cuando M no verifica P Uppaal retorna un contraejemplo representado como una secuencia de cambios de estado en TA_M y TA_P tal que el último alcanza el estado *bad*. La Figura 6.2 describe gráficamente el mecanismo descripto. La propiedad TCTL $\mathcal{A}\Box\neg TA_P.Error$ significa que TA_P nunca alcanza el estado *bad* en ninguno de los caminos de ejecución posibles. Equivalentemente se podría verificar la propiedad TCTL $\mathcal{E}\Diamond TA_P.Error$, la cual solo es válida si existe una forma de alcanzar tal estado. En otras palabras, combinando TA con fórmulas TCTL podemos verificar propiedades descriptas con MTL.

Las propiedades descriptas en el caso de estudio del SCF se verificaron con $N = 6$. Los TA

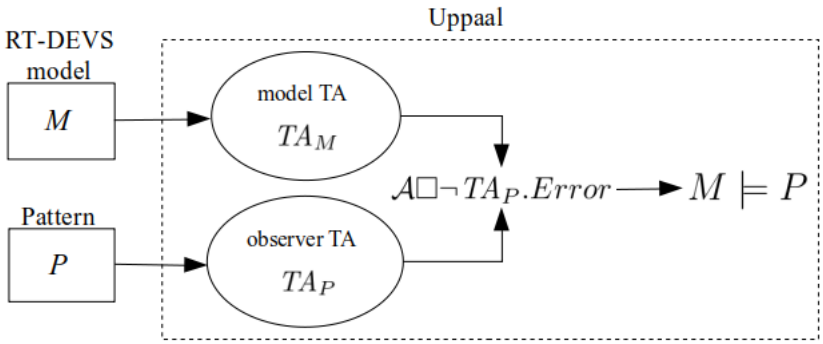


Fig. 6.2: Verificación de propiedades temporales cuantitativas con Uppaal

de los patrones se instancian definiendo un lapso de tiempo k y los predicados P y Q de la fórmula del patrón. Ejecutamos concurrentemente en Uppaal seis instancias del TA `Train`, una instancia del TA `Alarm` y una instancia del TA `Railroad-Controller`.

La tabla 6.1 muestra cómo se instanció cada patrón y el resultado de verificar la fórmula TCTL $\mathcal{A}\Box\neg TA_P.Error$, donde TA_P es el autómata del patrón correspondiente.

$N = 6$	k (seg.)	P()	Q()	$M \models Pi$
(P1)	122	`train_in_system == N`	`train_in_system == 0`	no
(P2)	10	`train_in_system >= 3`	`alarm_on?`	si
(P3)	9	`alarm_on?`	input_enabled==false	no

TABLA 6.1: Verificación de las propiedades de SCF con $N = 6$

6.4.1 Más Ejemplos de Verificación

Con el objetivo de analizar más comportamientos del SCF, se realizaron experimentos variando los parámetros de los patrones, el número de trenes e incluso utilizando la variante SCF_v para verificar el patrón *Time-Bounded Frequency*. Estos experimentos están alineados con el contexto de *Modeling and Simulation* (M&S), ya que también implican simulaciones del modelo RT-DEVS del SCF. En este caso, se trata de simulaciones que llevan al modelo a ciertos estados que invalidan una propiedad, cuando esto es posible.

Además, estos experimentos no solo permiten comprobar que el modelo cumple ciertas propiedades, sino que también resultan útiles para que un ingeniero pueda ajustar un requisito temporal, siempre que tenga la certeza de que el modelo del sistema es correcto.

Experimentos con la propiedad (P1)

La Tabla 6.2 muestra los resultados obtenidos al verificar variantes de (P1) que mantienen los predicados P y Q, pero con diferentes valores para el intervalo de tiempo k. Además, se realizaron experimentos con diferentes números de trenes (N).

Experimentos con la propiedad (P2)

La Tabla 6.3 muestra los resultados obtenidos al verificar variantes de (P2) que mantienen los predicados P y Q, pero con diferentes valores para el intervalo de tiempo k. Además, se realizaron experimentos con diferentes números de trenes (N).

N	k (seg.)	$M \models P1$
4	85	si
4	75	no
5	107	si
5	95	no
6	124	no
6	128	si

TABLA 6.2: Verificación de variantes de (P1).

N	k (seg.)	$M \models P2$
6	6	no
6	9	si
5	6	no
5	8	si
4	6	no
4	7	si

TABLA 6.3: Experimentos con la propiedad (P2).

Experimentos con la propiedad (P3)

La Tabla 6.4 muestra los resultados obtenidos al verificar variantes de (P3) que mantienen los predicados P y Q, pero con diferentes valores para el intervalo de tiempo k. Además, se realizaron experimentos con diferentes números de trenes (N).

N	k (seg.)	$M \models P3$
6	6	si
5	6	si
5	8	no

TABLA 6.4: Verificación de variantes de (P3).

Experimentos con el caso de estudio SCF$_v$

En el Ejemplo 6.1 describimos el caso de estudio del SCF$_v$, una variante del SCF presentado en el Capítulo 5. Esta versión considera que un tren vuelve a ingresar a la zona de cruce antes de que transcurra una cierta cantidad de segundos, y además, que el semáforo no cambia a rojo cuando todos los trenes se encuentran dentro de dicha zona.

Queremos verificar si el modelo que especifica el SCF$_v$ (M_v) satisface la propiedad (P4) ($M_v \models (P4)$), la cual se corresponde con el patrón *Time-Bounded Frequency*. Como mencionamos en el Ejemplo 6.1, esta propiedad establece que, cada cierto tiempo (k), debe haber al menos un tren en la zona de cruce; en otras palabras, no puede transcurrir un intervalo mayor a k u.t. con la zona de cruce vacía.

Con el fin de analizar el comportamiento del sistema, realizamos distintos experimentos variando, tanto en el modelo como en la propiedad, los siguientes parámetros:

- el intervalo de tiempo k del patrón *Time-Bounded Frequency*,

- el número de trenes (N),
- el tiempo máximo que un tren tarda en volver a ingresar a la zona de cruce (d).

En particular, para el análisis del SCF_v, la verificación de la propiedad (P4) comienza una vez que todos los trenes se encuentran dentro de la zona de cruce.

La Tabla 6.5 presenta los resultados de los experimentos.

N	d	k (seg.)	$M \models P4$
2	10	2	si
2	13	2	no
2	13	4	si
2	15	4	no
2	15	6	si
2	20	9	no
2	20	7	no
2	20	12	si
-	-	-	-
3	20	3	si
3	30	3	no
3	30	9	no
3	30	15	si
-	-	-	-
4	45	11	no
4	45	15	si
4	50	15	no
-	-	-	-
5	50	5	no
5	50	10	si

TABLA 6.5: Verificación del SCF_v.

Se puede observar que, en los casos en que el modelo satisface la propiedad, al disminuir la velocidad de ingreso de los trenes a la zona (es decir, al aumentar el valor de d), esta puede dejar de cumplirse si se mantiene constante tanto el número de trenes N como el intervalo de tiempo k. Por ejemplo, con $N = 2$, $d = 13$ y $k = 4$, la propiedad se cumple; sin embargo, con $N = 2$, $d = 15$ y $k = 4$, deja de cumplirse.

De manera similar, cuando la propiedad no se satisface para ciertos valores de los parámetros, es posible que se vuelva válida al incrementar el intervalo de tiempo k, manteniendo fijos N y d. Por ejemplo, con $N = 3$, $d = 30$ y $k = 9$, la propiedad no se cumple; pero con $N = 3$, $d = 30$ y $k = 15$, sí se cumple. Lo mismo ocurre al aumentar el número de trenes que pueden ingresar a la zona.

Capítulo 7

Encontrado errores con mutantes de patrones de propiedades temporales

Los errores en modelos RT-DEVS suelen ser difíciles de detectar, sobretodo si están relacionados a los aspectos temporales. La gran concurrencia e interacciones complejas entre los componentes hace poco visible el problema. En este sentido, los MC facilitan el análisis porque retornan un contraejemplo cuando una propiedad no se cumple. Sin embargo, a veces el contraejemplo es muy extenso y difícil de seguir. Por ejemplo, el contraejemplo resultante de la verificación de (P1) tiene 35 cambios de estados lo que dificulta su análisis. Analizar contraejemplos de modelos de nivel industrial puede resultar muy complejo [109, 110, 111]. Por esta razón, en esta sección proponemos una técnica complementaria para encontrar errores temporales cuantitativos en modelos RT-DEVS.

Si el modelo tiene errores temporales, una posible interpretación es pensar que verifica una propiedad temporal diferente. No obstante, se asume que el diseñador es experimentado y en consecuencia tales errores corresponden a leves diferencias de la versión correcta del modelo.

En este sentido, nuestra propuesta consiste en tomar la especificación de las propiedades temporales que el modelo *no* verifica, pero que *debería* verificar, y aplicarles una serie de transformaciones, llamadas *mutaciones*, que nos permitirán detectar errores sin tener que analizar contraejemplos complejos. Las mutaciones de propiedades temporales cuantitativas permiten abordar las posibles interpretaciones erróneas y fallas que pueden surgir al modelar formalmente requisitos temporales complejos.

Más concretamente la técnica que proponemos es la siguiente. El modelo RT-DEVS M debe verificar una propiedad P. Si M no verifica P nuestra conjetura es que M verifica una propiedad levemente diferente a P, que denominamos *mutante* y notamos con P'. Por lo tanto si sucede $M \not\models P$, significa que existen trazas de M que no son válidas para P, tal como se muestra en la Figura 7.1(a). Si el modelo satisface un mutante P' ($M \models P'$), se habrá descubierto el error cometido por el diseñador tal como se representa en la Figura 7.1(b). Los mutantes también son útiles para verificar si existen o no trazas del modelo que satisfacen la propiedad como se muestra en la Figura 7.1(c) y (d).

En [44] se describió una propuesta que permite generar las fórmulas P' que se habrían implementado en lugar de P. Los mutantes se obtienen trabajando a nivel de los patrones de propiedades temporales descriptos en la Sección 6.2. En consecuencia, si P es una instancia de un patrón, el diseñador puede recurrir a los mutantes de dicho patrón. En el trabajo mencionado los mutantes corresponden a mutaciones semánticas [112], lo que puede implicar introducir varios cambios sintácticos en el patrón. Las mutaciones sintácticas consisten en aplicar pequeños cambios en alguna entidad u operador para inyectar un posible error simple. Por el contrario, en las mutaciones semánticas se inyectan cambios con el fin de capturar errores de interpretación.

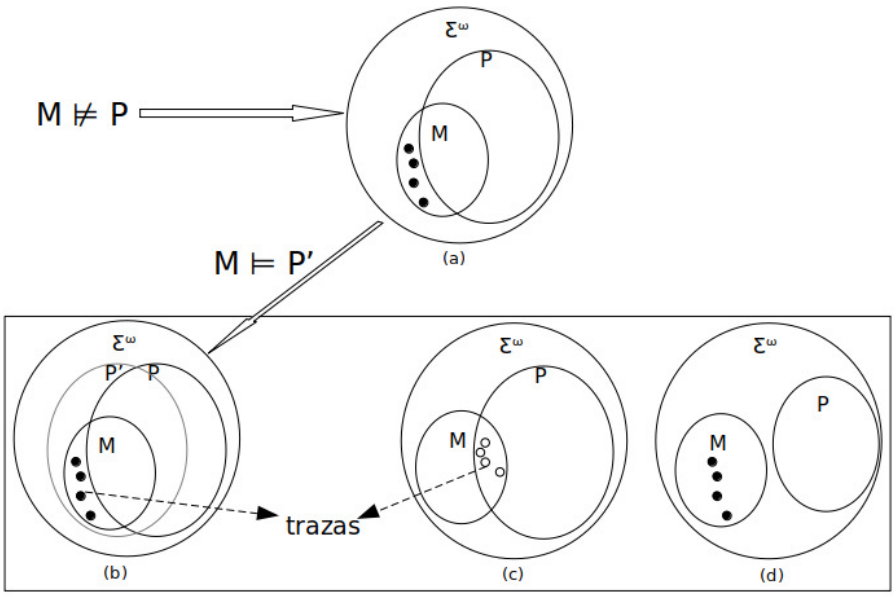


Fig. 7.1: El modelo no satisface la propiedad. Σ (conjunto de eventos o estados) el alfabeto y Σ^ω el conjunto posiblemente infinito de cadenas sobre Σ

Estas mutaciones se aplican normalmente a modelos o propiedades, aunque existen trabajos que las aplican sobre código [112].

Una vez que se identifica la causa de un error (es decir, por qué (P1) falla), el modelo RT-DEVS o los requisitos pueden modificarse de manera que la nueva versión satisfaga las propiedades deseadas. Además, cuando el modelo verifica todas las propiedades previstas (es decir, el modelo es correcto), tanto los mutantes como la verificación mediante model checking siguen siendo útiles para generar casos de prueba destinados a evaluar la implementación.

A continuación, presentamos la técnica de detección de errores en modelos RT-DEVS basada en la mutación de patrones de propiedades temporales, utilizando ejemplos extraídos del caso de estudio del SCF. En la Tabla 6.1, se observa que, con $N = 6$, el modelo RT-DEVS del cruce de trenes no satisface las propiedades (P1) y (P3). Ante esta situación, se analizan posibles errores en el modelo mediante el estudio de mutantes de (P1) y de (P3). De manera similar, este análisis puede extenderse a otras propiedades.

7.1 Mutantes del patrón Time-Bounded Response

Analizamos en esta sección posibles mutantes del patrón *Time-Bounded Response* aplicados a (P1) del caso de estudio del SCF.

Mutante 1

¿Se cumplirá (P1) si se extiende el tiempo k? Este mutante permite chequear si la respuesta Q se cumple, pero en un tiempo mayor (ver Figura 7.2). Formalmente el mutante se expresa en MTL como: $\square(P \rightarrow \Diamond_{[0,k']}Q)$, donde $k' > k$. Es decir, en lugar de tomar el intervalo $[0, k]$ tomamos $[0, k']$ donde hemos sustituido k por una constante mayor.

Por ejemplo, podríamos verificar si todos los trenes salen del cruce antes de 126 segundos, en lugar de los 122 originales. Esto se puede hacer instanciando el TA del *Pattern* 2 con $k = 126$.

Luego verificamos si M satisface o no el TA mutante (TA'_{tb_1}) con la fórmula $\mathcal{A}\Box\neg TA'_{tb\_1}.Error$, tal como se describió en la Sección 6.4.

En este caso, Uppaal responde que M verifica la propiedad mutante. Por lo tanto, el usuario puede identificar cuál es el problema con M con esta nueva información y en consecuencia corregir el modelo y volver a verificar con la propiedad original. Un posible error en el modelo podría estar relacionado al retardo de los trenes cuando salen de la cola de espera, al tiempo que tardan en cruzar el puente o a la demora en la aproximación. Por ejemplo, si establecemos que el cruce del puente puede demorar entre 3 y 4 segundos (es decir, setear $ti(Train.Cross) = [3, 4]$ en la Figura 5.1, en lugar de $[3, 5]$) y verificamos nuevamente (P1) con dicho cambio observamos que el modelo corregido satisface la propiedad.

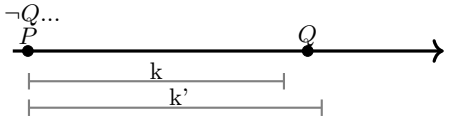


Fig. 7.2: Traza del *Mutante 1* del patrón *Time-Bounded Response*

Mutante 2

¿Existe alguna traza en donde Q se cumple antes de k u.t.? Formalmente el mutante se expresa en MTL como: $\Diamond(P \rightarrow \Diamond_{[0,k]}Q)$. Es decir, sustituimos el operador $\Box$ por $\Diamond$ al inicio de la fórmula. Este cambio captura una interpretación más débil de la propiedad que se debería cumplir puesto que se verifica si solo hay una traza en la cual la respuesta llega a tiempo. Por lo tanto, de alguna manera, estamos asumiendo que el ingeniero interpretó incorrectamente un requisito o propiedad.

Para responder esta pregunta debemos verificar si un estado *good* es alcanzable. Es decir, debemos crear una instancia del TA del patrón 2 de la Sección 6.2.2 (con $k = 122$), pero luego ejecutamos la siguiente consulta: $\mathcal{E}\Diamond TA'_{tb\_2}.B$, donde TA'_{tb_2} es la instancia del patrón. Notar que, en este caso, no es necesario introducir una mutación en el TA, ya que utilizamos una verificación TCTL diferente. En otras palabras, la mutación se implementa verificando una fórmula de alcanzabilidad adecuada para este mutante.

Si el chequeo no es satisfactorio, significa que todas las respuestas ocurren con retraso o nunca se cumplen. En el SCF, existe al menos un caso en el que los 6 trenes pueden salir de la zona de cruce en menos de 122 segundos. Sin embargo, si instanciamos el TA del patrón 2 con $k = 40$, no hay ninguna respuesta dentro de ese tiempo. Es decir, no es posible que todos los trenes crucen antes de los 40 segundos.

Mutante 3

¿Pueden P y Q cumplirse al mismo tiempo? Esta situación no es deseable en este patrón, dado que la respuesta debe cumplirse después de la pregunta. Formalmente el mutante se expresa en MTL como: $\Diamond(P \rightarrow \Diamond_{[0,0]}Q)$, sustituimos el valor k por 0. Es decir, Q debe cumplirse tan pronto como P se vuelva verdadero.

El TA del *Pattern* 2 debe mutar a TA'_{tb_3}, tal como se muestra en la Figura 7.3(a). Como podemos ver, el estado `A-B` es un estado *commit*, y k es seteado a cero. Como ya hemos explicado, un estado *commit* obliga a Uppaal a abandonarlo inmediatamente, esto permite chequear si P y Q se satisfacen al mismo momento. Por lo tanto, en este caso, si Q no se cumple en el momento en que se alcanza A-B, el estado Error nunca se alcanza.

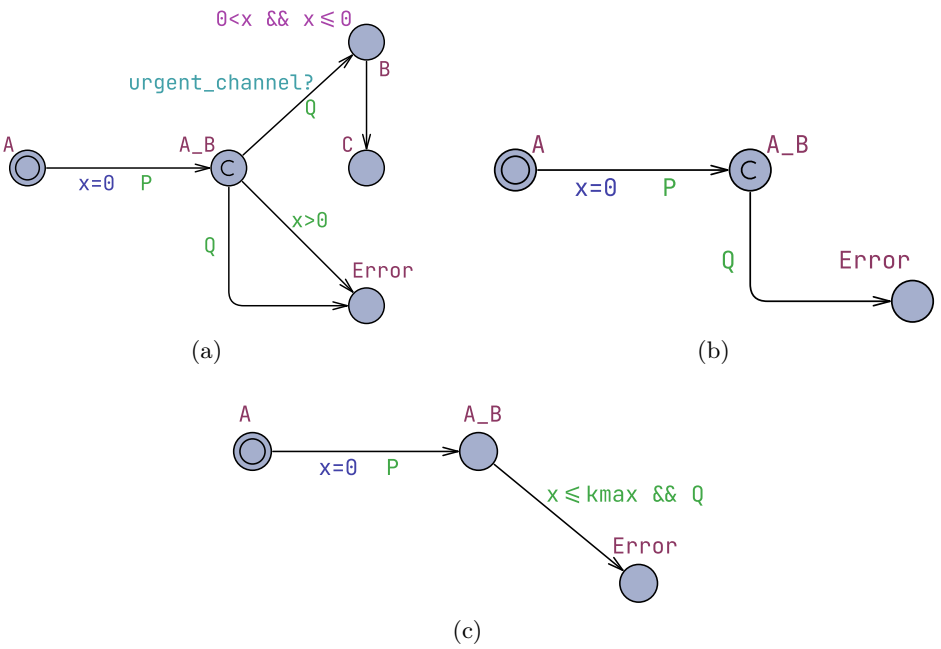


Fig. 7.3: (a) TA del *Mutante 3* del patrón *Time-Bounded Response*; (b) TA Simplificado del *Mutante 3*; y (c) TA del *Mutante 4* del patrón *Time-Bounded Response*

Sin embargo, este TA puede ser simplificado al TA que muestra la Figura 7.3(b). Dado que el invariante del nodo B es imposible de satisfacer, el fragmento del TA que comprende los nodos B y C puede ser eliminado. Además, la condición $x > 0$ es imposible de satisfacer porque A_B es un estado *commited*. De hecho, cuando el TA está en A_B, debe realizar la transición antes de que x sea incrementado desde cero. Por lo tanto, esta transición también puede ser eliminada del TA.

Para comprobar si se verifica el mutante usamos la fórmula $\mathcal{E}\Diamond TA'_{tb\_3}.B$. Si el chequeo es satisfactorio significa que al menos existe un camino en donde los predicados P y Q ocurren al mismo tiempo.

En el caso de estudio del SCF no es posible que los predicados `train_in_system == N` (P) y `train_in_system == 0` (Q) se satisfagan en el mismo tiempo dado que los trenes tardan un cierto tiempo en cruzar.

Mutante 4

¿La respuesta Q nunca se cumple? Formalmente el mutante se expresa en MTL como: $\Box(P \rightarrow \neg\Diamond_{[0,\infty]}Q)$ (o equivalentemente como $\Box(P \rightarrow \Box_{[0,\infty]}\neg Q)$). Es decir que introducimos la negación del consecuente y tomamos el intervalo $[0, \infty]$ donde hemos sustituido k por ∞.

Para trabajar con este mutante primero debemos encontrar una forma de representar ∞ en Uppaal. Una forma de hacerlo es tomar $\infty = k_{max}$, donde k_{max} es el máximo entero que admite Uppaal. En segundo lugar, debemos crear una instancia del TA del patrón 2 (TA'_{tb_4}) sustituyendo k por k_{max}, como se muestra en la Figura 7.3(c).

Finalmente, ejecutamos la expresión $\mathcal{A}\Box\neg TA'_{tb\_4}.Error$ para verificar si el modelo satisface o no el mutante.

Si la consulta es satisfactoria, TA'_{tb_4} nunca alcanza el estado Error, lo que a su vez significa que Q nunca se cumple en un tiempo "finito". Por el contrario, si el mutante llega al estado Error, sabemos que Q eventualmente se cumple, aunque no sepamos exactamente cuándo. En cualquier caso, hemos recopilado información valiosa para corregir el modelo. El TA de la Figura

7.3(c) se obtiene primero mutando el TA original y luego simplificando el TA resultante, tal como hicimos con *Mutante 3*—es decir, con las Figuras 7.3(a)-(b).

En el SCF si consideramos los predicados P y Q de la sección 6.2.2 y $k = k_{max}$ el mutante no se satisface, es decir, cuando en la zona de cruce están los $N = 6$ trenes luego de un cierto tiempo salen todos de la zona.

Sin embargo, si existiera un error en M que permitiera la satisfacción de este mutante, significaría que la zona de cruce nunca se vacía, es decir, siempre hay al menos un tren aproximándose, esperando para cruzar o atravesando el puente. En este caso, el diseñador podría sospechar que la especificación de la cola de espera o del semáforo es incorrecta.

7.2 Mutantes del patrón Conditional Security

En esta sección, analizamos posibles mutantes del patrón *Conditional Security* y ejemplificamos su aplicación en (P3) del caso de estudio del SCF.

Mutante 1

¿Se cumple la propiedad si se disminuye el lapso de tiempo?. Este mutante permite verificar si la propiedad se satisface al considerar un intervalo de tiempo menor que el de la propiedad original, como lo ilustra la traza en la Figura 7.4. Formalmente, el mutante se expresa en MTL como: $\Box(P \rightarrow \Box_{[0,k']}Q)$, con $k' < k$; es decir, en lugar de tomar el intervalo $[0, k]$ tomamos $[0, k']$ donde k ha sido reemplazado por una constante menor.

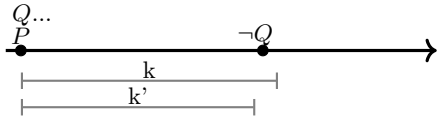


Fig. 7.4: Traza del *Mutante 1* del patrón *Conditional Security*

Por ejemplo, podríamos verificar si luego de dispararse la alarma no hay ingreso de trenes a la zona de cruce durante 6 segundos, en lugar de 9 segundos. El mutante se obtiene instanciando el TA del patrón 4 (TA'_{cs_1}) con un lapso de tiempo $k = 6$. Luego ejecutamos la consulta: $\mathcal{A}\Box\neg\, TA'_{cs\_1}.Error$. En este caso la consulta es satisfactoria, TA'_{cs_1} nunca alcanza el estado Error, esto significa que en todos los caminos el semáforo no permite el ingreso de trenes al menos durante 6 segundos desde que se dispara la alarma.

Por lo tanto, el usuario puede identificar cuál es el problema en el modelo con esta nueva información, el cual podría estar relacionado al tiempo del semáforo en rojo o a un retardo en el disparo de la alarma.

Mutante 2

¿Ocurre que $\neg Q$ vuelve a cumplirse n veces dentro del intervalo de tiempo k? Esta característica se conoce como *stuttering*. La siguiente traza en la Figura 7.5 ilustra un ejemplo de este caso.

No es posible expresar en MTL propiedades que especifiquen si un predicado se vuelve a cumplir un cierto número n de veces. No obstante, la fórmula MTL: $\Diamond(P \rightarrow \Box_{[0,k]}\Diamond\neg Q)$ proporciona una formalización aproximada que ayuda al ingeniero a entender este mutante. En este caso, se sustituye Q por $\Diamond\neg Q$ en el consecuente.

Implementamos en Uppaal una versión de este mutante que permite al usuario verificar si existe una traza en la que Q vuelva a cumplirse al menos n veces después de que se cumpla P.

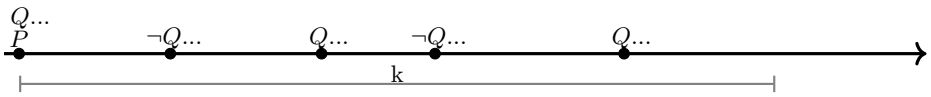


Fig. 7.5: Traza del *Mutante 2* del patrón *Conditional Security*

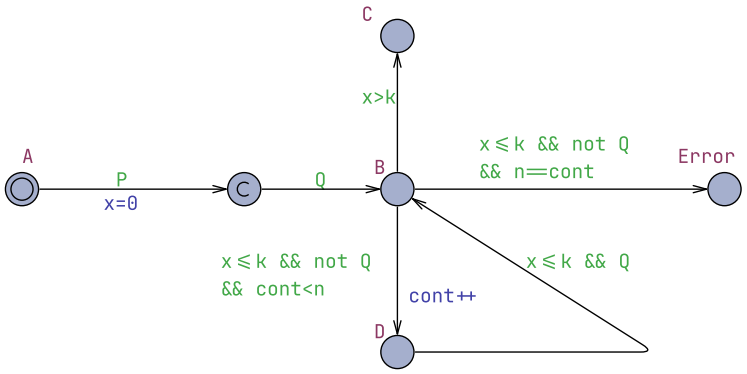


Fig. 7.6: TA del *Mutante* 2 del patrón *Conditional Security*

De este modo, n se introduce como un parámetro adicional en el TA observador que modela este mutante. Por ejemplo, si el usuario considera que el *stuttering* ocurre cuando Q vuelve a cumplirse al menos 3 veces, deberá configurar $n = 3$.

El mutante se obtiene instanciando el TA de la Figura 7.6 y asignando a n una constante que representa el número de veces que queremos verificar si $\neg Q$ vuelve a cumplirse. Por ejemplo, en el caso de estudio del SCF, si queremos verificar si, desde el momento en que se dispara la alarma, en los próximos 9 segundos el semáforo deja de estar en rojo al menos 2 veces, debemos instanciar el TA de la Figura 7.6 con $k = 9$ y $n = 2$ (TA'_{cs_2}). Luego, ejecutamos la consulta: $\mathcal{E}\Diamond\, TA'_{cs\_2}.Error$ (o, equivalentemente, $\mathcal{A}\Box\neg\, TA'_{cs\_2}.Error$).

Si el resultado es positivo, significa que dentro de los 9 segundos el semáforo dejó de estar en rojo ($\neg(input\_enabled == false)$) al menos 2 veces. En este caso particular, el resultado de la consulta es negativo. Sin embargo, si establecemos $k = 20$, la consulta retorna una traza del modelo que muestra que el semáforo deja de estar en rojo 2 veces.

Cuando la verificación arroja un resultado positivo, el ingeniero puede sospechar que la duración del semáforo en rojo y/o el tiempo que permanece activo el sonido de la alarma son demasiado breves.

Mutante 3

¿Dentro del intervalo $[0, k]$, Q deja de cumplirse en algún momento y nunca más vuelve a valer? Con este mutante, verificamos si existen trazas en las que Q deja de cumplirse en un cierto instante de tiempo t' y permanece en ese estado hasta el final del intervalo $(t', k]$. Formalmente, expresamos el mutante en MTL como: $\Diamond(P \rightarrow Q \cup_{[t',t']} (\Box_{(t',k]}\neg Q))$ (o, equivalentemente, $\Diamond(P \rightarrow (\Box_{[0,t']}Q \wedge \Box_{(t',k]}\neg Q)))$, con $t' < k$.

Como se puede observar, la semántica de este mutante introduce varios cambios sintácticos en la expresión MTL del patrón, particularmente en la subexpresión del consecuente.

Dado que no es posible determinar con precisión el instante t', implementamos el TA de este mutante de manera que el usuario pueda estimar aproximadamente a partir de qué momento $\neg Q$ comienza a cumplirse. Es decir, capturamos trazas en las que $\neg Q$ se cumple y se mantiene así

antes de un cierto tiempo t, como se ilustra en la Figura 7.7. Para implementar el TA observador de este mutante, es necesario considerar el tiempo t como un parámetro adicional.

En el caso de estudio del SCF, supongamos que queremos verificar lo siguiente: desde el momento en que se dispara la alarma, el semáforo deja de estar en rojo antes de los 7 segundos y permanece en ese estado hasta los 9 segundos. Para ello, debemos instanciar el TA de la Figura 7.8 (TA'_{cs_3}) con $t = 7$ y $k = 9$, y verificar la siguiente expresión TCTL: $\mathcal{E}\Diamond TA'_{cs\_3}.Error$. En este caso, el resultado de la consulta es afirmativo, por lo que Uppaal retorna una traza que caracteriza al mutante. Un posible error en el modelo podría estar en una transición que provoca un cambio de estado del semáforo antes de lo esperado.

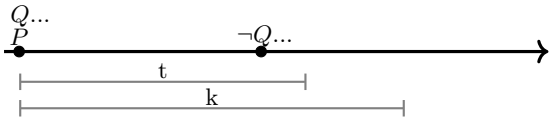


Fig. 7.7: Traza del *Mutante 3* del patrón *Conditional Security*

Mutante 4

¿Q comienza a cumplirse con retraso? Este mutante permite verificar si Q se satisface y permanece activo, pero solo a partir de un tiempo $0 < t' < k$. Con este mutante, capturamos trazas en las que Q no se cumple en el intervalo $[0, t']$, pero una vez que se satisface, se mantiene hasta el final del intervalo $(t', k]$.

Formalmente, expresamos el mutante en MTL como: $\Diamond(P \rightarrow \neg Q \cup_{[t',t']} (\Box_{(t',k]} Q))$ (o, equivalentemente, $\Diamond(P \rightarrow (\Box_{[0,t']} \neg Q \wedge \Box_{(t',k]} Q))$), con $t' < k$.

Al igual que el mutante anterior, este mutante permite al usuario estimar aproximadamente a partir de qué momento comienza a cumplirse Q.

Es decir, no es posible determinar el instante exacto en que Q se satisface, pero se puede asegurar que antes de un tiempo t comienza a valer, como se observa en la traza de la Figura 7.9.

En la implementación del TA correspondiente a este mutante, el tiempo t se considera como un parámetro adicional.

Si queremos verificar si los TA del modelo transicionan de manera que generan alguna traza que caracterice a este mutante, debemos utilizar una instancia del TA de la Figura 7.10.

En el caso de estudio del SCF, podríamos verificar si, en el momento en que se dispara la alarma, el semáforo no está en rojo, pero luego de 1 segundo cambia a rojo y se mantiene así hasta los 9 segundos. Para ello, debemos crear una instancia del TA de la Figura 7.10 (TA'_{cs_4}), asignando $t = 1$ y $k = 9$, y luego verificar la siguiente expresión TCTL: $\mathcal{E}\Diamond TA'_{cs\_4}.Error$. En

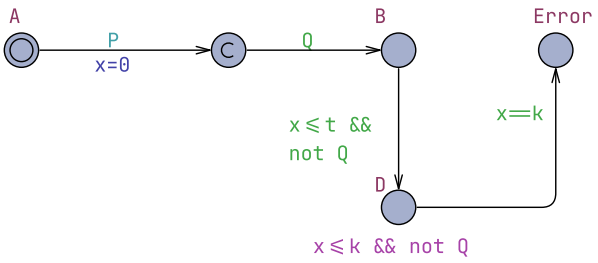


Fig. 7.8: TA del *Mutante* 3 del patrón *Conditional Security*

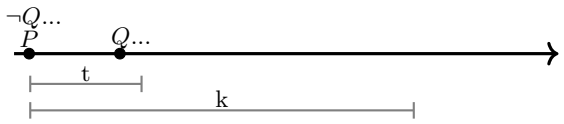


Fig. 7.9: Traza del *Mutante 4* del patrón *Conditional Security*

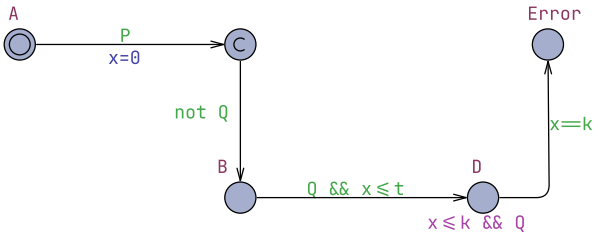


Fig. 7.10: TA del *Mutante* 4 del patrón *Conditional Security*

este caso de estudio, no es posible que los predicados que modelan el disparo de la alarma y el semáforo en rojo se cumplan simultáneamente al inicio del intervalo.

Con este patrón, no es posible determinar exactamente a partir de qué instante comienza a cumplirse Q. Sin embargo, si el usuario tiene una idea del problema, puede experimentar variando el parámetro t y aproximar dicho comportamiento. Consideramos que esta información es importante para localizar posibles errores en el modelo.

Por ejemplo, es común encontrar errores en el modelo cuando se utilizan operadores relacionales de comparación. Un caso típico es el uso incorrecto de los operadores $<$ ($>$) en lugar de $\leq$ ($\geq$). Si un error de este tipo ocurre en la guarda de una transición, puede provocar que un TA transicione más tarde de lo esperado, lo que a su vez retrasa levemente la llegada a ciertos estados del sistema. Como consecuencia, estos errores generan transiciones en los TA que producen trazas características de este mutante.

7.3 Más Mutantes de Patrones de PTC

A continuación se presentan los mutantes del patrón *Time-Restricted Precedence* y de los demás patrones definidos en la Sección 6.3. El ingeniero puede aplicar en sus problemas estos mutantes de la misma forma que aplicamos los mutantes de *Time-Bounded Response* y *Conditional Security* a las propiedades (P1) y (P2) del SCF.

7.3.1 Mutantes del patrón Time-Restricted Precedence

Se presentan en esta sección los mutantes del patrón *Time-Restricted Precedence*.

Mutante 1

¿Podría satisfacerse el patrón si se extiende el tiempo k? Este mutante permite verificar si Q se satisface después de P, pero considerando un intervalo de tiempo mayor que el de la propiedad original.

En MTL: $\Diamond_{(0,k']} Q \rightarrow (\neg Q \cup_{[0,k')} P)$, con $k' > k$.

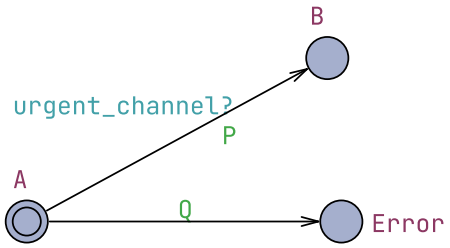


Fig. 7.11: TA del *Mutante* 2 del patrón *Time-Restricted Precedence* y del *Mutante* 2 del patrón *Precedence with Delay*

Aplicación:

- Crear una instancia del TA del *Pattern* 3 con un valor de k mayor que el de la propiedad original (TA'_{trp_1}).
- Verificar la siguiente expresión TCTL: $\mathcal{A}\Box\neg TA'_{trp\_1}.Error$. Si la respuesta es afirmativa, el modelo satisface el mutante.

Mutante 2

¿Ocurre que Q se satisface, pero previamente no se cumplió P? Este mutante permite verificar si la restricción de precedencia no se cumple, es decir, si Q es verdadero sin que P haya ocurrido antes.

En MTL: $\Diamond Q \rightarrow \neg P \cup Q$.

La Figura 7.12 muesta una traza que caracteriza al mutante.

Aplicación.

- Crear una instancia del TA de la Figura 7.11 (TA'_{pd_2}).
- Verificar la siguiente expresión TCTL: $\mathcal{E}\Diamond TA'_{pd\_2}.Error$. Si la respuesta es afirmativa, significa que existe un camino en el que Q se satisface sin que P haya ocurrido previamente.

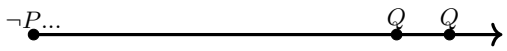


Fig. 7.12: Traza del *Mutante* 2 del patrón *Time-Restricted Precedence* y del *Mutante* 2 del patrón *Precedence with Delay*

7.3.2 Mutantes del patrón Precedence with Delay

Se presentan en esta sección los mutantes del patrón *Precedence with Delay*.

Mutante 1

¿El patrón se cumplirá si el tiempo k se reduce? Este mutante permite verificar si Q se cumple después de P, pero considerando un intervalo de tiempo más corto que el de la propiedad original.

En MTL: $\Diamond Q \rightarrow \neg Q \cup (P \wedge P \rightarrow (\Box_{[0,k']}\neg Q))$, con $k' < k$.

Aplicación:

- Crear una instancia del TA del *Pattern* 5 con un valor de k menor que el de la propiedad original (TA'_{pd_1}).
- Verificar la siguiente expresión TCTL: $\mathcal{A}\Box\neg\, TA'_{pd\_1}.Error$. Si la respuesta es afirmativa, el modelo satisface el mutante.

Mutante 2

En este patrón, también podemos utilizar el mutante 2 del patrón *Time-Restricted Precedence*, que permite verificar si la restricción de precedencia es la que no se cumple. Esto es posible porque ambas propiedades pertenecen a los patrones de precedencia. La aplicación debe realizarse de la misma manera que en 7.3.1 [Mutante 2].

7.3.3 Mutantes del patrón Time-Bounded Frequency

Se presenta en esta sección los mutantes del patrón *Time-Bounded Frequency*.

Mutante 1

¿Podría P cumplirse con frecuencia dentro de un intervalo mayor que k? Este mutante permite verificar si una condición se cumple con frecuencia, pero considerando un intervalo de tiempo más amplio que el de la propiedad original.

En MTL: $\Box\, \Diamond_{[0,k']} P$, con $k' > k$.

Aplicación:

- Crear una instancia del TA del *Pattern* 6 con un valor de k mayor que el de la propiedad original (TA'_{pbf_1}).
- Verificar la siguiente expresión TCTL: $\mathcal{A}\Box\neg\, TA'_{pbf\_1}.Error$. Si la respuesta es afirmativa, el modelo satisface el mutante.

Mutante 2

¿Se cumple P con frecuencia al menos n veces? Este mutante permite verificar si existe alguna traza en la que P se satisface nuevamente antes de que transcurran k u.t., pero solo hasta un máximo de n veces.

Como dijimos anteriormente, no es posible expresar en MTL propiedades que especifiquen si un predicado se satisface exactamente n veces. Sin embargo, si n es conocido a priori, por ejemplo, $n = 3$, este mutante puede formalizarse en MTL de la siguiente manera: $\Diamond_{[0,k]} P \cup (P \wedge (\Diamond_{[0,k]} P \cup (P \wedge (\Diamond_{[0,k]} P \cup P))))$.

Informalmente, esta expresión refleja el caso en que en el futuro P se cumplirá dentro de un lapso de k u.t.. Una vez que se cumple, volverá a cumplirse antes de que transcurran k unidades de tiempo, es decir: $\Diamond_{[0,k]} P \cup (P \wedge (\Diamond_{[0,k]} P ...)$. De esta manera, la especificación de este comportamiento se repite dos veces en la fórmula. Finalmente, después de la tercera ocurrencia de P, ya no se requiere que vuelva a cumplirse.

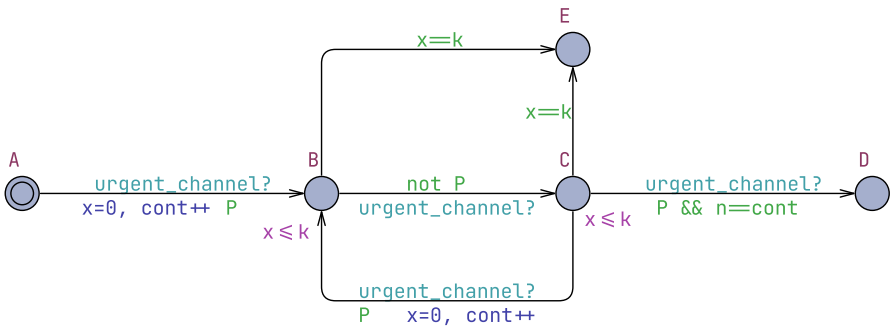


Fig. 7.13: TA del *Mutante* 2 del patrón *Time-Bounded Frequency*

Aplicación:

- La implementación del TA para este mutante considera n como un parámetro adicional. El mutante se obtiene creando una instancia del TA de la Figura 7.13, configurando el mismo tiempo k de la propiedad original y estableciendo n según la cantidad de veces que se desea verificar si P se cumple (TA'_{pbf_1}).
- Verificar la siguiente expresión TCTL: $\mathcal{E}\Diamond TA'_{pbf\_1}.D$, si la respuesta es afirmativa, entonces Uppaal retorna una traza que caracteriza el mutante.

Mutante 3

¿Se cumple P siempre dentro de intervalos consecutivos de longitud k? Este mutante captura una interpretación distinta de la frecuencia de P. Aquí, suponemos que el ingeniero interpretó que P debe cumplirse dentro de cada intervalo de longitud k. La traza de la Figura 7.14 caracteriza este mutante. Para formalizar este mutante, es necesario determinar dónde comienza cada intervalo de tiempo. Para ello, nos basamos en un predicado que especifica la ocurrencia del evento de inicio de cada intervalo de longitud k.

Expresamos este mutante en MTL de la siguiente manera: $(T \wedge \Box_{(0,k]}(\neg T \cup_{[k,k]} T)) \wedge T \rightarrow \Diamond_{(0,k]} P$, donde T es el predicado que se cumple al inicio de cada intervalo de tiempo $[0, k]$. Por ejemplo, T podría representar el comportamiento de un cronómetro que genera un evento cada vez que mide intervalos constantes de longitud k. Podemos notar que el primer miembro de la disyunción corresponde al patrón *Time-Constant Frequency* (ver Sección 6.3.3).

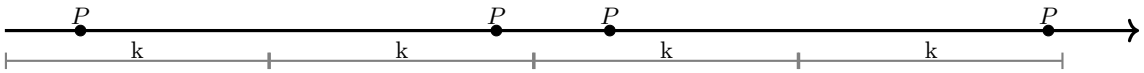


Fig. 7.14: Traza del Mutante 3 del patrón *Time-Bounded Frequency*

Aplicación:

- Crear una instancia del TA de la Figura 7.15, configurando el mismo tiempo k de la propiedad original (TA'_{pbf_3}).
- Verificar la siguiente expresión TCTL: $\mathcal{A}\Box\neg TA'_{pbf\_3}.Error$. Si la respuesta es afirmativa, el modelo satisface el mutante.

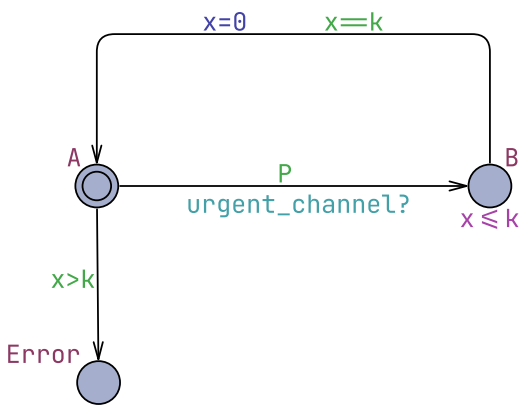


Fig. 7.15: TA del *Mutante* 3 del patrón *Time-Bounded Frequency*

7.3.4 Mutantes del patrón Time-Constant Frequency

Se presentan en esta sección los mutantes del patrón *Time-Constant Frequency*.

Mutante 1

¿Se cumplirá si se modifica el tiempo k? Este mutante permite verificar si P se cumple frecuentemente a intervalos constantes diferentes a los de la propiedad original. Es decir, busca capturar posibles errores en el modelado e implementación que surgen de fallos comunes, como utilizar el operador $<$ en lugar de $\leq$, como se explica en la Sección 7.2 [Mutante 4].

En MTL: $P \wedge \Box_{(0,k']}(\neg P \cup_{[k',k']} P)$, con $k' \neq k$.

Aplicación:

- Crear una instancia del TA del *Pattern* 7 con un valor de k diferente al de la propiedad original (TA'_{pd_1}).
- Verificar la siguiente expresión TCTL: $\mathcal{A}\Box\neg\, TA'_{pd\_1}.Error$. Si la respuesta es afirmativa, el modelo satisface el mutante.

Mutante 2

¿Se cumple P cada k u.t. y también en otros instantes? Este mutante permite verificar si P, además de ocurrir cada k u.t., también se cumple en otros momentos intermedios, como se muestra en la traza de la Figura 7.16.

En MTL : $P \wedge \Box_{(0,k]}(true \cup_{[k,k]} P)$.

Aplicación:

- Crear una instancia del TA de la Figura 7.17, configurando el mismo tiempo k de la propiedad original (TA'_{pcf_2}).
- Verificar la siguiente expresión TCTL: $\mathcal{A}\Box\neg\, TA'_{pcf\_2}.Error$. Si la respuesta es afirmativa, es modelo satisface el mutante.

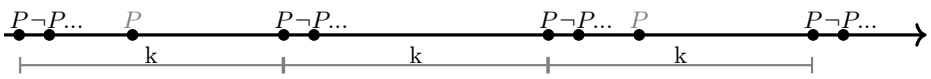


Fig. 7.16: Traza del *Mutante* 2 del patrón *Time-Constant Frequency*

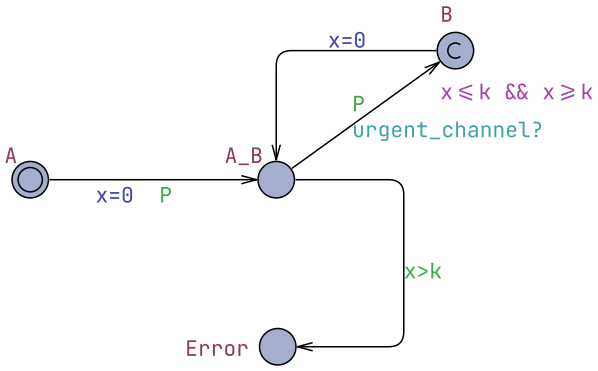


Fig. 7.17: TA del *Mutante* 2 del patrón *Time-Constant Frequency*

7.3.5 Mutantes del patrón Time-Restricted Disable

Se presentan en esta sección los mutantes del patrón *Time-Restricted Disable*.

Mutante 1

¿Podría Q desactivar a P más tarde? Este mutante permite verificar si Q siempre desactiva a P, pero en un intervalo de tiempo mayor al de la propiedad original.

En MTL: $\square\,(P \rightarrow ((P \wedge \neg Q)\, \cup_{(0,k']}\, (Q \wedge \neg P)))$, con $k' > k$.

Aplicación:

- Crear una instancia del TA del *Pattern* 8, configurando un valor de k mayor que el de la propiedad original (TA'_{ptrd_1}).
- Verificar la siguiente expresión TCTL: $\mathcal{A}\square\neg\, TA'_{ptrd\_1}.Error$. Si la respuesta es afirmativa el modelo satisface el mutante.

Mutante 2

¿P es desactivado en cualquier ocurrencia de Q? Este mutante captura una interpretación alternativa del ingeniero. Aquí, se asume que no hay inconveniente si Q ocurre antes de que P deje de cumplirse. Sin embargo, en el momento en que P deja de ser válido, Q debe cumplirse (es decir, Q desactiva a P).

Esta interpretación permite trazas en las que Q puede ocurrir antes de que P deje de cumplirse dentro del intervalo $[0, k]$. No obstante, al menos una ocurrencia de Q debe desactivar a P antes de que transcurran k u.t., como se muestra en las trazas de la Figura 7.18.

En MTL: $\square\,(P \rightarrow (P \cup_{(0,k]} \neg P \wedge Q))$

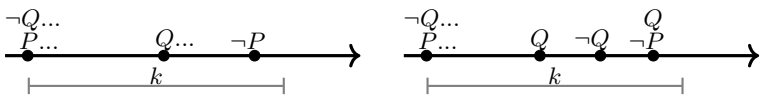


Fig. 7.18: Trazas del *Mutante* 2 del patrón *Time-Restricted Disable*

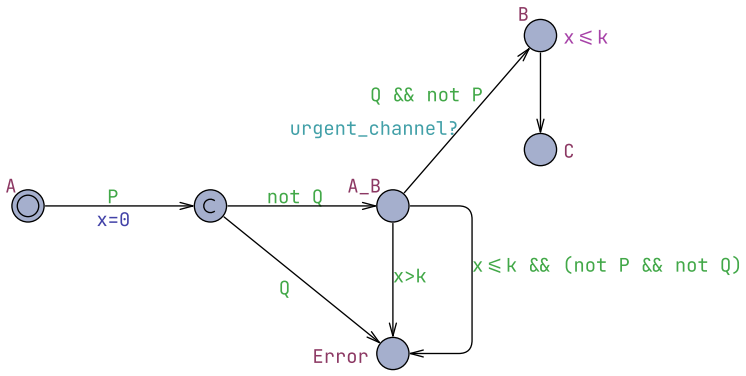


Fig. 7.19: TA del *Mutante* 2 del patrón *Time-Restricted Disable*

Aplicación:

- Crear una instancia del TA de la Figura 7.19, configurando el mismo tiempo k de la propiedad original (TA'_{ptrd_2}).
- Verificar la siguiente expresión TCTL: $\mathcal{A}\Box\neg\, TA'_{ptrd\_2}.Error$. Si la respuesta es afirmativa, el modelo satisface el mutante.

Mutante 3

¿Existe un caso en el que Q se cumple dentro del intervalo $[0, k]$ pero no desactiva a P? Este mutante es una variante más específica del mutante anterior y puede utilizarse cuando la verificación del Mutante 2 da una respuesta negativa. Dado que el estado *Error* en el TA del patrón puede alcanzarse por múltiples caminos, este mutante permite detectar un caso particular. En este caso, identifica una traza específica, como la mostrada en la Figura 7.18, aunque también pueden existir otros caminos que lleven al estado *Error*.

En MTL: $\Diamond\ (P \rightarrow ((P \wedge \neg Q) \cup_{(0,k]} Q \wedge P))$

Aplicación:

- Crear una instancia del TA de la Figura 7.20, configurando el mismo tiempo k de la propiedad original (TA'_{ptrd_3}).
- Verificar la siguiente expresión TCTL: $\mathcal{E}\Diamond\, TA'_{ptrd\_3}.D$. Si la respuesta es afirmativa, Uppaal retorna una traza que caracteriza el mutante.

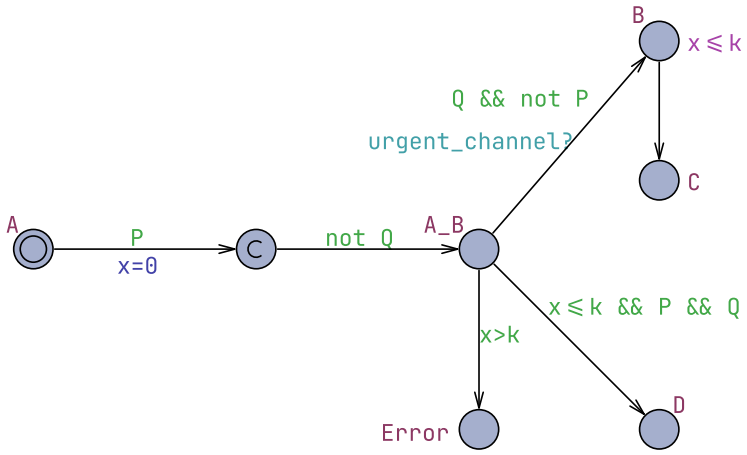


Fig. 7.20: TA del *Mutante* 3 del patrón *Time-Restricted Disable*

Mutante 4

¿Puede P dejar de cumplirse antes de que Q lo desactive dentro de k u.t.? Este mutante permite verificar si existe alguna traza en la que P deja de cumplirse antes de que Q ocurra, dentro del intervalo de k u.t., como se muestra en la Figura 7.21.

En MTL: $\Diamond\ (P \rightarrow ((P \wedge \neg Q \cup_{(0,k]} \neg P \wedge \neg Q) \wedge \Diamond_{(0,k]} Q))$

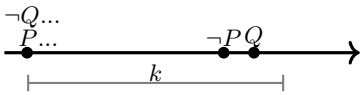


Fig. 7.21: Trazas del *Mutante* 4 del patrón *Time-Restricted Disable*

Aplicación:

- Crear una instancia del TA de la Figura 7.22, configurando el mismo tiempo k de la propiedad original (TA'_{ptrd_4}).
- Verificar la siguiente expresión TCTL: $\mathcal{E}\Diamond TA'_{ptrd\_4}.E$. Si la respuesta es afirmativa, Uppaal retorna una traza que caracteriza el mutante.

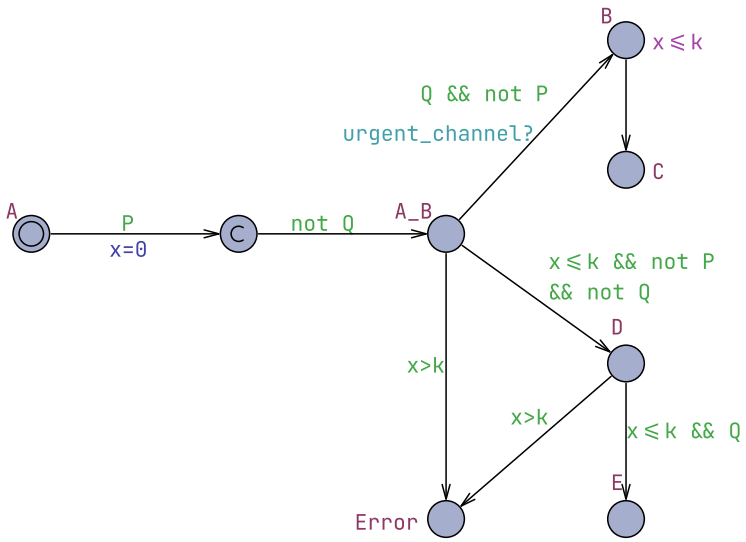


Fig. 7.22: TA del *Mutante* 4 del patrón *Time-Restricted Disable*

Capítulo 8

Generación de Casos de Prueba

Las pruebas de software se reconocen como una fase crítica en el ciclo de vida del desarrollo de software y representan una parte significativa del esfuerzo de desarrollo. Una forma de reducir este esfuerzo, al tiempo que se garantiza la efectividad de las pruebas, es la generación automática de casos de prueba a partir de artefactos utilizados en las primeras fases del desarrollo. Se han propuesto diversas técnicas para la generación de casos de prueba [113, 114, 115, 116], incluyendo enfoques aleatorios, orientados a caminos, orientados a objetivos y basados en modelos. En particular, las técnicas basadas en modelos o especificaciones han cobrado un gran interés en los últimos años.

A pesar de la principal limitación de las pruebas, que solo pueden demostrar la presencia de errores y nunca su ausencia [117], estas desempeñan un papel indispensable en el desarrollo de sistemas de software y hardware confiables. Las especificaciones formales constituyen un recurso valioso para las etapas posteriores del proceso de desarrollo de software, tal como lo señalan Stocks y Carrington [118]: *Una especificación formal de software es (también) uno de los documentos más útiles para realizar pruebas de software.*

Cuando el comportamiento de los sistemas es modelado utilizando lenguajes formales, el *Model-Based Testing* (MBT) es una herramienta fundamental para la generación de casos de prueba. El MBT se vincula al proceso y las técnicas para la derivación automática de casos de prueba a partir de modelos, la generación de scripts ejecutables y la ejecución manual o automatizada de estos. Además, incluye el modelado para la generación de pruebas, la reutilización de los modelos de requisitos, los criterios de selección de pruebas y la transformación de pruebas abstractas en pruebas ejecutables concretas.

En particular, el *model checking* es una técnica ampliamente utilizada en MBT. En este enfoque, la violación de una propiedad se expresa mediante un contraejemplo que representa la prueba que la invalida. En el contexto de los sistemas de tiempo real, tales contraejemplos se presentan en forma de trazas temporizadas, es decir, secuencias de cambios de estado y tiempos que llevan al sistema a un estado que viola la propiedad. Luego, estas trazas temporizadas (en adelante, trazas) pueden transformarse en casos de prueba para una implementación [96, 97]. Cuando estas trazas se generan tras inyectar un fallo en el modelo (mutante), suelen denominarse trazas *falsas*, ya que pertenecen a un modelo incorrecto. En consecuencia, la implementación debe ser capaz de matar al mutante. Si un desarrollador contara con una herramienta que lo asistiera en la generación de casos de prueba a partir de los mutantes, dispondría de una metodología sistemática y cuantificable para analizar las propiedades que su sistema debería verificar.

En nuestro trabajo, definimos un enfoque alineado con los principios de MBT para la generación de casos de prueba; no solo utilizamos modelos, sino que además hacemos uso de las PTC y sus mutantes. Los trabajos que describen metodologías de generación de casos basadas en

propiedades temporales son muy escasos. Por ejemplo, [97] es un trabajo relevante que presenta ciertas similitudes con nuestra propuesta. En este trabajo, los autores combinan criterios de métodos tradicionales con propiedades temporales en LTL. Proponen un criterio de cobertura basado en una métrica de cobertura de propiedad (*property-coverage metric*), que evalúa qué tan bien una propiedad LTL es cubierta por un conjunto de pruebas. Posteriormente, a partir de mutaciones del requisito, generan casos de prueba utilizando los contraejemplos proporcionados por un MC al verificar las subfórmulas de la propiedad LTL. Sin embargo, no emplean PTC ni las estrategias de generación de casos de prueba que se definen en esta tesis.

La generación de casos de prueba para implementaciones de sistemas modelados con el formalismo DEVS es un área de investigación en crecimiento. Existen trabajos que abordan este problema mediante la generación de pruebas basadas en simulaciones de modelos DEVS [119]; es decir, mediante simulaciones podríamos detectar que un modelo no satisface una cierta propiedad. Posteriormente, la traza generada por la simulación puede utilizarse para construir un caso de prueba y verificar si la implementación también incurre en el mismo error. Sin embargo, dado que las ejecuciones de un modelo mediante simulación no garantizan la exhaustividad respecto a todos los posibles caminos de ejecución, no es posible asegurar que el modelo alcanzará un estado que viole una propiedad del sistema. Además, con estas metodologías basadas en simulaciones, resulta difícil determinar qué escenarios generarían, a través de ejecuciones, casos de prueba que permitan verificar si la implementación no satisface alguna propiedad específica e importante del sistema.

En esta tesis se presenta una metodología para la generación de casos de prueba que utiliza modelos RT-DEVS y mutaciones de las PTC. Dicha metodología permite asegurar que la implementación no reproduce una serie de propiedades similares (mutantes) a la que debería implementar. Debe considerarse y emplearse como un complemento a las técnicas tradicionales [120, 121, 122] de generación de casos de prueba, e incluso como un refuerzo a las técnicas propias del área de simulación [123, 124].

Proponemos aquí dos estrategias para la generación de trazas (que luego son transformadas en casos de prueba) guiadas por mutación de PTC. En ambos casos, se utilizan los mutantes descritos en la Sección 7 y el MC Uppaal:

- Estrategia 1: partimos de la premisa de que el modelo satisface la propiedad original ($M \models P$). En este caso, podemos suponer que la implementación podría contener un error e implementar uno de los mutantes de P (P'). A partir de contraejemplos, generamos trazas del modelo (y, en consecuencia, de P) que no verifican P'. Luego, se puede analizar si la implementación se comporta como P'. En la Sección 8.1 se describe esta estrategia en detalle.
- Estrategia 2: suponemos que el modelo no satisface la propiedad original ($M \not\models P$). En este caso, generamos trazas que distinguen la propiedad original de la mutada; más precisamente, trazas del modelo que no satisfacen el patrón, pero sí a un mutante. Luego, una implementación correcta con respecto a P debería fallar. De lo contrario, si el error en el modelo se ha propagado a la implementación, su ejecución con la traza generada se comportará como el mutante. En la Sección 8.2 se describe esta estrategia.

En la Sección 8.3, mostramos cómo mapear las trazas generadas por MC a casos de prueba para una implementación.

8.1 Estrategia 1: El modelo satisface la propiedad original

Una vez que el modelo verifica todas las propiedades previstas, podemos utilizar mutantes de PTC y verificación de modelos para generar casos de prueba que evalúen la implementación del

modelo. La idea es que los mutantes representan posibles interpretaciones erróneas por parte de los desarrolladores. En otras palabras, se asume que los desarrolladores han malinterpretado el modelo, implementando una versión incorrecta, que corresponde a una implementación defectuosa. Por implementación, nos referimos al código fuente de un programa imperativo (posiblemente orientado a objetos) que fue escrito a partir del modelo. Aunque en ciertos contextos una implementación puede considerarse simplemente una versión más concreta de un modelo, los programas imperativos suelen perder muchas propiedades lógicas en comparación con aquellos descritos en lenguajes de alto nivel, como los lenguajes lógicos.

Más formalmente, sea M el modelo, I una implementación de M, P una PTC tal que $M \models P$, y P' un mutante de P. Si los desarrolladores han producido una implementación defectuosa I', podemos suponer que esta satisface el mutante P' en lugar de P. Por lo tanto, es necesario obtener trazas de M que no satisfagan P'. Cabe destacar que dichas trazas necesariamente verificarán P, dado que se asume que $M \models P$. Estas trazas pueden generarse ejecutando la siguiente consulta TCTL sobre M y el TA de P': $\mathcal{E}\Diamond TA_{P'}.error$, para algún estado *error* en $TA_{P'}$.

Actualmente, es posible generar un mayor número de trazas de este tipo considerando todos los estados *error* presentes en $TA_{P'}$. Estas trazas permitirán evaluar la implementación de manera ligeramente diferente. Supongamos que es posible transformar dicha traza en un caso de prueba, como se describe más adelante en la Sección 8.3, denotado como t, para la implementación. Si la implementación realmente implementa P', entonces t la conducirá a un estado que no satisface P, pero sí P'. Por el contrario, si la implementación implementa correctamente P, entonces t la conducirá a un estado que satisface P, pero no P'. En otras palabras, en el contexto de *mutation testing*, la ejecución de una implementación con un caso de prueba falso (normalmente generado a partir de mutantes) debe producir un fallo. Estos fallos pueden detectarse durante la ejecución o mediante el análisis de las salidas. En este caso, si la implementación corresponde a una implementación defectuosa I', entonces t es un caso falso para I'. Por lo tanto, si la ejecución de I' con entrada t produce un fallo, es probable que I' esté implementando P'. En cambio, si I es correcta con respecto a P, entonces t no producirá ningún fallo.

En resumen, dada una PCT instanciada a partir de uno de los patrones introducidos en la Sección 6.2, se recorren sistemáticamente todos sus mutantes para generar contraejemplos, los cuales serán transformados en casos de prueba. A continuación mostramos un ejemplo de esta estrategia.

Ejemplo 8.1. Si sospechamos que una implementación I' del patrón *Time-Bounded Response* considera que las respuestas deben llegar antes de $k' < k$ u.t., donde k es el intervalo de la propiedad original, podemos crear una instancia del mutante 1 (Sección 7.1) de este patrón (TA'_{tb}) utilizando k'. Luego, si verificamos $\mathcal{E}\Diamond TA'_{tb}.error$, podemos obtener una traza en la que la respuesta Q llega después de k' (pero antes de k). Si transformamos esta traza en un caso de prueba y ejecutamos I' con dicha entrada, este fallará, dado que I' solo espera respuestas antes de que transcurran k' u.t.

En la propiedad (P1) del caso de estudio del SCF, si $k' = 120$, entonces I' no admitiría que los trenes salgan de la zona de cruce después de 120 u.t. Por lo tanto, un caso de prueba generado con este método y el entorno de la Figura 5.1 producirá un fallo. □

Mutantes inútiles. Encontrar trazas de M que no satisfacen un mutante P' no siempre es posible, ya que M podría verificar tanto P como P'. Por ejemplo, si el modelo satisface la propiedad "todos los trenes salen del área de cruce en un máximo de 122 segundos", entonces también verificará el mutante "todos los trenes salen del área de cruce en un máximo de 300 segundos". Por lo tanto, en el contexto de la generación de casos de prueba, los mutantes más

interesantes son aquellos que verifican $M \models P \wedge \neg P'$. Lo bueno es que, si M satisface P', la consulta TCTL mencionada anteriormente no devolverá una traza testigo.

8.2 Estrategia 2: El modelo no satisface el patrón

En la industria del software, cuando los ingenieros cometen errores en la construcción de los modelos, estos suelen propagarse a la implementación de los sistemas. Es de esperar que una implementación existente, aún cuando se encuentre en sus etapas iniciales, también contenga los mismos errores que los modelos, dado que los desarrolladores se guían por las especificaciones y tratan de respetarlas. Incluso cuando la implementación no ha comenzado su desarrollo, se entiende que los desarrolladores pueden malinterpretar una especificación y, en consecuencia, reproducir en el código los mismos errores presentes en el modelo. Por lo tanto, si al verificar una propiedad el modelo RT-DEVS no la satisface, sería útil contar con un método que genere casos de prueba para evaluar si la implementación también cometió los mismos errores que el modelo. Presentamos aquí dicho método, en el cual utilizamos el modelo incorrecto, la propiedad original y un mutante de esta que verifica el modelo.

Más formalmente, si tenemos $M \not\models P$ y $M \models P'$, donde P' es un mutante de P, queremos generar trazas $t \in M$ tales que $t \in \neg P \cap P'$, como se muestra en la Figura 7.1 (b).

Siguiendo el mecanismo de verificación de propiedades cuantitativas definido en la Sección 6.4, podemos generar estas trazas verificando la alcanzabilidad de un estado *bad* en P y un estado *good* en P' mediante la siguiente fórmula TCTL: $\mathcal{E}\Diamond P.bad \wedge P'.good$. Uppaal devolverá al menos una traza de M que verifica la expresión TCTL descrita.

Ejemplo 8.2. En la verificación de la propiedad (P1) del caso de estudio del SCF, es posible generar una traza en la que los trenes salen de la zona de cruce después de 122 u.t., pero antes de 124 u.t. Esto se logra verificando la siguiente expresión de alcanzabilidad:

$$\mathcal{E}\Diamond\, TA_{tb}.Error \wedge TA'_{tb\_1}.B$$

donde TA_{tb} es el TA de la Figura 2 y TA'_{tb_1} es el TA del mutante 1 del patrón *Time-Bounded Response* de la Sección 7.1, instanciado con $k = 124$. Si transformamos esta traza en un caso de prueba y ejecutamos la implementación I con dicha entrada, esta fallará si I es correcta respecto a P. En cambio, si no falla, significa que I está implementando P'. □

Esta estrategia permite generar trazas más específicas, ya que, en lugar de generar cualquier traza de M que no satisface P, se obtienen trazas que, además, pertenecen a un mutante configurado con ciertos parámetros que facilitan la detección de fallos en una implementación defectuosa.

8.3 Desde Trazas a Casos de Prueba

Para testear un sistema en tiempo real, se debe considerar cuándo estimular el sistema, cuándo esperar respuestas y cómo asignar veredictos (falló o no falló el sistema). En los sistemas de tiempo real, un caso de prueba es una secuencia de acciones con retardos entre ellas, introducida para estimular el sistema. Las palabras temporizadas observables (en adelante, palabras temporizadas), definidas en [25], especifican estas secuencias de acciones.

Sea A un TA definido como en la Sección 4.2 y Σ_A el conjunto de acciones asociadas a las transiciones de A. Una palabra temporizada $\sigma \in (\Sigma_A \cup \mathbb{R}_{\geq 0})^*$ tiene la forma:

$$\sigma = d_1 \cdot a_1 \cdot d_2 \cdot a_2 ... d_k \cdot a_k \cdot d_{k+1}$$

donde $d_i \in \mathbb{R}_{\geq 0}$ representa el tiempo transcurrido entre las acciones a_{i-1} y $a_i \in \Sigma_A$.

Por otro lado, considerando la definición de la Sección 4.2 una ejecución r de A se define como una secuencia de estados de A de la siguiente forma:

$$(S_0, u_0) \xrightarrow{\gamma_1} (S_1, u_1) \xrightarrow{\gamma_2} (S_2, u_2)... \xrightarrow{\gamma_n} (S_n, u_n)$$

donde S_i es el estado corriente de A, u_i la valuación de los relojes de A en S_i y γ_j es el disparo de una transición de acción $\xrightarrow{g_j, a_j, r_j}$ o retardo $\xrightarrow{d_j}$.

Generalizando, una ejecución de m TA concurrentes $(A_1, A_2, ..., A_m)$ se representa de la siguiente forma:

$$((S1_0, S2_0, ..., Sm_0), u_0) \xrightarrow{\gamma_1} ((S1_1, S2_1, ..., Sm_1), u_1)... \xrightarrow{\gamma_n} ((S1_n, S2_n, ..., Sm_n), u_n)$$

donde Sx_i es el estado corriente de A_x, u_i es la valuación de los relojes de los m TA y γ_j es el disparo de una transición de acción de un TA concurrente o el disparo de una transición temporal $((S1_{j-1}, S2_{j-1}, ..., Sm_{j-1}), u_{j-1}) \xrightarrow{d_j} ((S1_j, S2_j, ..., Sm_j), u_j)$, tal que $\forall d' : 0 \leq d' \leq d_j \cdot u_{j-1} + d' \in I(Sx_j)$, con $1 \leq x \leq m$.

Los canales de sincronización de Uppaal representan un tipo de acción. Si un TA A_i dispara una transición con un canal de salida `p!`, también produce un cambio de estado en un TA A_j (con $i \neq j$) si este tiene una transición activa (es decir, se satisface la guarda, se cumple el invariante en el estado destino y el nodo origen es el estado corriente de A_j) con un canal de entrada `p?`. A continuación se muestra una traza del SCF. Por simplicidad, mostramos solo la acción asociada a una transición. Cuando en una transición t se omite la acción, describimos el cambio de estado como $\mathsf{T}_{id}.\mathsf{S}_o\text{-to-}\mathsf{S}_d$, donde T_{id} es la instancia del TA, y S_o y S_d son los estados origen y destino de t.

Ejemplo 8.3. Si ejecutamos 3 instancias del TA `Train` (T_0, T_1 y T_2) y una instancia del TA `Alarm` (T_a) del SCF, una posible traza es:

$((T_0.Safe, T_1.Safe, T_2.Safe, T_a.Off), (T_0.x = 0, T_1.x = 0, T_2.x = 0, T_a.y = 0)) \xrightarrow{5}$
$((T_0.Safe, T_1.Safe, T_2.Safe, T_a.Off), (T_0.x = 5, T_1.x = 5, T_2.x = 5, T_a.y = 5)) \xrightarrow{T_0.appr[0]!}$
$((T_0.Talarm, T_1.Safe, T_2.Safe, T_a.Off), (T_0.x = 0, T_1.x = 5, T_2.x = 5, T_a.y = 5)) \xrightarrow{T_0.alarm!}$
$((T_0.Appr, T_1.Safe, T_2.Safe, T_a.Off), (T_0.x = 0, T_1.x = 5, T_2.x = 5, T_a.y = 5)) \xrightarrow{12}$
$((T_0.Appr, T_1.Safe, T_2.Safe, T_a.Off), (T_0.x = 12, T_1.x = 17, T_2.x = 17, T_a.y = 17)) \xrightarrow{T_0.Appr-to-Cross}$
$((T_0.Cross, T_1.Safe, T_2.Safe, T_a.Off), (T_0.x = 12, T_1.x = 17, T_2.x = 17, T_a.y = 17)) \xrightarrow{T_1.appr[1]!}$
$((T_0.Cross, T_1.Talarm, T_2.Safe, T_a.Off), (T_0.x = 0, T_1.x = 0, T_2.x = 17, T_a.y = 17)) \xrightarrow{T_1.alarm!}$
$((T_0.Cross, T_1.Appr, T_2.Safe, T_a.Off), (T_0.x = 0, T_1.x = 0, T_2.x = 17, T_a.y = 17)) \xrightarrow{2}$
$((T_0.Cross, T_1.Appr, T_2.Safe, T_a.Off), (T_0.x = 2, T_1.x = 2, T_2.x = 19, T_a.y = 19)) \xrightarrow{T_1.stop[1]?}$
$((T_0.Cross, T_1.Stop, T_2.Safe, T_a.Off), (T_0.x = 2, T_1.x = 2, T_2.x = 19, T_a.y = 19)) \xrightarrow{2}$
$((T_0.Cross, T_1.Stop, T_2.Safe, T_a.Off), (T_0.x = 4, T_1.x = 4, T_2.x = 21, T_a.y = 21)) \xrightarrow{T_2.appr[2]!}$
$((T_0.Cross, T_1.Stop, T_2.Talarm, T_a.Off), (T_0.x = 4, T_1.x = 4, T_2.x = 0, T_a.y = 21)) \xrightarrow{T_2.alarm!}$
$((T_0.Cross, T_1.Stop, T_2.Appr, T_a.Warning), (T_0.x = 4, T_1.x = 4, T_2.x = 0, T_a.y = 0)) \xrightarrow{T_0.leave[0]!}$
$((T_0.Safe, T_1.Stop, T_2.Appr, T_a.Warning), (T_0.x = 4, T_1.x = 4, T_2.x = 0, T_a.y = 0))...$ □

Por lo tanto, para pasar de trazas a casos de prueba, es necesario transformar las ejecuciones de redes de TA en palabras temporizadas. Una ejecución puede transformarse en una palabra temporizada considerando los siguientes casos:

1. Si $\gamma_i = \xrightarrow{d_i}$ y $\gamma_{i+1} = \xrightarrow{g_{i+1}, a_{i+1}, r_{i+1}}$, entonces $d_i \cdot a_{i+1}$ es sumada a la palabra temporizada.

2. Si $\gamma_i = \xrightarrow{g_i,a_i,r_i}$ y $\gamma_{i+1} = \xrightarrow{g_{i+1},a_{i+1},r_{i+1}}$, entonces $a_i \cdot 0 \cdot a_{i+1}$ es sumada a la palabra temporizada.

3. Si $\gamma_i = \xrightarrow{d_i}$ y $\gamma_{i+1} = \xrightarrow{d_{i+1}}$, entonces $d_i + d_{i+1}$ es sumada a la palabra temporizada.

Por ejemplo, la ejecución del SCF que se muestra en el Ejemplo 8.3, se corresponde con la palabra temporizada: $\sigma = 5 \cdot T_0.appr[0]! \cdot 0 \cdot T_0.alarm! \cdot 12 \cdot T_0.Appr - to - Cross \cdot 0 \cdot T_1.appr[1]! \cdot 0 \cdot T_1.alarm! \cdot 2 \cdot T_1.stop[1]? \cdot 2 \cdot T_2.appr[2]! \cdot 0 \cdot T_2.alarm! \cdot 0 \cdot T_0.leave[0]!...$

En general, la palabra temporizada obtenida de esta manera contiene información que no es necesaria para probar la implementación. Dado que la red de TA incluye autómatas que representan tanto el sistema (por ejemplo, `Talarm` y `Railroad-Controller` en la Figura 5.1) como el entorno (por ejemplo, `Train`), la palabra temporizada contendrá acciones de ambos componentes. Para construir un caso de prueba, debemos hacer una abstracción de la ejecución y observar solo las acciones y tiempos de los TA que forman parte del entorno, ya que estas son las que estimularán el sistema. Por lo tanto, todas las acciones y retardos producidos por los TA que representan el sistema deben ser eliminados de la palabra temporizada. En el ejemplo del SCF, el TA `Train` modela el entorno, por lo que debemos capturar las acciones y retardos de todas las instancias de los trenes.

El desarrollador dispone ahora de una palabra temporizada que representa un caso de prueba para una implementación I de los TA `Talarm` y `Railroad-Controller`. Los casos de prueba deben ejecutarse en tiempo real, es decir, el sistema de ejecución de pruebas, en sí mismo, se convierte en un sistema en tiempo real.

Por ejemplo, una traza falsa generada con la estrategia de la Sección 8.2 es una ejecución (r_f) testigo del TA del modelo, el cual se ejecuta concurrentemente con el TA de la propiedad (P) y el TA del mutante (P'), que verifica la expresión TCTL $\mathcal{E}\Diamond P.bad \wedge P'.good$.

La palabra temporizada obtenida desde r_f con el método descrito especifica un caso de prueba falso cp_f. La ejecución de I con entrada cp_f debe fallar si I es conforme a las PTC del problema.

Entonces, cuando verificamos $\mathcal{E}\Diamond TA_{tb}.Error \wedge TA'_{tb\_1}.B$, Uppaal genera una ejecución (si existe) de los TA del SCF que satisface la expresión anterior, es decir, una secuencia de cambios de estado en los TA de los trenes, la alarma y el controlador que muestra un camino en el que los seis trenes salen de la zona de cruce entre 122 y 124 segundos. Luego, como se describió anteriormente, esta traza puede transformarse en una palabra temporizada falsa que especifica el caso de prueba falso. Si ejecutamos este caso de prueba sobre I, se deberá detectar un comportamiento inesperado del sistema (fallo).

No está dentro del alcance de este trabajo presentar métodos para detectar comportamientos que indiquen la violación de una propiedad en una implementación. Sin embargo, existen trabajos en la literatura que abordan este problema. Por ejemplo, en [125, 126, 127] se inyecta código en la implementación para detectar, durante la ejecución (*on-the-fly*), si el sistema alcanza un estado no deseado que invalida una propiedad expresada en MTL. Es decir, se implementa un monitor de MTL que proporciona veredictos en un formato comprensible, indicando por qué una propiedad fue satisfecha o violada durante el monitoreo en línea.

Por otro lado, las verificaciones *offline* realizan este análisis a partir de las salidas de I. Por ejemplo, los trabajos [128, 129] se basan en el análisis de trazas previamente registradas, obtenidas a partir de múltiples ejecuciones, con el fin de determinar si son conformes a una propiedad expresada en MTL.

Capítulo 9

Conclusiones y trabajo futuro

9.1 Reflexiones Finales

El formalismo DEVS es adecuado para modelar sistemas de tiempo real que deben cumplir con propiedades temporales cuantitativas. Una de las variantes más adecuada para formalizar estos problemas es RT-DEVS, dado que, a diferencia de los DEVS clásicos y otras variantes, permite especificar la ocurrecia de eventos dentro de un intérvalo de tiempo y no necesariamente en un momento concreto (o específico). Las simulaciones de estos modelos son muy útiles para recrear ciertos comportamientos de un sistema e incluso para descubrir propiedades, pero no son suficientes para asegurar la corrección de un modelo respecto a una propiedad temporal.

En el Capítulo 3 describimos diversos trabajos que proponen métodos formales para la verificación de variantes de los modelos DEVS. No obstante, dichos métodos no abordan específicamente las PTC. La lógica MTL resulta adecuada para especificar PTC, pero los problemas relacionados con la decidibilidad dificultan la construcción de MC que soporten completamente esta lógica.

Tras analizar múltiples estudios que emplean MTL para especificar PTC, identificamos un fragmento verificable mediante un MC utilizando la estrategia del modelo observador o testigo. Esta estrategia es ampliamente empleada cuando una propiedad no puede expresarse directamente en la lógica del MC. En este enfoque, se utiliza un modelo auxiliar, en nuestro caso un TA, que evoluciona en paralelo al modelo del sistema. Si el modelo observador alcanza un estado un estado que codifica o representa un error o condición anómala, se concluye que la propiedad representada por el TA ha sido violada.

Los patrones de PTC encapsulan propiedades recurrentes que los sistemas de tiempo real deben satisfacer. En esta tesis, optamos por una representación textual de los patrones debido a que los desarrolladores no siempre están familiarizados con las lógicas formales. Además, definimos la semántica de cada patrón utilizando el fragmento verificable de MTL soportado por los MC.

Los modelos RT-DEVS carecen de herramientas específicas para la verificación formal de PTC. Por ello, en esta tesis se propone un método que, utilizando una herramienta existente (Uppaal) y un formalismo ampliamente aceptado en la comunidad de sistemas de tiempo real (TA), permite no solo verificar PTC, sino también identificar posibles errores en la especificación del problema.

Para la detección de errores, empleamos mutantes de la PTC que el sistema debe satisfacer. Nuestra hipótesis es que el diseñador puede haber modelado incorrectamente debido a una interpretación distinta de la propiedad, consistente con la semántica de algún mutante. Asumimos que el diseñador posee experiencia, por lo que los errores que comete no son groseros, sino que surgen de una interpretación ligeramente diferente del comportamiento temporal del sistema.

Además, los mutantes se utilizan para generar casos de prueba destinados a evaluar una implementación. Estos casos de prueba permiten verificar si la implementación ha introducido los mismos errores presentes en el modelo, ya sea debido a la propagación de errores o a interpretaciones divergentes de los requisitos por parte de los desarrolladores. El método descrito en el Capítulo 8 genera entradas específicas para probar una implementación. Cabe destacar que este enfoque no sustituye los métodos tradicionales de testing, sino que está diseñado para complementarlos, ofreciendo una capa adicional de validación.

En resumen, en esta tesis presentamos una herramienta para la comunidad de diseñadores y desarrolladores de modelos RT-DEVS que permite:

- Verificar propiedades temporales cuantitativas.
- Brindar un mecanismo para encontrar errores en el modelo a través de la técnica de mutación de propiedades.
- Generar casos de prueba para testear una implementación.

El caso de estudio analizado en este trabajo muestra una aplicación práctica de nuestra técnica de verificación, destacando su capacidad para identificar y abordar de manera efectiva requisitos temporales cuantitativos complejos en sistemas en tiempo real.

Las contribuciones de la tesis se sustentan en las publicaciones de los trabajos preliminares [41, 42, 43, 44] y los resultados principales de [45].

9.2 Análisis de Escalabilidad

El uso de *model checkers* para la verificación de sistemas de nivel industrial puede generar preocupaciones sobre la aplicabilidad de la técnica presentada en esta tesis. En particular, el llamado problema de explosión de estados podría hacer que nuestra técnica sea impráctica para ciertos problemas del mundo real. No obstante, Uppaal incorpora diversas optimizaciones, como técnicas de reducción de modelos y abstracción basada en zonas [130, 131], que reducen la complejidad de los modelos y mejoran la eficiencia del algoritmo de verificación. Estas técnicas han demostrado su eficacia en la práctica, como lo evidencian múltiples proyectos que han aplicado Uppaal a problemas industriales de tiempo real [103, 132, 133, 134].

Además del caso de estudio presentado en esta tesis, hemos validado nuestro método aplicando el patrón *Time-Bounded Response* a la verificación de dos PTC del Sistema de Control de Cambios de un automóvil (*Gear Control System*), descrito por Lindahl et al. [103]. Los datos experimentales están disponibles en nuestro repositorio de GitHub [101].

El hecho de que varios sistemas de nivel industrial hayan sido verificados con Uppaal respalda la viabilidad de nuestro método y su potencial escalabilidad para abordar problemas más complejos.

9.3 Trabajo Futuro

Actualmente, la implementación de nuestra técnica debe realizarse de forma manual. El desarrollo de una herramienta que automatice el proceso de verificación, detección de errores y generación de casos de prueba es una tarea a considerar en el futuro. Esto es factible mediante el uso de herramientas existentes, complementadas con el desarrollo de soluciones específicas. A continuación, se describen brevemente las actividades a considerar junto con las posibles herramientas que podrían emplearse:

- **Representación de los modelos**: Definir una representación XML (*extensible markup language*) [135] de los modelos RT-DEVS. Tomar como base la representación XML de los DEVS clásicos definida en [41, 42].

- **Construcción de modelos RT-DEVS**: Para el diseño de modelos RT-DEVS, se puede utilizar PowerDevs [46]. Aunque PowerDevs fue diseñada para soportar el formalismo clásico DEVS, su implementación en C++ facilita su adaptación a variantes como STDEVS [136], RTA-DEVS [38], y la variante utilizada en esta tesis, entre otras. Posteriormente, PowerDevs debe extenderse[1] para guardar los modelos en formato XML conforme a la representación definida anteriormente (RT-DEVS-XML).

- **Traducción de RT-DEVS a TA**: Mediante lenguajes de transformación de modelos como QVT [57] o ATL [65], es posible definir reglas de transformación para convertir un modelo RT-DEVS-XML en un modelo XML compatible con los TA admitidos por Uppaal. Esta actividad se enmarca en la metodología MDD [52]. Transformaciones similares ya han sido descritas en los trabajos iniciales de esta tesis [41, 42].

- **Verificación de los patrones y mutantes**: Diseñar una herramienta interactiva que facilite al usuario la configuración y uso de los patrones y mutantes. Esta herramienta debe ser transparente, permitiendo verificar las PTC y sus mutantes tal como se describe en el Capítulo 6. La API de Java proporcionada por Uppaal puede ser empleada para realizar verificaciones desde la línea de comandos.

- **Generación de casos de prueba**: Los resultados obtenidos de las verificaciones deben traducirse a casos de prueba en un formato sencillo de interpretar y ampliamente usado, como por ejemplo el estándar XML. Posteriormente, los desarrolladores deben procesar cada caso de prueba desde sus generadores de entrada para estimular la implementación.

También, es de interés añadir nuevos patrones y mutantes para enriquecer el conjunto de propiedades que se pueden verificar. De esta manera, también podrían verificarse propiedades que aún no son soportadas con nuestra técnica.

Otras variantes del formalismo DEVS podrían verificarse con la misma metodología; más aún, otros formalismos de especificación de sistemas de tiempo real podrían adaptarse a la actual propuesta.

[1] PowerDevs es de distribución libre (bajo licencia GPL)

Referencias

[1] C. Jones, O. Bonsignour, The Economics of Software Quality, Addison-Wesley, 2011. URL http://books.google.com.ar/books?id=_t5l5Cn0NBEC

[2] F. P. Brooks, Jr., The mythical man-month (anniversary ed.), Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[3] L. Bass, P. Clements, R. Kazman, Software Architecture in Practice, 2nd Edition, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[4] H. Krasner, The cost of poor software quality in the us: A 2020 report, Proc. Consortium Inf. Softw. QualityTM (CISQTM) 2 (2021).

[5] M. Daran, P. Thévenod-Fosse, Software error analysis: A real case study involving real faults and mutations, ACM SIGSOFT Software Engineering Notes 21 (3) (1996) 158–171.

[6] J. W. S. W. Liu, Real-Time Systems, 1st Edition, Prentice Hall PTR, USA, 2000.

[7] E. M. Clarke, T. A. Henzinger, H. Veith, R. Bloem (Eds.), Handbook of Model Checking, Springer, 2018. doi:10.1007/978-3-319-10575-8.

[8] G. Behrmann, A. David, K. G. Larsen, A tutorial on Uppaal., in: M. Bernardo, F. Corradini (Eds.), SFM, Vol. 3185 of LNCS, Springer, 2004, pp. 200–236. doi:10.1007/978-3-540-30080-9_7.

[9] G. J. Holzmann, The model checker SPIN, IEEE Trans. Softw. Eng. 23 (5) (1997) 279–295. doi:10.1109/32.588521.

[10] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, A. Tacchella, Nusmv 2: An opensource tool for symbolic model checking, in: E. Brinksma, K. G. Larsen (Eds.), Computer Aided Verification, Springer Berlin Heidelberg, Berlin, Heidelberg, 2002, pp. 359–364.

[11] M. Utting, B. Legeard, Practical Model-Based Testing: A Tools Approach, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2006.

[12] T. Sugeta, J. C. Maldonado, W. E. Wong, Mutation testing applied to validate sdl specifications, in: R. Groz, R. M. Hierons (Eds.), TestCom, Vol. 2978 of LNCS, Springer, 2004, pp. 193–208. doi:10.1007/978-3-540-24704-3_13.

[13] T. J. Schaefer, The complexity of satisfiability problems, in: Proceedings of the Tenth Annual ACM Symposium on Theory of Computing, STOC '78, Association for Computing Machinery, New York, NY, USA, 1978, p. 216?226. doi:10.1145/800133.804350. URL https://doi.org/10.1145/800133.804350

[14] D. Jackson, Software Abstractions: Logic, Language, and Analysis, The MIT Press, 2012.

[15] N. Eén, N. Sörensson, An extensible sat-solver, in: E. Giunchiglia, A. Tacchella (Eds.), Theory and Applications of Satisfiability Testing, 6th International Conference, SAT 2003. Santa Margherita Ligure, Italy, May 5-8, 2003 Selected Revised Papers, Vol. 2919 of Lecture Notes in Computer Science, Springer, 2003, pp. 502–518. doi:10.1007/978-3-540-24605-3_37.
URL https://doi.org/10.1007/978-3-540-24605-3_37

[16] The Coq development team, The Coq proof assistant reference manual, Version 8.0 (2004).
URL /brokenurl#{http://coq.inria.fr}

[17] T. Nipkow, L. C. Paulson, M. Wenzel, Isabelle/HOL: a proof assistant for higher-order logic, Vol. 2283, Springer Science & Business Media, 2002.

[18] B. P. Douglass, Real-Time UML: Developing Efficient Objects for Embedded Systems, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.

[19] H. S. Song, T. G. Kim, Application of Real-Time DEVS to analysis of safety-critical embedded control systems: Railroad crossing control example, SIMULATION 81 (2) (2005) 119–136. doi:10.1177/0037549705052229.

[20] A. Furfaro, L. Nigro, Embedded control systems design based on RT-DEVS and temporal analysis using Uppaal, in: 2008 International Multiconference on Computer Science and Information Technology, 2008, pp. 601–608. doi:10.1109/IMCSIT.2008.4747305.

[21] H. Saadawi, G. Wainer, M. Moallemi, Principles of DEVS Model Verification for Real-Time Embedded Applications, 2012, pp. 63–96. doi:10.1201/b12667.

[22] D. B. Stewart, Twenty-five most common mistakes with real-time software development, 1999.
URL https://api.semanticscholar.org/CorpusID:2764888

[23] I. Suzuki, H. Lu, Temporal Petri nets and their application to modeling and analysis of a handshake daisy chain arbiter, IEEE Transactions on Computers 38 (5) (1989) 696–704. doi:10.1109/12.24271.

[24] H. G. Molter, Discrete event system specification, in: SynDEVS Co-Design Flow, Springer Fachmedien Wiesbaden, 2012, pp. 9–42. doi:10.1007/978-3-658-00397-5.

[25] R. Alur, D. L. Dill, A theory of timed automata, Theoretical Computer Science 126 (2) (1994) 183–235. doi:10.1016/0304-3975(94)90010-8.

[26] J. S. Hong, H.-S. Song, T. G. Kim, K. H. Park, A real-time discrete event system specification formalism for seamless real-time software development, Discrete Event Dynamic Systems 7 (4) (1997) 355–375. doi:10.1023/A:1008262409521.

[27] L. Lamport, Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers, Addison-Wesley, 2002.
URL http://research.microsoft.com/users/lamport/tla/book.html

[28] A. Pnueli, The temporal logic of programs, in: 18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977, IEEE Computer Society, 1977, pp. 46–57. doi:10.1109/SFCS.1977.32.

[29] E. M. Clarke, E. A. Emerson, A. P. Sistla, Automatic verification of finite-state concurrent systems using temporal logic specifications, ACM Trans. Program. Lang. Syst. 8 (2) (1986) 244–263. doi:10.1145/5397.5399.

[30] A. Cimatti, E. M. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, A. Tacchella, Nusmv 2: An opensource tool for symbolic model checking, in: Proceedings of the 14th International Conference on Computer Aided Verification, CAV '02, Springer-Verlag, Berlin, Heidelberg, 2002, pp. 359–364.

[31] R. Alur, C. Courcoubetis, D. L. Dill, Model-checking for real-time systems, in: Proceedings of the Fifth Annual Symposium on Logic in Computer Science (LICS '90), Philadelphia, Pennsylvania, USA, June 4-7, 1990, IEEE Computer Society, 1990, pp. 414–425. doi:10.1109/LICS.1990.113766.

[32] R. Koymans, Specifying real-time properties with metric temporal logic, Real-Time Systems 2 (4) (1990) 255–299. doi:10.1007/BF01995674.

[33] Y. Van Tendeloo, H. Vangheluwe, An evaluation of devs simulation tools, Simulation 93 (2) (2017) 103–121. doi:10.1177/0037549716678330.

[34] E. M. Clarke, W. Klieber, M. Nováček, P. Zuliani, Model Checking and the State Explosion Problem, Springer Berlin Heidelberg, Berlin, Heidelberg, 2012, pp. 1–30. doi:10.1007/978-3-642-35746-6_1.

[35] A. Mekki, M. Ghazel, A. Toguyeni, Validating time-constrained systems using UML Statecharts patterns and timed automata observers, 2009. doi:10.14236/ewic/VECOS2009.11.

[36] J. D. Backes, M. W. Whalen, A. Gacek, J. Komp, On implementing real-time specification patterns using observers, in: S. Rayadurgam, O. Tkachuk (Eds.), NASA Formal Methods, Springer International Publishing, Cham, 2016, pp. 19–33. doi:10.1007/978-3-319-40648-0_2.

[37] H. P. Dacharry, N. Giambiasi, A formal verification approach for DEVS, in: Proceedings of the 2007 Summer Computer Simulation Conference, SCSC '07, Society for Computer Simulation International, San Diego, CA, USA, 2007, pp. 312–319.
URL https://dl.acm.org/doi/abs/10.5555/1357910.1357960

[38] H. Saadawi, G. Wainer, Principles of discrete event system specification model verification, Simulation 89 (1) (2013) 41–67. doi:10.1177/0037549711424424.

[39] R. Alur, Techniques for automatic verification of real-time systems, Ph.D. thesis, Stanford, CA, USA, uMI Order No. GAX92-06729 (1992).

[40] S. Konrad, B. H. C. Cheng, Real-time specification patterns, in: Proceedings of the 27th International Conference on Software Engineering, ICSE '05, ACM, New York, NY, USA, 2005, pp. 372–381. doi:10.1145/1062455.1062526.
URL http://doi.acm.org/10.1145/1062455.1062526

[41] A. Gonzalez, C. Luna, M. Daniele, R. Cuello, M. Perez, Towards an automatic model transformation mechanism from UML state machines to DEVS models, CLEI Electron. J. 18 (2) (2015). doi:10.19153/cleiej.18.2.3.

[42] A. Gonzalez, C. D. Luna, R. Abella, UML state machine as modeling language for DEVS formalism, in: XLII Latin American Computing Conference, CLEI 2016, Valparaíso, Chile, October 10-14, 2016, IEEE, 2016, pp. 1–12. doi:10.1109/CLEI.2016.7833350.

[43] A. Gonzalez, M. Cristiá, C. Luna, Mutants for metric temporal logic formulas, in: Proceedings of the XXII Iberoamerican Conference on Software Engineering, CIbSE 2019, La Habana, Cuba, April 22-26, 2019, Curran Associates, 2019, pp. 349–362.
URL https://api.semanticscholar.org/CorpusID:199465702

[44] A. Gonzalez, M. Cristiá, C. Luna, Error finding in real-time systems using mutants of temporal properties, in: 2021 40th International Conference of the Chilean Computer Science Society (SCCC), 2021, pp. 1–8. doi:10.1109/SCCC54552.2021.9650361.

[45] A. González, M. Cristiá, C. Luna, Verification of quantitative temporal properties in realtime-devs, Simulation: Transactions of the Society for Modeling and Simulation International (2025) pp. 1–20.doi:10.1177/00375497251340070.

[46] F. Bergero, E. Kofman, Powerdevs: A tool for hybrid system modeling and real-time simulation, Simulation 87 (1-2) (2011) 113–132.
URL http://dx.doi.org/10.1177/0037549710368029

[47] A. Zengin, H. Sarjoughian, Devs-suite simulator: a tool teaching network protocols, in: Proceedings of the Winter Simulation Conference, WSC '10, Winter Simulation Conference, 2010, p. 2947?2957.

[48] T. G. Kim, DEVSim++ Users Manual. C++ Based Simulation with Hierarchical Modular DEVS Models, Korea Advance Institute of Science and Technology (1994).

[49] Object management group (Last access: April 2025).
URL http://www.omg.org

[50] D. A. Hollmann, M. Cristiá, C. Frydman, Cml-devs: A specification language for devs conceptual models, Simulation Modelling Practice and Theory 57 (2015) 100–117. doi:https://doi.org/10.1016/j.simpat.2015.06.007.
URL https://www.sciencedirect.com/science/article/pii/S1569190X15001021

[51] G. Wainer, Q. Liu, Tools for graphical specification and visualization of devs models, Simulation 85 (3) (2009) 131?158. doi:10.1177/0037549708101182.
URL https://doi.org/10.1177/0037549708101182

[52] B. Selic, The pragmatics of model-driven development, IEEE Softw. 20 (5) (2003) 19–25.

[53] F. Dalmasso, M. J. Blas, S. Gonnet, Enriching uml statecharts through a metamodel: A model driven approach for the graphical definition of devs atomic models, IEEE Latin America Transactions 21 (1) (2023) 27–34. doi:10.1109/TLA.2023.10015142.

[54] Q. Lin, J. Yang, H. Zhang, A devs-based formal description method for complex product behavior models, Journal of System Simulation 34 (4) (2022) 661–669.

[55] P. A. Noreña C., C. M. Zapata J., Simulating events in requirements engineering by using pre-conceptual-schema-based components from scientific software domain representation, Advances in Systems Science and Applications 21 (4) (2022) 1?15. doi:10.25728/assa.2021.21.4.1104.
URL https://ijassa.ipu.ru/index.php/ijassa/article/view/1104

[56] B. P. Zeigler, T. G. Kim, H. Praehofer, Theory of Modeling and Simulation, 2nd Edition, Academic Press, Inc., Orlando, FL, USA, 2000.

[57] OMG, Meta object facility (mof) 2.0 query/view/transformation specification, version 1.1 (2011).
URL http://www.omg.org/spec/QVT/1.1/

[58] M. Kay, SAXON. The XSLT and XQuery Processor, Saxonica (Last access: April 2025).
URL https://www.saxonica.com/

[59] D. Steinberg, F. Budinsky, M. Paternostro, E. Merks, EMF: Eclipse Modeling Framework 2.0, 2nd Edition, Addison-Wesley Professional, 2009.

[60] Eclipse, Acceleo (2012).
URL http://www.eclipse.org/acceleo/

[61] S. Gérard, C. Dumoulin, P. Tessier, B. Selic, Papyrus: A uml2 tool for domain-specific language modeling, in: Proceedings of the 2007 International Dagstuhl Conference on Model-based Engineering of Embedded Real-time Systems, MBEERTS'07, Springer-Verlag, Berlin, Heidelberg, 2010, pp. 361–368.
URL http://dl.acm.org/citation.cfm?id=1927558.1927582

[62] S. Yovine, Kronos: a verification tool for real-time systems, International Journal on Software Tools for Technology Transfer 1 (1997) 123–133.

[63] T. A. Henzinger, P.-H. Ho, H. Wong-Toi, Hytech: a model checker for hybrid systems, Int. J. Softw. Tools Technol. Transf. 1 (1-2) (1997) 110–122. doi:10.1007/s100090050008.
URL https://doi.org/10.1007/s100090050008

[64] K. G. Samuel, N.-D. M. Bouare, O. Maiga, M. K. Traoré, A devs-based pivotal modeling formalism and its verification and validation framework, SIMULATION 96 (12) (2020) 969–992. arXiv:https://doi.org/10.1177/0037549720958056, doi:10.1177/0037549720958056.

[65] ATL Transformation Language (Last access: April 2025).
URL http://www.eclipse.org/atl

[66] M. B. Dwyer, G. S. Avrunin, J. C. Corbett, Patterns in property specifications for finite-state verification., in: B. W. Boehm, D. Garlan, J. Kramer (Eds.), ICSE, ACM, 1999, pp. 411–420.
URL http://dblp.uni-trier.de/db/conf/icse/icse99.html#DwyerAC99

[67] S. Boukelkoul, R. Maamri, M. Chihoub, A discrete event model for analysis and verification of time-constrained business processes, Concurrency and Computation: Practice and Experience 33 (1) (2021) e5753. arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/cpe.5753, doi:https://doi.org/10.1002/cpe.5753.
URL https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.5753

[68] A. Yacoub, M. E. A. Hamri, C. Frydman, Dev-promela: modeling, verification, and validation of a video game by combining model-checking and simulation, Simulation 96 (11) (2020) 881?910. doi:10.1177/0037549720946107.
URL https://doi.org/10.1177/0037549720946107

[69] B. P. Zeigler, J. J. Nutaro, C. Seo, Combining devs and model-checking: concepts and tools for integrating simulation and analysis, International Journal of Simulation and Process Modelling 12 (1) (2017) 2–15. arXiv:https://www.inderscienceonline.com/doi/pdf/10.1504/IJSPM.2017.082781, doi:10.1504/IJSPM.2017.082781.

[70] M. H. Hwang, B. P. Zeigler, Reachability graph of finite and deterministic devs networks, IEEE Transactions on Automation Science and Engineering 6 (3) (2009) 468–478. doi:10.1109/TASE.2009.2021352.

[71] C. Seo, B. P. Zeigler, R. Coop, D. Kim, Devs modeling and simulation methodology with ms4 me software tool, in: Proceedings of the Symposium on Theory of Modeling & Simulation - DEVS Integrative M&S Symposium, DEVS 13, Society for Computer Simulation International, San Diego, CA, USA, 2013.

[72] S. Gholami, H. S. Sarjoughian, Modeling and verification of network-on-chip using constrained-DEVS, in: Proceedings of the Symposium on Theory of Modeling & Simulation, TMS/DEVS '17, Society for Computer Simulation International, San Diego, CA, USA, 2017.

[73] A. Inostrosa-Psijas, M. Oyarzún-Silva, F. Medina-Quispe, F. García-Barrera, R. Solar-Gallardo, Verificación formal de un modelo de simulación DEVS de una aplicación Storm, Ingeniare. Revista chilena de ingeniería 27 (2019) 682 – 695.

[74] B. Zeigler, H. Praehofer, T. Kim, Theory of Modeling and Simulation: Integrating Discrete Event and Continuous Complex Dynamic Systems, Academic Press, 2000.
URL http://books.google.com.uy/books?id=REzmYOQmHuQC

[75] T. Murata, Petri nets: Properties, analysis and applications., Proceedings of the IEEE 77 (4) (1989) 541–580.

[76] D. Harel, M. Politi, Modeling Reactive Systems with Statecharts: The Statemate Approach, 1st Edition, McGraw-Hill, Inc., New York, NY, USA, 1998.

[77] A. H. Buss, Modeling with event graphs, in: Proceedings of the 28th Conference on Winter Simulation, WSC '96, IEEE Computer Society, USA, 1996, p. 153?160. doi:10.1145/256562.256597.

[78] B. P. Zeigler, S. Vahie, Devs formalism and methodology: Unity of conception/diversity of application, in: Proceedings of the 25th Conference on Winter Simulation, WSC '93, ACM, New York, NY, USA, 1993, pp. 573–579. doi:10.1145/256563.256724.

[79] G. Wainer, Cd++: A toolkit to develop devs models, Softw. Pract. Exper. 32 (13) (2002) 1261–1306.
URL http://dx.doi.org/10.1002/spe.482

[80] G. A. Wainer, R. Goldstein, A. Khan, Introduction to the discrete event system specification formalism and its application for modeling and simulating cyber-physical systems, in: 2018 Winter Simulation Conference (WSC), 2018, pp. 177–191. doi:10.1109/WSC.2018.8632408.

[81] J. Bengtsson, W. Yi, Timed Automata: Semantics, Algorithms and Tools, Springer Berlin Heidelberg, Berlin, Heidelberg, 2004, pp. 87–124. doi:10.1007/978-3-540-27755-2_3.

[82] J. Ouaknine, J. Worrell, On the decidability and complexity of metric temporal logic over finite words, Logical Methods in Computer Science Volume 3, Issue 1 (Feb. 2007). doi:10.2168/LMCS-3(1:8)2007.
URL https://lmcs.episciences.org/2230

[83] W. Visser, P. Mehlitz, Model checking programs with java pathfinder, in: Proceedings of the 12th International Conference on Model Checking Software, SPIN'05, Springer-Verlag, Berlin, Heidelberg, 2005, p. 27. doi:10.1007/11537328_5.
URL https://doi.org/10.1007/11537328_5

[84] Y. Yu, P. Manolios, L. Lamport, Model checking tla+ specifications, in: International Conference on Correct Hardware Design and Verification Methods, Springer, 1999, pp. 54–66.

[85] E. Omar, S. Ghosh, D. Whitley, Homaj: A tool for higher order mutation testing in aspectj and java, in: Proceedings of the 5th International Workshop on Mutation Analysis (MUTATION'14), Cleveland, USA, 2014.

[86] R. A. DeMillo, R. J. Lipton, F. G. Sayward, Hints on test data selection: Help for the practicing programmer, Computer 11 (4) (1978) 34–41.
URL http://dx.doi.org/10.1109/C-M.1978.218136

[87] M. Delamaro, J. Maidonado, A. Mathur, Interface mutation: an approach for integration testing, IEEE Transactions on Software Engineering 27 (3) (2001) 228–247. doi:10.1109/32.910859.

[88] S. Ghosh, A. P. Mathur, Interface mutation, Software Testing, Verification and Reliability 11 (4) (2001) 227–247. arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/stvr.239, doi:https://doi.org/10.1002/stvr.239.
URL https://onlinelibrary.wiley.com/doi/abs/10.1002/stvr.239

[89] A. J. Offutt, Investigations of the software testing coupling effect, ACM Trans. Softw. Eng. Methodol. 1 (1) (1992) 5–20.
URL http://doi.acm.org/10.1145/125489.125473

[90] S. C. P. F. Fabbri, J. C. Maldonado, T. Sugeta, P. C. Masiero, Mutation testing applied to validate specifications based on statecharts, in: Proceedings 10th International Symposium on Software Reliability Engineering (Cat. No. PR00443), IEEE, 1999, pp. 210–219.

[91] S. C. P. F. Fabbri, J. C. Maldonado, P. C. Masiero, M. E. Delamaro, E. Wong, Mutation testing applied to validate specifications based on petri nets, in: Formal Description Techniques VIII: Proceedings of the IFIP TC6 Eighth International Conference on Formal Description Techniques, Montreal, Canada, October 1995, Springer, 1996, pp. 329–337.

[92] S. d. R. Senger de Souza, S. C. P. F. Fabbri, W. Lopes de Souza, J. C. Maldonado, Mutation testing applied to estelle specifications, in: Proceedings of the 33rd Hawaii International Conference on System Sciences-Volume 8-Volume 8, 2000, p. 8011.

[93] B. K. Aichernig, F. Lorber, D. Nickovic, Model-based mutation testing with timed automata, Graz University of Technology, Graz (2013).

[94] B. K. Aichernig, F. Lorber, D. Ničković, Time for mutants – model-based mutation testing with timed automata, in: Tests and Proofs: 7th International Conference, TAP 2013, Budapest, Hungary, June 16-20, 2013. Proceedings 7, Springer, 2013, pp. 20–38.

[95] A. Khalilov, T. Tuglular, F. Belli, Mutation operators for decision table-based contracts used in software testing, in: 2020 Turkish National Software Engineering Symposium (UYMS), 2020, pp. 1–6. doi:10.1109/UYMS50627.2020.9247061.

[96] M. Büchler, J. Oudinet, A. Pretschner, Security mutants for property-based testing., in: M. Gogolla, B. Wolff (Eds.), TAP, Vol. 6706 of LNCS, Springer, 2011, pp. 69–77. doi:10.1007/978-3-642-21768-5_6.

[97] L. Tan, O. Sokolsky, I. Lee, Specification-based testing with linear temporal logic, in: Proceedings of the 2004 IEEE International Conference on Information Reuse and Integration, 2004. IRI 2004., 2004, pp. 493–498. doi:10.1109/IRI.2004.1431509.

[98] M. B. Trakhtenbrot, Mutation patterns for temporal requirements of reactive systems., in: ICST Workshops, IEEE Computer Society, 2017, pp. 116–121. doi:10.1109/ICSTW.2017.27.

[99] A. Armando, D. Basin, Y. Boichut, Y. Chevalier, L. Compagna, J. Cuellar, P. H. Drielsma, P. C. Heám, O. Kouchnarenko, J. Mantovani, S. Mödersheim, D. von Oheimb, M. Rusinowitch, J. Santiago, M. Turuani, L. Viganò, L. Vigneron, The avispa tool for the automated validation of internet security protocols and applications, CAV'05, Springer-Verlag, Berlin, Heidelberg, 2005, p. 281?285. doi:10.1007/11513988_27.
URL https://doi.org/10.1007/11513988_27

[100] H. Saadawi, G. Wainer, Verification of real-time devs models, in: Proceedings of the 2009 Spring Simulation Multiconference, SpringSim '09, Society for Computer Simulation International, San Diego, CA, USA, 2009.

[101] A. González, Experimental data for verification of quantitative temporal properties in RealTime-DEVS with Uppaal (2024).
URL https://github.com/agonzalez2020/Train-Controller-Alarm-Rtdevs

[102] F. Boniol, V. Wiels, The landing gear system case study, in: F. Boniol, V. Wiels, Y. Ait Ameur, K.-D. Schewe (Eds.), ABZ 2014: The Landing Gear Case Study, Springer International Publishing, Cham, 2014, pp. 1–18.

[103] M. Lindahl, P. Pettersson, W. Yi, Formal design and analysis of a gear controller, in: Proceedings of the 4th International Conference on Tools and Algorithms for Construction and Analysis of Systems, TACAS '98, Springer-Verlag, Berlin, Heidelberg, 1998, pp. 281–297.

[104] J. Ouaknine, J. Worrell, Some recent results in metric temporal logic, in: Proceedings of the 6th International Conference on Formal Modeling and Analysis of Timed Systems, FORMATS '08, Springer-Verlag, Berlin, Heidelberg, 2008, pp. 1–13.
URL http://dx.doi.org/10.1007/978-3-540-85778-51

[105] V. Gruhn, R. Laue, Patterns for timed property specifications., Electr. Notes Theor. Comput. Sci. 153 (2) (2006) 117–133. doi:10.1016/j.entcs.2005.10.035.

[106] N. Abid, S. Dal Zilio, D. Le Botlan, Real-Time Specification Patterns and Tools, Springer Berlin Heidelberg, 2012, pp. 1–15. doi:10.1007/978-3-642-32469-7_1.

[107] J. Ouaknine, J. Worrell, Safety metric temporal logic is fully decidable, in: H. Hermanns, J. Palsberg (Eds.), Tools and Algorithms for the Construction and Analysis of Systems, Springer Berlin Heidelberg, Berlin, Heidelberg, 2006, pp. 411–425.

[108] L. Aceto, A. Burgueño, K. G. Larsen, Model checking via reachability testing for timed automata., in: B. Steffen (Ed.), TACAS, Vol. 1384 of Lecture Notes in Computer Science, Springer, 1998, pp. 263–280. doi:10.1007/BFb0054177.

[109] I. Beer, S. Ben-David, H. Chockler, A. Orni, R. Trefler, Explaining counterexamples using causality, in: A. Bouajjani, O. Maler (Eds.), Computer Aided Verification, Springer Berlin Heidelberg, Berlin, Heidelberg, 2009, pp. 94–108. doi:10.1007/978-3-642-02658-4_11.

[110] H. Debbi, Counterexamples in model checking - a survey, Informatica (Slovenia) 42 (2018). URL https://api.semanticscholar.org/CorpusID:51874797

[111] A. P. Kaleeswaran, A. Nordmann, T. Vogel, L. Grunske, A systematic literature review on counterexample explanation, Information and Software Technology 145 (2022) 106800. doi:10.1016/j.infsof.2021.106800.

[112] J. A. Clark, H. Dan, R. M. Hierons, Semantic mutation testing, Science of Computer Programming 78 (4) (2013) 345 – 363, special section on Mutation Testing and Analysis (Mutation 2010) - Special section on the Programming Languages track at the 25th ACM Symposium on Applied Computing.
URL http://www.sciencedirect.com/science/article/pii/S0167642311000992

[113] G. Di Lucca, A. Fasolino, F. Faralli, U. De Carlini, Testing web applications, in: International Conference on Software Maintenance, 2002. Proceedings., 2002, pp. 310–319. doi:10.1109/ICSM.2002.1167787.

[114] H. Reza, K. Ogaard, A. Malge, A model based testing technique to test web applications using statecharts, in: Fifth International Conference on Information Technology: New Generations (itng 2008), 2008, pp. 183–188. doi:10.1109/ITNG.2008.145.

[115] I. K. El-Far, J. A. Whittaker, Model-based software testing, Encyclopedia of software engineering (2002).

[116] C.-H. Liu, D. C. Kung, P. Hsia, Object-based data flow testing of web applications, in: Proceedings First Asia-Pacific Conference on Quality Software, IEEE, 2000, pp. 7–16.

[117] O. Dahl, E. Dijkstra, C. Hoare, Structured programming, APIC studies in data processing, Academic Press Inc., United States, 1972.

[118] P. Stocks, D. A. Carrington, Test template framework: A specification-based testing case study, in: T. J. Ostrand, E. J. Weyuker (Eds.), Proceedings of the 1993 International Symposium on Software Testing and Analysis, ISSTA 1993, Cambridge, MA, USA, June 28-30, 1993, ACM, 1993, pp. 11–18. doi:10.1145/154183.154190.

[119] K. Henares, J. L. Risco-Martín, J. L. Ayala, R. Hermida, Unit testing platform to verify devs models, in: Proceedings of the 2020 Summer Simulation Conference, SummerSim '20, Society for Computer Simulation International, San Diego, CA, USA, 2020.

[120] N. Kosindrdecha, J. Daengdej, A test case generation process and technique, journal of Software Engineering 4 (4) (2010) 265–287.

[121] Y. Labiche, M. Shafique, A systematic review of model based testing tool support (2010).

[122] S. Rayadurgam, M. P. E. Heimdahl, Coverage based test-case generation using model checkers, in: Proceedings. Eighth Annual IEEE International Conference and Workshop On the Engineering of Computer-Based Systems-ECBS 2001, IEEE, 2001, pp. 83–91.

[123] A. Arrieta, J. A. Agirre, G. Sagardui, A tool for the automatic generation of test cases and oracles for simulation models based on functional requirements, in: 2020 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW), IEEE, 2020, pp. 1–5.

[124] A. Arrieta, S. Wang, A. Arruabarrena, U. Markiegi, G. Sagardui, L. Etxeberria, Multi-objective black-box test case selection for cost-effectively testing simulation models, in: Proceedings of the Genetic and Evolutionary Computation Conference, GECCO '18, Association for Computing Machinery, New York, NY, USA, 2018, p. 1411?1418. doi:10.1145/3205455.3205490.
URL https://doi.org/10.1145/3205455.3205490

[125] L. Lima, J. J. Huerta y Munive, D. Traytel, Explainable online monitoring of metric first-order temporal logic, in: International Conference on Tools and Algorithms for the Construction and Analysis of Systems, Springer, 2024, pp. 288–307.

[126] H.-M. Ho, J. Ouaknine, J. Worrell, Online monitoring of metric temporal logic, in: B. Bonakdarpour, S. A. Smolka (Eds.), Runtime Verification, Springer International Publishing, Cham, 2014, pp. 178–192.

[127] H. Gunadi, A. Tiu, Efficient runtime monitoring with metric temporal logic: A case study in the android operating system, ArXiv abs/1311.2362 (2013).
URL https://api.semanticscholar.org/CorpusID:17902855

[128] M. Hendriks, M. Geilen, A. R. B. Behrouzian, T. Basten, H. A. Ara, D. Goswami, Checking metric temporal logic with trace, 2016 16th International Conference on Application of Concurrency to System Design (ACSD) (2016) 19–24.
URL https://api.semanticscholar.org/CorpusID:16744520

[129] D. Bianculli, C. Ghezzi, S. Krstic, P. S. Pietro, Offline trace checking of quantitative properties of service-based applications, in: 2014 IEEE 7th International Conference on Service-Oriented Computing and Applications, 2014, pp. 9–16. doi:10.1109/SOCA.2014.14.

[130] P. Bouyer, P. Gastin, F. Herbreteau, O. Sankur, B. Srivathsan, Zone-based verification of timed automata: Extrapolations, simulations and what next?, in: Formal Modeling and Analysis of Timed Systems: 20th International Conference, FORMATS 2022, Warsaw, Poland, September 13?15, 2022, Proceedings, Springer-Verlag, Berlin, Heidelberg, 2022, p. 16?42. doi:10.1007/978-3-031-15839-1_2.
URL https://doi.org/10.1007/978-3-031-15839-1_2

[131] G. Behrmann, J. Bengtsson, A. David, K. G. Larsen, P. Pettersson, W. Yi, Uppaal implementation secrets, in: Formal Techniques in Real-Time and Fault-Tolerant Systems, Springer Berlin Heidelberg, Berlin, Heidelberg, 2002, pp. 3–22.

[132] H. Bowman, G. Faconti, J.-P. Katoen, D. Latella, M. Massink, Automatic verification of a lip-synchronisation protocol using uppaal, Form. Asp. Comput. 10 (5) (1998) 550?575. doi:10.1007/s001650050032.
URL https://doi.org/10.1007/s001650050032

[133] A. P. Ravn, J. Srba, S. Vighio, Modelling and verification of web services business activity protocol, TACAS'11/ETAPS'11, Springer-Verlag, Berlin, Heidelberg, 2011, p. 357?371.

[134] Z. Bakhshi, G. Rodriguez-Navas, H. Hansson, Using uppaal to verify recovery in a fault-tolerant mechanism providing persistent state at the edge, in: 2021 26th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA), IEEE Press, 2021, p. 1?6. doi:10.1109/ETFA45728.2021.9613178.
URL https://doi.org/10.1109/ETFA45728.2021.9613178

[135] toExcel, Extensible Markup Language (Xml) 1.0 Specifications: From the W3c Recommendations, iUniverse, Incorporated, 2000.

[136] R. Castro, E. Kofman, G. Wainer, A formal framework for stochastic devs modeling and simulation, in: Proceedings of the 2008 Spring Simulation Multiconference, SpringSim '08, Society for Computer Simulation International, San Diego, CA, USA, 2008, p. 421?428.

Appendix A

Definición matemática del modelo RT-DEVS del SCF

Esta sección presenta la versión matemática de los modelos RT-DEVS *Train* y *Alarm* representados en la Figura 5.1. Para ello, consideramos la siguiente versión simplificada de RT-DEVS:

$$\textbf{RT-DEVS Train} = \langle X, Y, S, \delta_{ext}, \delta_{int}, \lambda, ti \rangle$$

$X = \{stop, go\} \times \mathbb{N}$

$Y = (\{appr, leave\} \times \mathbb{N}) \cup (\{alarm\} \times \{\tau\}) \cup \phi$
donde τ representa una señal de alarma y ϕ una salida "ficticia". Dado que RT-DEVS requiere una salida cada vez que se activa una transición interna, usamos ϕ cuando esta salida no es recibida por otro componente RT-DEVS.

$S = \{Safe, Talarm, Appr, Stop, Start, Cross\}$

$$\delta_{ext}((s,e),(p,v)) = \begin{cases} Stop, & \text{if } s = Appr \wedge p = stop \wedge\ e \leq 10 \\ Start, & \text{if } s = Stop \wedge p = go \end{cases}$$

$$\delta_{int}(s) = \begin{cases} Talarm, & \text{if } s = Safe \wedge input\_enabled \\ Appr, & \text{if } s = Talarm \\ Cross, & \text{if } s = Appr \vee s = Start \\ Safe, & \text{if } s = Cross \end{cases}$$

$$\lambda(s) = \begin{cases} (appr, id), & \text{if } s = Safe \\ (alarm, \tau), & \text{if } s = Talarm \\ (leave, id), & \text{if } s = Cross \\ \phi, & \text{Otherwise} \end{cases}$$

donde id es el identificador de un tren.

$$ti(s) = \begin{cases} [0,\infty] & \text{if } s = Safe \\ [0,0], & \text{if } s = Talarm \\ [10,20], & \text{if } s = Appr \\ [\infty,\infty], & \text{if } s = Stop \\ [7,15], & \text{if } s = Start \\ [3,5], & \text{if } s = Cross \end{cases}$$

$[\infty, \infty]$ representa estados pasivos, es decir, el sistema solo puede abandonarlo si ocurre una entrada.

$$\textbf{RT-DEVS Alarm} = \langle X, Y, S, \delta_{ext}, \delta_{int}, \lambda, ti \rangle$$

$X = \{alarm\} \times \{\tau\}$

$Y = (\{alarm\_on, alarm\_off\} \times \{\tau\}) \cup \phi$

$S = \{Off, Warning, Danger, Howl, On\_Off\}$

$\delta_{ext}((s,e),(p,v)) = Warning, \text{if } s = Off \wedge p = alarm \wedge trains\_in\_system \geq 3$

$$\delta_{int}(s) = \begin{cases} Danger, & \text{if } s = Warning \wedge e \geq 2 \\ Off, & \text{if } s = Warning \wedge trains\_in\_system \leq 3 \\ Howl, & \text{if } s = Danger \\ On\_Off, & \text{if } s = Howl \wedge (e == 3 \ \vee trains\_in\_system < 3) \\ Off, & \text{if } s = On\_Off \wedge !decompressing\_area \end{cases}$$

$$\lambda(s) = \begin{cases} (alarm\_on, \tau), & \text{if } s = Warning \wedge e \geq 2 \\ (alarm\_off, \tau), & \text{if } s = On\_Off \\ \phi, & \text{Otherwise} \end{cases}$$

$$ti(s) = \begin{cases} [\infty,\infty] & \text{if } s = Off \\ [0,7], & \text{if } s = Warning \\ [7,7], & \text{if } s = Danger \\ [0,3], & \text{if } s = Howl \\ [\infty,\infty], & \text{if } s = On\_Off \end{cases}$$

input_enabled y *decompressing_area* son variables booleanas compartidas entre ambos modelos. *input_enabled* modela el semáforo; *decompressing_area* indica si el área de cruce se está vaciando; *trains_in_system* representa el número de trenes en el área de cruce.

Acoplamiento mediante puertos del SCF

$$\textbf{RT-DEVS SCF} = \langle X, Y, D, M_d, EIC, EOC, IC, Select \rangle$$

X es el conjunto de eventos de entrada del modelo acoplado:
$X = \emptyset$

Y es el conjunto de eventos de salida del modelo acoplado:
$Y = \{alarm\_on, alarm\_off\}$

D es el conjunto de referencias a los componentes de M_d:
$D = \{d_{RT-DEVS\ Train}, d_{RT-DEVS\ Alarm}, d_{RT-DEVS\ Railroad-Controller}\}$

M_d es el conjunto de modelos RT-DEVS contenidos en el acoplado:
$M_d = \{RT-DEVS\ Train, RT-DEVS\ Alarm, RT-DEVS\ Railroad-Controller\}$

EIC es el conjunto de conexiones desde las entradas del RT-DEVS SCF hacia las entradas de los modelos de M_d:
$EIC = \emptyset$

EOC es el conjunto de conexiones desde las salidas de los M_d hacia las salidas del RT-DEVS SCF:
$EOC = \{[(d_{RT-DEVS\ Alarm}, alarm\_on), (d_{RT-DEVS\ SCF}, alarm\_on)],$
$[(d_{RT-DEVS\ Alarm}, alarm\_off), (d_{RT-DEVS\ SCF}, alarm\_off)]\}$

IC es el conjunto de conexiones desde las salidas de los M_d hacia las entradas de los M_d:
$IC = \{[(d_{RT-DEVS\ Train}, alarm), (d_{RT-DEVS\ Alarm}, alarm)],$
$[(d_{RT-DEVS\ Train}, appr, d_{RT-DEVS\ Railroad-Controller}, appr)],$
$[(d_{RT-DEVS\ Train}, leave, d_{RT-DEVS\ Railroad-Controller}, leave)],$
$[(d_{RT-DEVS\ Railroad-Controller}, stop, d_{RT-DEVS\ Train}, stop)],$
$[(d_{RT-DEVS\ Railroad-Controller}, go, d_{RT-DEVS\ Train}, go)]\}$

$Select$ es la lista de prioridades (de mayor a menor) en caso de eventos simultáneos:
$Select = [RT-DEVS\ Railroad-Controller, RT-DEVS\ Train, RT-DEVS\ Alarm]$