Técnicas Automáticas para la Elaboración, Validación y Verificación de Requisitos de Software

por Renzo Gastón Degiovanni

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

Mayo, 2015
@FaMAF - UNC 2015
Director: Nazareno Matías Aguirre

Resumen

La *Ingeniería de Software* tiene como objetivo brindar mecanismos para crear, diseñar y mantener, de forma sistemática y efectiva, software de calidad y de gran escala. Numerosos estudios han mostrado que muchos de los errores manifestados en el software se deben a inconsistencias, errores de comprensión e imprecisiones durante las etapas tempranas de los procesos de desarrollo de software, en particular durante la captura, elaboración y especificación de requisitos. Más aún, los errores son más costosos y difíciles de resolver a medida que el proceso de desarrollo avanza. Luego, obtener una especificación de requisitos de buena calidad es de fundamental importancia práctica y económica en la mayoría de las metodologías de desarrollo de software modernas.

Es por esta razón que en las últimas décadas los lenguajes y metodologías *formales* para la ingeniería de requisitos (como los statecharts, SCR y notaciones tabulares, el método KAOS y Clafer) han recibido mayor atención. Típicamente, estas metodologías formales de requisitos tienen asociados poderosos mecanismos de análisis que permiten evaluar y mejorar la calidad de las especificaciones de requisitos, tales como análisis de *consistencia* y *completitud*.

Por otra parte, en la actualidad existen varios algoritmos que eficientemente pueden resolver diferentes tipos de análisis sobre formalismos lógicos, como el problema de satisfactibilidad booleana (SAT Solving) o el cálculo de interpolantes. Estos mecanismos se caracterizan por su eficiencia y relativa escalabilidad, que han resultado en diversas aplicaciones prácticas, con una escala sorprendente. Se han utilizado exitosamente en diversas áreas, para atacar problemas como la verificación de software, el testing y planning en Inteligencia Artificial.

En esta tesis, presentaremos varias técnicas automáticas aplicadas a tareas ligadas a la *elaboración*, *validación* y *verificación* de requisitos de software. En breve, éstas se basan en manipulaciones de fórmulas lógicas, explotando la eficiencia de los mecanismos de análisis provenientes de los métodos formales, como SAT Solving, model checking e interpolación. Evaluaremos nuestras técnicas de análisis sobre variados casos de estudio, y mostraremos que alcanzan mejores niveles de escalabilidad que otras técnicas previas a nuestras propuesta.

Palabras Claves: Ingeniería de Requisitos, Análisis Automático, SAT Solving, Interpolación, Model Checking.
PACS: D.2.1 Requirements/Specifications, D.3.1 Formal Definitions and Theory.

Abstract

Software Engineering aims at providing mechanisms for developing, designing and maintaining large-scale quality software, in an effective and systematic way. Several studies have shown that most software errors are caused by inconsistencies, misunderstandings and imprecisions that arise in early phases of software development, e.g., during requirements engineering, and in particular during the elicitation, elaboration and specification of software requirements. In addition, requirements related errors are more expensive and difficult to solve as the development process progresses. Therefore, the quality of software requirements specifications is very important, for practical and economical reasons, in most modern software development methodologies.

For this reason, in the last decades some *formal* requirements engineering methods (e.g., statecharts, SCR and other tabular notations, the KAOS method and Clafer) have gained more attention. Typically, these formal requirements engineering methodologies have associated powerful analysis mechanisms to assess and improve the quality of requirements specifications, such as the analysis of *consistency* (no contradicting requirements) and *completeness* (no missing cases).

On the other hand, there currently exist various algorithms that efficiently solve different kinds of analysis over logical formalisms, such as boolean satisfiability (SAT solving) or the computation of interpolants. Because of the efficiency and relative scalability that characterise these algorithms, they have been successfully used in many practical applications, such as software verification, testing and AI planning.

In this thesis, we present automated techniques that concern software requirements *elaboration*, *validation* and *verification*. Briefly, our techniques are based on the manipulation of logical formulas, taking advantage of the efficiency provided by the mechanisms associated with formal analysis techniques such as SAT solving, model checking and interpolation computation. We will assess our approaches on a variety of case studies, to show they can reach better levels of scalability when compared with related state-of-the-art techniques.

Keywords: Software Engineering, Automated Analysis, SAT Solving, Interpolation, Model Checking.
PACS: D.2.1 Requirements/Specifications, D.3.1 Formal Definitions and Theory.

Agradecimientos

Primero y principal, quiero agradecer a Naza, que con su constante apoyo y confianza, me ha brindado un sin fin de oportunidades que de otra manera hubiesen sido imposible de conseguir. No sólo un gran director y un ejemplo a seguir, sino además un amigo con el que he compartido grandes momentos.

En segundo lugar, y no menos importante, a todos mis compañeros del DC (Germán, Ponzio, Vale, Nico, Ceci, Chino, Gastón, Marta, Sonia, Pancho, Pablo, Valen, Ramiro, Marcelo, Simón), que con su entusiasmo, dedicación y amistad, hacen que pueda disfrutar todos los días de mi trabajo. En especial a Germán y Ponzio con quienes compartí muchas horas de trabajo, numerosas discusiones, y la satisfacción de haber logrado alguna que otra publicación. Al grupo de doctorandos (Vale, Nico, Ceci, Sonia, Valen) que siempre estuvieron a mi lado. A Ivana por simplificarme la vida muchas veces. A El Profe por su interés diario en la educación.

También quiero agradecer a Dalal y Sebas por brindarme su tiempo y amistad en mis visitas a Imperial. Gran parte de esta tesis se debe a sus aportes. A Dippi por su compañia y ayuda en muchas ocasiones cuando estuve lejos de casa. A Marcelo Frias por tenernos siempre en cuenta. A Carlos, Pedro y Javier, mi Comisión Asesora del doctorado, gracias por sus consejos y devoluciones.

Gracias a la Universidad Nacional de Río Cuarto por haberme formado, y hoy darme un lugar en el que me siento muy a gusto. A CONICET, que gracias a su financiamiento, no sólo este trabajo, sino muchas de las cosas que he logrado, han sido posibles. A FaMAF, y toda la gente que allí trabaja, por su cordial e inmediata respuesta a todo lo que he necesitado.

Finalmente, agradecer a mis padres, hermanos, suegros, toda mi familia y amigos, por su incondicional apoyo y aguante en los momentos más complicados. En especial, a Nadia y Jose, por su hermosa compañia de todos los días. Sin ellas nada sería posible.

Dedicatoria

Para Nadia y Jose con todo mi amor.

Gracias Juana por tu incondicional amistad.

Índice general

Índice de figuras

1

Introducción

En los inicios de la computación, los programas en general resolvían cálculos numéricos específicos y otros problemas de naturaleza matemática. En este contexto, los problemas a resolver contaban usualmente con una descripción formal precisa, y además, quienes los requerían (los usuarios finales) solían ser los mismos programadores, con lo cual la barrera de entendimiento entre el desarrollador y el usuario o cliente quedaba trivialmente resuelta. En los años 60's, el incremento del poder de cómputo de los componentes de una computadora y la reducción significativa de sus costos permitieron ampliar significativamente el campo de aplicación de la computación. Esto motivó el surgimiento de nuevos clientes y usuarios de programas (como empresas y organizaciones gubernamentales) con necesidades más ambiciosas. La falta de precisión y la complejidad de los nuevos problemas a resolver llevaron a que los proyectos de desarrollo no finalizaran a término, costaran más de lo estipulado, y presentaran numerosos errores. Esta *crisis del software* demostró que la complejidad y el tamaño de los sistemas de software demandan metodologías sistemáticas de desarrollo [Naur et al., 1969]. Surge así la idea de tratar el proceso de desarrollo como una actividad ingenieril. La *Ingeniería de Software* tiene como objetivo brindar mecanismos para crear, diseñar y mantener de manera efectiva, software de calidad y de gran escala. Típicamente, el proceso de desarrollo de software cuenta con etapas de *análisis*, *diseño*, *implementación* y *testing*. Numerosos estudios han mostrado que muchos de los errores manifestados en el software, se deben a imprecisiones en la etapa de análisis [Bell and Thayer, 1976; Brooks, 1987]. Más aún, los errores son más costosos y difíciles de resolver a medida que las etapas de desarrollo avanzan: se estima que cada error no detectado en la etapa de análisis cuesta 5 veces más en ser resuelto durante la etapa de diseño, 10 veces más en la implementación, 20 veces más durante la etapa de testing y 200 más después de haber entregado el producto final [Boehm and Papaccio, 1988]. Es por esto que la mayoría de las metodologías de desarrollo de software modernas [Ghezzi et al., 2002; Sommerville, 2006; Jalote, 2005] ponen suficiente énfasis en la etapa de análisis, teniendo como objetivo en esta etapa obtener una correcta comprensión del problema a resolver, plasmada en una especificación de requisitos de buena calidad.

Un *requisito* es una característica esperada del sistema. Puede ser desde una simple capacidad que el usuario espera del software a construir, hasta un objetivo que requiera la cooperación de varios componentes del sistema (personas u otras entidades del negocio, software de terceros, dispositivos de hardware, etc.). El *proceso de ingeniería de requisitos* consiste en identificar *cuál* es el problema a resolver y discernir *qué* debe hacer nuestro software para resolverlo [Jackson, 1995]. Lograr una descripción precisa de los requisitos es de suma importancia para construir el software esperado por el usuario. Es por esto que el *proceso de ingeniería de requisitos* implica varias tareas, como la captura, elaboración, especificación y validación de los requisitos. Sin embargo, este proceso puede ser muy complejo. Inevitablemente es necesaria la interacción con personas para resolver ambigüedades e imprecisiones, y poder así consolidar y negociar con el cliente las posibles soluciones. Inicialmente, las necesidades del cliente pueden no estar claras debido a la dificultad de visualizar el futuro sistema, por lo que los requisitos pueden cambiar de manera frecuente. Además, el lenguaje que escojamos para la documentación debe ser el indicado, ya que será utilizado para la interacción con los clientes (quienes no comprenden de programación) y guiará a los programadores en el proceso de desarrollo (quienes desconocen el dominio de aplicación).

La Ingeniería de Requisitos, reconocida como una disciplina en sí misma desde los 90's, provee metodologías para asistir a los desarrolladores durante el proceso de construcción de requisitos. Como ya mencionamos, este proceso está compuesto por diferentes etapas e involucra varios tipos de actores, o *stakeholders*, tales como usuarios finales, clientes, especialistas del dominio, desarrolladores, autoridades de certificación, etc. La *etapa de captura de requisitos* intenta comprender las necesidades del cliente desde su propia descripción informal del problema. Para esto debe lograr una buena comprensión del dominio del problema, identificar los stakeholders involucrados y la estructura organizacional del negocio, además de las fortalezas y debilidades del sistema actual. En esta etapa es esencial la cooperación con los stakeholders, para que nuestro entendimiento de *cuál* es el problema sea correcto. La *etapa de elaboración de los requisitos* consiste en definir *qué* funcionalidades (más importantes) deberá proveer el sistema. Aquí debemos identificar potenciales conflictos y riesgos para lograr los objetivos, comparar y elegir entre posibles alternativas, priorizar algunas funcionalidades sobre otras, etc., y junto al cliente elaborar los requisitos del software a construir. La *etapa de especificación de requisitos* consiste en redactar el *documento de requisitos*, describiendo precisamente toda la información que obtuvimos en las etapas anteriores (objetivos, conocimiento del dominio, definición de responsabilidades, asunciones realizadas, etc.), y estructurarlos de una manera coherente. Además podemos incluir toda información que consideremos relevante para el desarrollo: costos estimados, cambios en los requisitos, datos

para testing de aceptación, etc. El documento de requisitos establecerá una especie de contrato entre el cliente y el proveedor del software. Finalmente, la etapa de *validación de requisitos* busca brindar garantías de calidad de la especificación de requisitos. Es decir, es deseable que los requisitos especificados gocen de ciertas propiedades, como completitud, consistencia, no ambigüedad, realizabilidad, adaptabilidad, entre otras. Para evaluar si cumplen con estas características, los requisitos especificados son contrastados con las expectativas del cliente, o en muchos casos, si el lenguaje de especificación es adecuado, se pueden analizar de forma (semi-)automática. Las actividades mencionadas previamente, no se aplican obligatoriamente en un orden secuencial. Usualmente, necesitaremos volver a etapas anteriores, intercambiarlas, o realizar varias al mismo tiempo.

Variadas técnicas se han desarrollado para asistir al proceso de ingeniería de requisitos. Por ejemplo, para la etapa de captura, [Whitten and Bentley, 2007] provee una listas de preguntas frecuentes sobre el software a construir, que el ingeniero de requisitos siempre debería realizar a los stakeholders. Por otro lado, los *escenarios* particulares de uso del sistema han demostrado ser más fáciles de comprender por parte de los clientes, por lo que muchas metodologías de ingeniería de requisitos proponen lenguajes de descripción de requisitos basados en escenarios [Weidenhaupt et al., 1998]. En lo que respecta a la especificación de requisitos existen varios *lenguajes de especificación* que podemos utilizar. El *lenguaje natural* es una buena opción que nos brinda flexibilidad, amplia expresividad, y lo más importante, no impone barreras de comunicación con el cliente. Varias plantillas se han propuesto para estandarizar la estructura de los documentos de requisitos, como por ejemplo IEEE-1998 [IEEE, 1998]. Otras notaciones informales suelen utilizarse para complementar al lenguaje natural. En particular, las notaciones diagramáticas han tenido gran aceptación para la descripción de requisitos, como es el caso de los *diagramas de flujo de datos* (DFDs) [Stevens et al., 1974; DeMarco, 1979], los *casos de uso* [Cockburn, 2000; Maiden and Alexander, 2004], y los *diagramas de secuencias de mensajes* [Rumbaugh et al., 1999]. Al igual que el lenguaje natural, estas notaciones usualmente se utilizan durante la etapa de captura de requisitos para comunicarnos con el cliente, pero además, brindan un mecanismo útil para documentar la información obtenida. Los DFDs muestran cómo los datos fluyen y se transforman a lo largo del sistema, centrando su atención en las operaciones involucradas y la dependencia de datos entre ellas. Los casos de uso, en cambio, especifican el comportamiento del sistema mostrando su interacción con los usuarios y otros sistemas (stakeholders) en busca de alcanzar un objetivo. Por otro lado, los diagramas de secuencias de mensajes, una de las notaciones más utilizadas en la práctica, permiten describir de manera clara escenarios de ejecución o uso de un sistema.

También existen varias técnicas para la validación de requisitos, ya que obtener una especificación de requisitos de buena calidad, es de fundamental importancia práctica y económica en la mayoría de las metodologías de desarrollo de software modernas [Ghezzi et al., 2002; Sommerville, 2006; Jalote, 2005]. Por ejemplo, la estructuración del documento de requisitos que proponen las plantillas como [IEEE, 1998] (por ejemplo, definición de un glosario de términos), permiten lidiar con ciertos problemas asociados a la ambigüedad propia del lenguaje natural, que pueden conducir a múltiples o incorrectas interpretaciones de los requisitos. Por otro lado, las notaciones diagramáticas (como los DFDs, casos de uso y diagramas de secuencias de mensajes) cuentan con reglas *sintácticas* para su formación, posibilitando que algunas herramientas soporten editar gráficamente estos diagramas, chequear si sus estructuras son correctas, o analizar si cumplen algunas propiedades (por ejemplo, es posible analizar la ausencia de deadlock de protocolos de comunicaciones descriptos mediante diagramas de secuencias de mensajes [Mitchell, 2008]). Sin embargo, para poder realizar tipos de análisis más sofisticados a la especificación de requisitos, como *consistencia* (ausencia de requisitos contradictorios) y *completitud* (contemplación de todos los casos posibles), necesitamos que el lenguaje escogido tenga *sintaxis* y *semántica formales*. Es decir, la especificación de requisitos en un lenguaje formal no sólo sigue reglas estrictas para su formación, sino que además tiene reglas precisas para su interpretación (libre de ambigüedades).

En las últimas décadas las *notaciones formales* para la especificación de requisitos han adquirido mayor atención, surgiendo así varios lenguajes y metodologías formales para la ingeniería de requisitos, tales como los statecharts [Harel and Politi, 1998], SCR y otras notaciones tabulares [Heitmeyer et al., 2005], el método KAOS [van Lamsweerde, 2009] y Clafer [Bak et al., 2011]. En general, las metodologías formales de requisitos tienen asociados poderosos mecanismos (semi-)automáticos de análisis, que contribuyen y asisten al ingeniero en las diferentes etapas del proceso de ingeniería de requisitos. Por ejemplo, es posible realizar simulaciones sobre la especificación de requisitos [Heitmeyer et al., 2005; Van et al., 2004], chequear automáticamente consistencia y completitud [Heitmeyer et al., 1996; Bak et al., 2011; van Lamsweerde, 2008], generar casos de test que servirán como tests de aceptación [Gargantini and Heitmeyer, 1999], verificar si la especificación de requisitos cumple con sus objetivos [Letier et al., 2008], detectar conflictos u obstáculos entre los objetivos que el sistema debe alcanzar [Alrajeh et al., 2012] y sintetizar un modelo inicial de los componentes del software [Letier and Heaven, 2013], entre otros.

1.1 MOTIVACIÓN Y OBJETIVOS

Por lo mencionado anteriormente, la Ingeniería de Requisitos es probablemente el área más *informal* de la Ingeniería de Software. Naturalmente, el proceso de ingeniería de requisitos debe incluir al futuro cliente del sistema, y demás stakeholders, durante las etapas de captura, elaboración y validación de los requisitos. Esto conlleva a que el ingeniero de requisitos, y las técnicas desarrolladas en esta área, deban necesariamente lidiar con la *incertidumbre* e incluso el *desconocimiento* de ciertos requisitos, además de la *parcialidad* de muchas descripciones de los mismos.

Como ya mencionamos, el uso de un lenguaje formal, por ejemplo a través de algún formalismo lógico, elimina todo tipo de ambigüedad en sus expresiones, sin quedar sujetas a diferentes interpretaciones. Con esta misma motivación, las metodologías *formales* de requisitos surgidas recientemente, utilizan algún formalismo lógico para especificar los objetivos que persigue el sistema (por ejemplo, el método KAOS utiliza lógica temporal lineal [Dardenne et al., 1993]). La especificación formal de los requisitos busca resolver, en alguna medida, los problemas intrínsecos de los requisitos (como su incertidumbre y parcialidad), haciendo posible el desarrollo de herramientas de análisis que ayuden al ingeniero durante el complejo proceso de ingeniería de requisitos.

En la actualidad existen muchos algoritmos que eficientemente pueden resolver diferentes tipos de análisis lógicos, como el problema de satisfactibilidad booleana (SAT Solving) o el cálculo de interpolantes. Estos formalismos se caracterizan por su gran poder de cómputo y su buen manejo de fórmulas grandes y complejas. Cabe mencionar además, que han sido utilizados exitosamente en diversas áreas para atacar problemas como la verificación de software [Biere et al., 1999; Clarke et al., 2003; Jackson and Vaziri, 2000], el testing [Stephan et al., 1996] y planning en Inteligencia Artificial [Kautz and Selman, 1992].

En base a estas observaciones, nuestro objetivo en esta tesis es claro: "investigar la posibilidad de explotar el poder de análisis de algunas técnicas formales modernas, como SAT Solving e interpolación, para que mediante la manipulación lógica de fórmulas, podamos resolver de manera automática problemas actuales de la ingeniería de requisitos". Más concretamente, podemos resumir nuestros objetivos en los siguientes puntos:

- Utilizar de manera novedosa, técnicas de análisis automático provenientes de métodos formales de desarrollo, para tareas ligadas a la *elaboración* y *validación* de requisitos de software. En particular, técnicas basadas en satisfactibilidad booleana, y aquellas vinculadas al cómputo automático de interpolantes, así como también el model checking y la abstracción.

- Conseguir que las técnicas de análisis desarrolladas lleguen a niveles de escalabilidad aceptables. Esto demandará, como objetivo secundario, el desarrollo de mecanismos de optimización de las mismas que aprovechen características específicas del dominio de la ingeniería de requisitos, de las notaciones particulares empleadas, y de cada una de las actividades a automatizar.
- Producir herramientas para los mecanismos de análisis automáticos desarrollados, que además sean aplicables a las notaciones estudiadas. Con este objetivo intentamos que este trabajo no quede en una propuesta teórica exclusivamente, sino que pueda derivar en una metodología, sustentada en herramientas de análisis, que pueda ser aplicada en el desarrollo de software.
- Recompilar de la literatura diversos casos de estudios y aplicar sistemáticamente nuestras técnicas desarrolladas. Evaluar los resultados obtenidos y compararlos con técnicas que actualmente se encuentran en el estado-del-arte de la ingeniería de requisitos que atacan el mismo problema en cuestión.

1.2 ESTADO DEL ARTE

A pesar de que contamos con una amplia gama de herramientas para capturar, elaborar, especificar y validar requisitos de software, en este trabajo estamos principalmente interesados en las metodologías formales para la ingeniería de requisitos. En particular, nos centraremos en una familia de métodos que han demostrado ser exitosos para el proceso de ingeniería de requisitos, como son los denominados métodos orientados a objetivos y las notaciones tabulares de requisitos. Los métodos orientados a objetivos, entre los cuales se destacan KAOS [Dardenne et al., 1993] e I* [Yu, 1997], proponen describir explícitamente los requisitos mediante objetivos claramente definidos, la forma en que esos objetivos pueden conseguirse (descomposición y refinamiento en objetivos más simples), y los agentes que deberán involucrarse en su realización. Por otro lado, las notaciones tabulares de requisitos, como el método Software Cost Reduction (SCR) [Heitmeyer et al., 1996], proveen un lenguaje muy simple e imponen una estructura sobre la especificación, permitiendo organizar requisitos grandes y complejos, en expresiones más pequeñas que resultan ser más simples de comprender y modificar.

Los métodos orientados a objetivos han sido el foco de numerosas investigaciones por parte de la comunidad de ingeniería de software. Un aspecto interesante estudiado de estos métodos, por ejemplo en [Anton, 1997], es la relación de

refinamiento entre objetivos de alto nivel, realizables a través de la cooperación de varios agentes, en objetivos más simples que serán asignados a agentes específicos, como por ejemplo, el software a construir. Trabajos como [Letier and van Lamsweerde, 2002a; Letier and van Lamsweerde, 2004] presentan técnicas sistemáticas para producir diseños alternativos del sistema utilizando diferentes refinamientos posibles de los objetivos, análisis de conflictos y obstáculos entre los mismos, o especificando una noción de satisfacción parcial de los objetivos mediante probabilidades. En [van Lamsweerde and Willemet, 1998] se presenta un método basado en Explanation-Based Learning (EBL) [Mitchell, 1997] para inferir nuevos objetivos a partir de escenarios. Por otro lado, en [Uchitel et al., 2003] se presenta una técnica para sintetizar el comportamiento del sistema a partir de un conjunto de escenarios descriptos mediante diagramas de secuencias de mensajes. Esta técnica está integrada en la herramienta Labeled Transition System Analyzer (LTSA) [Magee and Kramer, 2006], por lo que el Sistema de Transición de Estados (LTS) que se obtiene puede ser analizado mediante simulación y model checking.

La *operacionalización de objetivos* es otro de los problemas extensamente estudiados. Básicamente, dados los objetivos que debe garantizar el software y el conjunto de operaciones a ser implementadas, la *operacionalización de objetivos* consiste en producir condiciones adecuadas para las operaciones que serán provistas y ejecutadas por el software, en pos de alcanzar los objetivos. Entre algunas de las técnicas que existen para la operacionalización de objetivos podemos mencionar a NFR [Mylopoulos et al., 1992] y CREWS [Rolland et al., 1998]. Sin embargo, éstas se centran en requisitos no funcionales o son informales, por lo que no pueden ser verificadas. Por otro lado, técnicas como [Fuxman et al., 2001; Fuxman et al., 2004] permiten chequear que una operacionalización es correcta en vez de brindar un soporte al proceso de elaboración de un modelo operacional de requisitos. En el trabajo [Letier and van Lamsweerde, 2002b] se propone una técnica basada en patrones para derivar requerimientos operacionales. Esta técnica provee algunas estrategias para derivar, desde objetivos expresados en Linear-Time Temporal Logic (LTL), un conjunto de precondiciones y condiciones de triggering para operaciones, que garantizan la satisfacción de dichos objetivos. Este enfoque está limitado a tipos particulares de objetivos LTL (impone ciertas restricciones sintácticas) y requiere que el modelo de objetivos esté completamente refinado (es decir, cada objetivo de bajo nivel debe estar asignado a un agente específico).

Más recientemente, Alrajeh et al. presentaron un enfoque semi-automático que aprende requisitos operacionales a partir de un conjunto de objetivos LTL [Alrajeh et al., 2009]. Este enfoque utiliza model checking para verificar si la especificación operacional de los requisitos satisface los objetivos. Si la

verificación falla, el usuario examina el contraejemplo retornado por el model checker, identifica la operación incorrectamente ejecutada, y provee escenarios positivos ilustrando "buenas" ocurrencias de dicha operación. Estos escenarios son utilizados por un motor de aprendizaje inductivo para automáticamente computar nuevas precondiciones o condiciones de triggering para la operación seleccionada. Los requerimientos operacionales obtenidos aseguran que el contraejemplo es removido y que el comportamiento de los escenarios positivos es preservado. Este enfoque de Alrajeh et al. es semi-automático ya que requiere intervención del ingeniero de requisitos para proveer los escenarios positivos.

Por otro lado, las notaciones tabulares SCR fueron utilizadas originalmente para documentar requisitos por Parnas y otros [K. Heninger and Shore, 1978], y han probado ser en la práctica un medio útil para caracterizar requerimientos complejos [Butler, 1996; Heitmeyer et al., 1998; K. Heninger and Shore, 1978]. El método SCR tiene asociado dentro del SCR Toolset un conjunto de herramientas de soporte que proveen diversos tipos de análisis, por ejemplo, chequeo de sintaxis, análisis de consistencia de la especificación y verificación de propiedades sobre los requisitos [Atlee and Gannon, 1991; Bultan and Heitmeyer, 2008; Heitmeyer et al., 2005]. Como en general los análisis del SCR Toolset se basan en model checking o demostración automática, usualmente presentan serios problemas a la hora de analizar sistemas complejos. Entre los intentos para mejorar su escalabilidad, en [Bharadwaj and Heitmeyer, 1999] se presentan técnicas de abstracción para especificaciones SCR, como el slicing automático. Sin embargo, la evaluación experimental de estas técnicas en [Heitmeyer et al., 2005], mostró que en varios casos es necesaria la reducción manual ad hoc de la especificación, para que el proceso de análisis mediante model checking finalice exitosamente.

La generación de casos de test para validación de requisitos de software, ha sido un tema de interés dentro de las especificaciones tabulares SCR. En particular, el simulador del SCR Toolset permite al desarrollador cargar posibles escenarios y asociar ciertas propiedades, para luego chequear si las ejecuciones que representan los escenarios las violan o no. Por otro lado, Gargantini y Heitmeyer utilizaron un model checker para obtener ejecuciones de las tablas (tests) que visiten diferentes *modos* del sistema [Gargantini and Heitmeyer, 1999]. Recientemente, Fraser y Gargantini realizaron un extensa comparación entre varios model checkers (simbólicos, acotados, de estados explícitos, etc.) para la generación automática de casos de test para especificaciones SCR, analizando la cobertura alcanzada y sus problemas de escalabilidad [Fraser and Gargantini, 2009].

1.3 CONTRIBUCIONES

La principal contribución de este trabajo es el desarrollo de novedosas técnicas *automáticas* que brindan soporte al proceso de elaboración, validación y verificación de requisitos de software. Estas técnicas realizan originales manipulaciones lógicas de fórmulas explotando la eficiencia de varios mecanismos de análisis provenientes de los métodos formales, como SAT solving e interpolación, así como también, el model checking y la abstracción.

Más precisamente, en esta tesis presentaremos dos técnicas automáticas que buscan resolver problemas que surgen en diferentes etapas del proceso de ingeniería de requisitos:

- Primero, presentaremos una técnica para la *elaboración formal de requisitos de software*. Concretamente, el problema a atacar es la *operacionalización automática de objetivos*. Dadas las operaciones que serán provistas por el componente de software a construir, y un conjunto de objetivos LTL a alcanzar, nuestra técnica refina automáticamente las precondiciones y condiciones de triggering adecuadas para las operaciones, garantizando la satisfacción de los objetivos. El proceso de refinamiento se basa en una combinación de interpolación y SAT solving. Nuestra técnica se aplica a objetivos de safety y progreso como lo hacen algunos enfoques anteriores, y además puede lidiar con un amplio rango de propiedades de *liveness*, principalmente, aquellas que pueden expresarse con el patrón de reactividad [Manna and Pnueli, 1992].

- La segunda técnica se aplica a la *validación y verificación* de requisitos. Básicamente, dada una propiedad (de safety) y una especificación de requisitos, nuestro método aplica automáticamente un proceso de abstracción, utilizando SMT solving, para verificar la validez de la propiedad sobre el modelo abstracto. En caso de obtener falsos positivos (contraejemplos del modelo abstracto que no pueden ser reproducidos en el modelo concreto), la técnica los remueve mediante un refinamiento automático de la abstracción utilizando interpolación. En particular, mostraremos que nuestra técnica puede ser utilizada para generar casos de tests y verificar propiedades sobre las especificaciones de requisitos, lidiando con el nivel de detalle original de las especificaciones, sin someterlas a reducciones manuales o fallar en el proceso de análisis como otros enfoques relacionados. Esto permite, entre otras cosas, que los casos de tests generados (o las violaciones detectadas) puedan ser directamente contrastados con las expectativas del usuario (validación) y con el comportamiento real del sistema implementado (verificación).

Evaluamos experimentalmente las técnicas propuestas sobre diversos casos de estudio, y comparamos los resultados obtenidos con varias de las técnicas del estado del arte mencionadas previamente. Entre los casos de estudio considerados, podemos mencionar modelos simples de: un controlador de una bomba de agua de una mina, un sistema de refrigeración de una planta nuclear, un control crucero de un automóvil, el piloto automático de un avión de combate, y un protocolo inteligente para sobrepasar automóviles. Cada modelo es acomopañado por una descripción informal del sistema, una especificación de requisitos, y los objetivos y propiedades a analizar. Los experimentos aquí presentados pueden descargarse desde `http://dc.exa.unrc.edu.ar/staff/rdegiovanni/es/casos-de-estudio.html` y reproducirse siguiendo las instrucciones que allí se pueden encontrar.

En particular, nuestro enfoque de operacionalización de objetivos fué comparado con la técnica semi-automática basada en Programación Lógica Inductiva (ILP) introducida en [Alrajeh et al., 2009]. La primera diferencia surge en base a que nuestra técnica es completamente automática, mientras que [Alrajeh et al., 2009] necesita que el usuario provea manualmente los escenarios positivos, para que ILP compute la operacionalización. Para desarrollar estos escenarios, seguimos las guías provistas en [Alrajeh et al., 2013]. Veremos que nuestro enfoque efectivamente produce la misma operacionalización, y en algunos casos más precisa, que la técnica basada en ILP, y además puede hacerlo de manera más eficiente, aprovechando la eficiencia de interpolación y SAT solving. Por último, es importante remarcar que nuestra técnica de operacionalización de objetivos es la primera en ser completamente automática y capaz de soportar objetivos de liveness expresados con el patrón de reactividad.

Por otro lado, la evaluación de nuestra técnica de análisis de requisitos basada en abstracción, consiste en medir su desempeño para tareas como la generación de casos de tests y la verificación de invariantes sobre las especificaciones. Los resultados fueron comparados con varios model checkers (Spin, NuSMV, Cadence SMV y SAL), que han probado ser un mecanismo útil para analizar especificaciones de requisitos [Fraser and Gargantini, 2009; Heitmeyer et al., 2005]. La comparación realizada muestra que nuestra técnica puede analizar exitosamente numerosos casos en los cuales model checking falla debido a que consume los recursos asignados, ya sea memoria o tiempo. Además, aún cuando ambos enfoques logran finalizar su análisis, nuestra técnica logra mejor escalabilidad, incluso de un orden de magnitud, en muchos casos. Por último, es importante mencionar que nuestra técnica basada en abstracción puede lidiar con el nivel de detalle original de las especificaciones, sin la necesidad de realizar reducciones manuales a las mismas.

Para finalizar, cabe aclarar que las técnicas presentadas en esta tesis están basadas y extienden artículos que hemos publicado recientemente [Degiovanni et al., 2011; Degiovanni et al., 2014]. Esta tesis debe considerarse como el resultado final del trabajo realizado a lo largo del doctorado. En colaboración con otros investigadores, y gracias a lo aprendido en esta tesis, además hemos publicado [Scilingo et al., 2013; Scilingo et al., 2014; Regis et al., 2015].

1.4 ORGANIZACIÓN

El resto de la tesis se organiza de la siguiente manera. El Capítulo 2 introduce los conceptos preliminares sobre los cuales basaremos las técnicas desarrolladas. Principalmente, presentaremos la sintáxis y semántica de las lógicas utilizadas como lenguaje de especificación por las metodología formales de requisitos, además de los mecanismos de análisis automáticos provenientes de los métodos formales, asociados a estas lógicas, que explotaremos en nuestras técnicas. En el Capítulo 3 presentamos los fundamentos de la Ingenieniería de Requisitos y dos metodologías formales para la especificación de requisitos de software: el método KAOS y el método SCR. Más precisamente, presentamos los lenguajes empleados por estos métodos para la especificación de requisitos y varios tipos de análisis que suelen aplicarse sobre ellos.

En el Capítulo 4 presentamos nuestra técnica automática para la operacionalización de objetivos. Primero, la Sección 4.2 presenta el ejemplo motivador que nos servirá para introducir nuestra técnica, y la Sección 4.3 describe formalmente el problema de operacionalización de objetivos. Luego, en la Sección 4.4, presentamos en detalle nuestra técnica para operacionalizar objetivos de safety, y en la Sección 4.5 mostramos como puede ser extendida para lidiar con objetivos de liveness. En la Sección 4.6 proponemos la metodología a seguir para lograr exitosamente operacionalizar los objetivos. En la Sección 4.7 evaluamos nuestra técnica, y la comparamos con otras relacionadas, sobre varios casos de estudio. Finalmente, discutimos los trabajos relacionados en Sección 4.8 y un resumen del capítulo en la Sección 4.9.

El Capítulo 5 presenta los detalles de nuestra técnica basada en abstracción para el análisis de especificaciones de requisitos. La Sección 5.2 motiva la técnica, describiendo el problema de aplicar abstracción cuando la especificación contiene demasiadas variables numéricas. La Sección 5.3 presenta nuestro algoritmo de abstracción especialmente diseñado para analizar especificaciones de requisitos SCR. La Sección 5.4 presenta en detalle toda nuestra técnica para analizar especificaciones de requisitos, utilizando el algoritmo de abstracción de la sección anterior. La Sección 5.5 describe en que consisten la generación de casos de test

y la verificación de invariantes para especificaciones de requisitos SCR. Luego, presentamos una extensa evaluación experimental de nuestra técnica y discutimos los resultados obtenidos en la Sección 5.6. Finalmente, presentamos los trabajos relacionados en la Sección 5.7 y resuminos el capítulo en la Sección 5.8.

La tesis finaliza en el Capítulo 6 presentando las conclusiones obtenidas a lo largo del desarrollo de este trabajo y mencionando algunas líneas de trabajos futuros.

2

Preliminares

2.1 Introducción

Este capítulo introduce brevemente las nociones básicas que utilizaremos a lo largo de este trabajo. Además incluye varias citas a trabajos previos relacionados, por si el lector desea una mayor descripción de cada tema. Principalmente, presentaremos las *lógicas* utilizadas como *lenguajes declarativos* para la especificación formal de requisitos de software. Además, presentaremos varios de los mecanismos de análisis automático asociados a estas notaciones, que serán explotados por las técnicas que desarrollaremos en los siguientes capítulos.

Primero, en la Sección 2.2 presentamos la *Lógica Proposicional*, la notación declarativa formal más simple que usaremos, y dos poderosos mecanismos de análisis automático asociados a esta lógica: *SAT solving* e *interpolación*. Luego, en la Sección 2.3 presentamos un formalismo ampliamente utilizado para modelar sistemas de software y sus comportamientos: los *Sistemas de Transición de Estados* (LTS). Además presentamos tres notaciones textuales para describir LTSs como FSP, Promela y Action Language. La Sección 2.4 introduce las *lógicas temporales* Linear-Time Temporal Logic (LTL) y Fluent Linear-Time Temporal Logic (FLTL), dos ricas extensiones a la Lógica Proposicional, que nos brindan la posibilidad de *especificar* de manera declarativa las propiedades que deseamos que nuestros modelos satisfagan. Por último, introducimos la noción de *Model Checking*, un mecanismo automático para verificar si un modelo (por ejemplo, un LTS) satisface una propiedad determinada descripta con alguna lógica temporal. En particular, presentamos los model checkers Labeled Transition System Analyser (LTSA), Spin y ALV que serán utilizados como soporte para varias de nuestras tareas.

2.2 Lógica Proposicional

Con el objetivo de especificar declarativamente los requisitos, vamos a necesitar un lenguaje lo suficientemente expresivo y con una semántica formal que nos permita realizar automáticamente diferentes análisis que nos interesan.

A continuación presentamos la Lógica Proposicional, uno de los lenguajes más simples que usaremos para describir requisitos, pero que nos dará las bases para introducir otros más expresivos. Un punto importante de esta lógica es que cuenta con poderosos mecanismos de análisis automático, como SAT solving e interpolación (ver siguientes secciones), sobre los cuales se basarán las técnicas desarrolladas en esta tesis.

La *Lógica Proposicional* (LP) es un formalismo matemático muy simple que permite caracterizar propiedades del mundo mediante *proposiciones* (sentencias que pueden ser verdades o falsas) y describir cómo éstas se relacionan entre sí mediante *operadores lógicos*. Algunos ejemplos de proposiciones pueden ser: "p: está nublado" y "q: está lloviendo"; u otras que prediquen sobre sistemas de software, como: "*crit*: el proceso se encuentra en la región crítica" y "*pumpOn*: la bomba de agua está encendida".

Las *variables proposicionales* (como p, q, *crit* y *pumpOn* en los ejemplos), junto a los operadores lógicos ($\neg, \vee, \wedge, \rightarrow$) y los valores de verdad constantes T(verdadero) y F(falso), forman el *alfabeto* de la LP, y sus elementos pueden combinarse para formar expresiones más complejas y expresivas. Sin embargo, hay ciertas reglas estrictas a seguir para generar estas fórmulas que definen el *lenguaje* de la LP. Veamos formalmente como se caracterizan las fórmulas proposicionales.

Definición 2.1 (Sintaxis de fórmulas LP)**.** *Sea PA un conjunto de variables proposicionales; luego:*

- *Los valores de verdad T(verdadero) y F(falso) son fórmulas de LP;*
- *Toda proposición $p \in PA$ es una fórmula de LP;*
- *Si φ_1, φ_2 son fórmulas de LP, entonces también son fórmulas de LP: $\neg\varphi_1$ y $\varphi_1 \vee \varphi_2$.*
- *Podemos definir los operadores derivados $\wedge$, $\rightarrow$ y $\leftrightarrow$ de la siguiente manera: $\varphi_1 \wedge \varphi_2 \equiv \neg(\neg\varphi_1 \vee \neg\varphi_2)$, $\varphi_1 \rightarrow \varphi_2 \equiv \neg\varphi_1 \vee \varphi_2$ y $\varphi_1 \leftrightarrow \varphi_2 \equiv (\varphi_1 \rightarrow \varphi_2) \wedge (\varphi_2 \rightarrow \varphi_1)$.*

Usando las proposiciones anteriores, podemos formar por ejemplo las siguientes fórmulas LP:

- $q \rightarrow p$: si "está lloviendo" entonces "está nublado",
- $\neg p \wedge \neg q$: no "está nublado" y no "está lloviendo",
- *crit* $\leftrightarrow \neg$*pumpOn*: "el proceso se encuentra en la región crítica", si y sólo si, no ocurre que "la bomba está encendida".

Es necesario incluir paréntesis debido a que, de acuerdo a la semántica de las fórmulas (que veremos más adelante), diferentes asociaciones de conectivos dan lugar a diferentes interpretaciones. Por ejemplo, $\neg(p \wedge q)$ no significa lo mismo que $\neg p \wedge q$. En otros casos, por ejemplo $(p \wedge q) \rightarrow (\neg p \vee q)$, puede evitarse el uso de paréntesis ya que se define una relación de *precedencia* entre los conectivos lógicos (de mayor a menor): $\neg, \wedge, \vee, \rightarrow, \leftrightarrow$. Además, se considera que los operadores binarios asocian a derecha, por lo que $p \rightarrow q \rightarrow r$ es lo mismo que $p \rightarrow (q \rightarrow r)$.

Para la manipulación automática de fórmulas proposicionales, usualmente es importante que éstas sean llevadas a una *forma normal*. Existen varias formas normales, pero en este trabajo nos interesa particularmente la *Forma Normal Conjuntiva* (CNF), cuya estructura es explotada por los algoritmos de SAT Solving que introduciremos más adelante.

Definición 2.2 (Forma Normal Conjuntiva)**.** *Un* literal *es una variable proposicional o su negación (por ejemplo, p y $\neg q$). Una* cláusula *es una disyunción de literales (por ejemplo, $p \vee \neg q \vee r$). Diremos que una fórmula está en* Forma Normal Conjuntiva *(CNF) si es una conjunción de cláusulas. Por ejemplo, $(p \vee \neg q \vee r) \wedge (\neg p \vee s \vee t \vee \neg u)$.*

El significado de una fórmula LP φ depende de los valores de verdad de cada una de las proposiciones presentes en φ.

Definición 2.3 (Interpretación)**.** *Sea φ una fórmula LP y PA^{φ} el conjunto de proposiciones que aparecen en φ. Una* interpretación $I \subseteq PA^{\varphi}$ *para la fórmula φ, es un subconjunto de las proposiciones de φ que indica cuáles proposiciones son verdaderas. Es decir, para toda $p \in PA^{\varphi}$, si $p \in I$ significa que p tiene el valor de verdad verdadero (T). Por lo contrario, si $p \notin I$, entonces p es falso (F).*

Definición 2.4 (Semántica de fórmulas LP)**.** *Sea φ una fórmula de LP e I una interpretación para φ. Diremos que φ es verdadera bajo I si y sólo si $I \models_{LP} \varphi$, donde $\models_{LP}$ se define inductivamente de la siguiente manera:*

$$
\begin{array}{ll}
I \models_{LP} \mathsf{T} & \\
I \not\models_{LP} \mathsf{F} & \\
I \models_{LP} p & \textit{iff } p \in I \\
I \models_{LP} \neg\phi & \textit{iff } I \not\models_{LP} \phi \\
I \models_{LP} \phi_1 \wedge \phi_2 & \textit{iff } I \models_{LP} \phi_1 \textit{ y } I \models_{LP} \phi_2 \\
I \models_{LP} \phi_1 \vee \phi_2 & \textit{iff } I \models_{LP} \phi_1 \textit{ ó } I \models_{LP} \phi_2 \\
I \models_{LP} \phi_1 \rightarrow \phi_2 & \textit{iff, } I \not\models_{LP} \phi_1 \textit{ ó } I \models_{LP} \phi_2 \\
I \models_{LP} \phi_1 \leftrightarrow \phi_2 & \textit{iff } I \models_{LP} \phi_1 \textit{ y } I \models_{LP} \phi_2\textit{, ó } I \not\models_{LP} \phi_1 \textit{ y } I \not\models_{LP} \phi_2
\end{array}
$$

Por ejemplo, consideremos la fórmula $\varphi = (\neg p \vee q) \rightarrow (r \wedge q)$ y la interpretación $I = \{p, q, r\}$. Como p es verdadera bajo I ($p \in I$), la subfórmula $\neg p$ es falsa, pero como q es verdadera ($q \in I$), la subfórmula $\neg p \vee q$ es verdadera. Además, como r también es verdadera ($r \in I$), la subfórmula $r \wedge q$ también es verdadera. Luego, la fórmula φ resulta verdadera bajo I.

Definición 2.5 (Satisfactibilidad)**.** *Decimos que una fórmula proposicional φ es* satisfactible *si* existe *una interpretación I bajo la cuál φ evalúa a verdadero ($I \models_{LP} \varphi$). En otro caso, φ se dice* insatisfactible*.*

Definición 2.6 (Validez)**.** *Decimos que una fórmula proposicional φ es* válida *si es verdadera bajo* toda *interpretación posible.*

Note que los conceptos de satisfactibilidad y validez son duales. Una fórmula φ es válida si y sólo si $\neg\varphi$ es insatisfactible. De la misma manera, si φ es satisfactible, $\neg\varphi$ no puede ser válida. Debido a esta dualidad, lo que haremos a continuación es centrarnos sobre el concepto de satisfactibilidad, pero los resultados pueden expresarse en términos de validez.

2.2.1 *Satisfactibilidad Booleana*

El problema de satisfactibilidad booleana (*SAT*) es un problema de decisión que, dada una fórmula proposicional φ, consiste en buscar una interpretación bajo la cual φ evalúa a verdadero. Se sabe que el problema de SAT es NP-completo [Cook, 1971]. Sin embargo, en la actualidad existen muchos algoritmos que resuelven el problema de satisfactibilidad booleana muy eficientemente, para clases de fórmulas importantes.

A lo largo de esta tesis, SAT estará detrás de muchas tareas. Por ejemplo, la técnica de operacionalización de objetivos utiliza SAT para asegurar que cada refinamiento es *consistente*. Precisamente, SAT permite chequear que la condición de triggering de la operación refinada no contradice su precondición (es decir, si la operación esta obligada a ejecutarse, también debe estar habilitada a hacerlo). También utilizamos SAT para chequear si una operación está habilitada a ejecutarse en cierto estado del sistema. Por otro lado, nuestra técnica de análisis de especificaciones de requisitos utiliza SAT, no sólo para realizar varios chequeos de consistencia como los anteriores, sino que además nos permitirá construir de forma eficiente modelos abstractos de sistemas.

Un *SAT Solver* es un programa que automáticamente decide si una fórmula proposicional es satisfactible. En general, los procedimientos modernos para decidir SAT se basan en el algoritmo Davis-Putnam-Logemann-Loveland (*DPLL*)

[Davis and Putnam, 1960; Davis et al., 1962]. Si el lector está interesado, puede encontrar una breve descripción del algoritmo DPLL a continuación.

Algoritmo DPLL para SAT

DPLL toma una fórmula proposicional en CNF (ver Definición 2.2) e intenta construir una interpretación que haga verdadera a la fórmula; si no logra encontrarla, responde que la fórmula es insatisfactible. Note que, en el peor caso, deberá considerar todas las posibles asignaciones de valores de verdad a las variables proposicionales de la fórmula para construir la interpretación. Para simplificar la exploración de todas estas posibilidades, DPLL utiliza *Boolean Constraint Propagation* (BCP), una estrategia que aprovecha las cláusulas unitarias presentes en la fórmula y acelera así la convergencia en la búsqueda de interpraciones que hagan a la fórmula verdadera.

Definición 2.7 (Resolución Unitaria)**.** *Una* clausula unitaria *consiste de un único literal ℓ ($\ell = p$ ó $\ell = \neg p$, para alguna variable proposicional p). Si tenemos una clausula unitaria ℓ y otra clausula que contiene su negación ($C[\neg\ell]$), luego mediante* Resolución Unitaria *podemos deducir $C[\mathsf{F}]$:*

$$\frac{\ell \quad C[\neg\ell]}{C[\neg\ell \mapsto \mathsf{F}]}$$

Note que Resolución Unitaria nos permite reducir el número de clausulas y de literales de las fórmulas proposicionales. La clausula resultante $C[\neg\ell \mapsto \mathsf{F}]$ es similar a la clausula C pero la ocurrencia de $\neg\ell$ es remplazada por falso, ya que la clausula unitaria ℓ debe ser necesariamente verdadera si queremos construir una interpretación que haga verdadera a la fórmula.

Definición 2.8 (Boolean Constraint Propagation)**.** *Dada una fórmula proposicional φ en CNF,* Boolean Constraint Propagation *(BCP) consiste en aplicar repetidamente* Resolución Unitaria *a φ hasta obtener una fórmula φ' que ya no puede reducirse.*

Por ejemplo, sea inicialmente $\varphi = (p) \wedge (\neg p \vee q) \wedge (r \vee \neg q \vee s)$. Luego BCP aplicaría las siguientes reducciones:

1. $\dfrac{(p) \quad (\neg p \vee q)}{q}$ produciendo $\varphi' = (q) \wedge (r \vee \neg q \vee s)$.
2. $\dfrac{(q) \quad (r \vee \neg q \vee s)}{r \vee s}$ produciendo $\varphi'' = (r \vee s)$.

3. BCP finaliza y retorna φ'', ya que no puede aplicar más Resolución Unitaria.

Como muestra el ejemplo, la aplicación de Resolución Unitaria puede generar otras clausulas unitarias. DPLL utiliza BCP para eliminar la mayor cantidad de clausulas unitarias posibles. Se estima que los SAT Solvers modernos dedican entre el 80-90 % del tiempo de análisis aplicando BCP. Puede encontrar una breve descripción del algoritmo DPLL en Algoritmo 2.1.

Algorithm 2.1 DPLL

```
1:  function DPLL(φ: formula proposicional CNF)
2:      φ' ← BCP(φ)
3:      if φ' = ⊤ then
4:          return SAT
5:      else if φ' = ⊥ then
6:          return UNSAT
7:      else
8:          P ← Choose(vars(φ'))
9:          if DPLL(φ'[P ↦ ⊤]) = UNSAT then
10:             if DPLL(φ'[P ↦ ⊥]) = UNSAT then
11:                 return UNSAT
12:         return SAT
```

2.2.2 *Interpolación*

El *Teorema de Interpolación de Craig* fué introducido en al año 1957 por William Craig [Craig, 1957a] y es considerado como uno de los últimos resultados más profundos en el campo de la lógica y la matemática. La primer versión del teorema fue formulada para la Lógica de Primer Orden, pero muchas variantes fueron presentadas, incluso para lógicas muy expresivas que contienen enteros y reales [Cimatti et al., 2010]. En la última década, a partir del paper de McMillan [McMillan, 2003], Interpolación ha sido reconocida como una herramienta útil para la la verificación de sistemas [Mcmillan, 2005; Cabodi et al., 2006; Li and Somenzi, 2006]. En este trabajo seguiremos la definición de Interpolación para la Lógica Proposicional presentada en [McMillan, 2003].

Definición 2.9 (Interpolante)**.** *Dadas dos fórmulas proposicionales φ y ψ, tales que "$\varphi \wedge \psi$ es insatisfacible", un interpolante para φ y ψ es una fórmula $\hat{\varphi}$ con las siguientes propiedades:*

1. *$\varphi \rightarrow \hat{\varphi}$ es válido,*

2. $\hat{\varphi} \wedge \psi$ *es insatisfacible,*

3. $\boldsymbol{vars}(\hat{\varphi}) \subseteq \boldsymbol{vars}(\varphi) \cap \boldsymbol{vars}(\psi)$*, es decir,* $\hat{\varphi}$ *sólo incluye variables proposicionales de* φ *y* ψ*.*

Intuitivamente, podríamos pensar que $\hat{\varphi}$ sólo se centra en la información importante de φ que lo llevan a ser inconsistente con ψ. Más precisamente, $\hat{\varphi}$ es una abstracción de φ desde el punto de vista de ψ, que explica porqué φ es inconsistente con ψ en términos del lenguaje que comparten. Como un simple ejemplo, considere $\varphi = (\neg p) \wedge (p \vee \neg q)$ y $\psi = (\neg p \vee q) \wedge (p)$, luego el interpolante para estas formulas es $\hat{\varphi} = (\neg p \wedge \neg q)$.

Al igual que SAT, el cómputo de interpolantes juega un rol central en las técnicas presentadas en esta tesis. Nuestra técnica de operacionalización de objetivos utiliza interpolación para identificar la parte de la especificación (operacional) de los requisitos que viola los objetivos, es decir, la parte de la especificación que es inconsistente con los objetivos. Luego, mediante una manipulación lógica del interpolante, nuestra técnica produce los refinamientos necesarios sobre las operaciones, para finalmente lograr satisfacer los objetivos.

Para el caso de nuestra técnica basada en abstracción, puede ocurrir que al verificar si una propiedad vale sobre una especificación de requisitos, el análisis arroje *contraejemplos espurios* (o falsos positivos). Básicamente, un contraejemplo espurio es una ejecución del modelo abstracto que viola la propiedad, pero que no puede ser reproducida en el modelo concreto. Interpolación permite detectar porqué el contrajemplo espurio es inconsistente con el modelo concreto. Nuestra técnica agrega el interpolante a la abstracción para hacerla más precisa, y de esta manera remover del modelo abstracto los falsos positivos.

A partir de una prueba de insatisfactibilidad es posible computar eficientemente un interpolante, por lo que muchos SAT Solvers incluyen algoritmos de interpolación. En nuestro caso utilizaremos MathSAT, herramienta que presentaremos más adelante, para computar interpolantes. Es importante remarcar que utilizamos interpolación como una caja negra, es decir, no implementamos ningún algoritmo propio de interpolación. Para el lector interesado, a continuación presentamos brevemente el algoritmo de interpolación que MathSAT implementa.

Algoritmo de Interpolación

Craig mostró que, para fórmulas de primer orden inconsistentes, siempre existe un interpolante [Craig, 1957b]. Además, se ha probado que para dos fórmulas proposicionales φ y ψ, es posible obtener un interpolante desde la

prueba de refutación para $\varphi \wedge \psi$ en tiempo lineal [Krajícek, 1997; Pudlák, 1997]. El algoritmo de Interpolación que utilizaremos es tomado de [McMillan, 2003], que presentaremos a continuación.

Definición 2.10 (Resolvente)**.** *Dadas dos clausulas* $c_1 = v \vee \alpha$ *y* $c_2 = \neg v \vee \beta$*, llamaremos* resolvente *de* c_1 *y* c_2 *a la clausula* $\alpha \vee \beta$*, siempre y cuando* $\alpha \vee \beta$ *no sea tautología (verdadera bajo cualquier interpretación). Por ejemplo, el resolvente de* $p \vee q$ *y* $\neg p \vee r$ *es* $q \vee r$*. Pero,* $p \vee q$ *y* $\neg p \vee \neg q$ *no tienen resolvente ya que* $q \vee \neg q$ *es una tautología. Usualmente a la variable* v *se la llama* variable pivot *de* c_1 *y* c_2*.*

Definición 2.11 (Prueba de Insatisfactibilidad)**.** *Una* Prueba de Insatisfactibilidad Π *para un conjunto de clausulas* C *(denotada* (Π, C)*)es una Grafo Dirigido Acíclico* (V_Π, E_Π)*, donde* V_Π *es un conjunto de clausulas tales que:*

- *para todo vértice* $c \in V_\Pi$*, ocurre (sólo) una de las siguientes condiciones:*
  - $c \in C$*, y* c *es una raíz, ó*
  - c *tiene exactamente dos predecesores,* c_1 *y* c_2*, tales que* c *es el resolvente de* c_1 *y* c_2*,*
- *y la clausula vacía* ($\square$) *es la única hoja.*

Si tenemos dos fórmulas proposicionales φ y ψ en CNF y podemos construir una Prueba de Insatisfactibilidad $(\Pi, \varphi \cup \psi)$, entonces $\varphi \wedge \psi$ es insatisfactible.

Definición 2.12 (Variables Globales y Locales)**.** *Dada una* Prueba de Insatisfactibilidad $(\Pi, \varphi \cup \psi)$ *para dos fórmulas proposicionales* φ *y* ψ *en CNF, diremos que una variables es* global *si aparece en* φ *y* ψ*, y será* local a φ *si aparece sólo en* φ*. De manera similar, un literal es global o local según la variable que contiene. Dada una clausula* c*, denotaremos:*

- $g(c)$ *a la disyunción de todos los literales de* c *que son globales,*
- $l(c)$ *a la disyunción de todos los literales de* c *que son locales a* φ*.*

Por ejemplo, si $\varphi = (p \vee q \vee \neg r)$ y $\psi = (q \vee r \vee \neg s)$, luego $g(\varphi) = (q \vee \neg r)$, $l(\varphi) = (p)$, $g(\psi) = (q \vee r)$ y $l(\psi) = \mathsf{F}$.

Definición 2.13 (Π-interpolante)**.** *Sean* φ *y* ψ *dos fórmulas proposicionales en CNF, y* $(\Pi, \varphi \cup \psi)$ *una Prueba de Insatisfactibilidad, con único vértice hoja* $\square$*. Para todos los vértices* $c \in V_\Pi$*, definimos* $p(c)$ *como una fórmula, tal que:*

- *si* c *es raíz, entonces:*

- *si* $c \in \varphi$ *entonces* $p(c) = g(c)$,
- *sino* $p(c) = \mathsf{T}$ *(la constante verdadera).*

▪ *si* c *no es raíz, sean* c_1 *y* c_2 *los predecesores de* c *y sea* v *su variable pivot:*

- *si* v *es local a* φ *entonces* $p(c) = p(c_1) \vee p(c_2)$,
- *sino* $p(c) = p(c_1) \wedge p(c_2)$.

El Π*-interpolante de* (φ, ψ)*, denotado como* $ITP(\Pi, \varphi, \psi)$ *es* $p(\square)$.

En [McMillan, 2003] se prueba que $ITP(\Pi, \varphi, \psi)$ es efectivamente un interpolante para φ y ψ. Veamos un ejemplo del cómputo de un interpolante utilizando el algoritmo recién presentado. Considere $\varphi = (\neg p) \wedge (p \vee \neg q)$ y $\psi = (\neg p \vee q) \wedge (p)$. La Figura 1 muestra una Prueba de Insatisfactibilidad $(\Pi, \varphi \cup \psi)$ para estas fórmulas. Luego, la Figura 2 muestra la ejecución del algoritmo, para finalmente computar el interpolante $ITP(\Pi, \varphi, \psi) = \neg p \wedge (p \vee \neg q) = (\neg p \wedge \neg q)$.

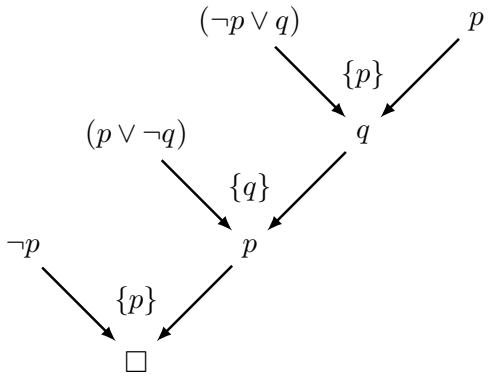

Figura 1: $(\Pi, \varphi \cup \psi)$

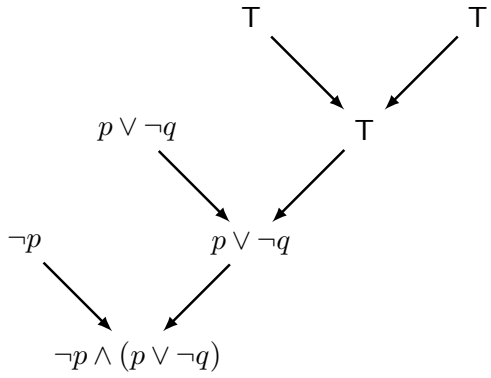

Figura 2: $ITP(\Pi, \varphi, \psi)$

2.2.3 *Satisfactibilidad Modulo Teorías*

Notoriamente, *SAT Solving* ha sido utilizado en diversas áreas, incluyendo la verificación de software [Biere et al., 1999; Clarke et al., 2003; Jackson and Vaziri, 2000], testing [Stephan et al., 1996] y planning en Inteligencia Artificial [Kautz and Selman, 1992]. Esto no sólo se debe al constante incremento en la eficiencia de los SAT Solvers, sino también a las diferentes funcionalidades que estos proveen, como la generación de modelos, producción de pruebas, extracción de núcleos insatisfactibles y el computo automático de interpolantes. Una de las limitaciones reconocidas sobre los métodos de análisis basados en SAT

yace sobre su poder expresivo. Utilizando sólo lógica proposicional, puede ser muy complicado analizar sistemas prácticos que contienen variables de tipos que proveen los lenguajes de programación, como enteros, reales y cadenas de caracteres. Esto motivó la búsqueda de fragmentos de la Lógica de Primer Orden en los cuales sea posible encontrar automáticamente una interpretación para una fórmula descripta dentro de ese fragmento (por ejemplo, aritmética lineal).

Satisfactibilidad Modulo Teorías (*SMT Solving*) es una extensión de SAT Solving que incluye teorías para razonar sobre fórmulas que involucran tipos de datos que no son booleanos (por ejemplo, cadenas de caracteres, arreglos, enteros, etc). A diferencia de SAT Solving, la asignación de valores encontrada por SMT Solving puede involucrar valores no booleanos. Las teorías asociadas con SMT Solving son decidibles (SMT es restringido a fragmentos decidibles de teorías para tipos de datos), y procesos de desición para estas teorías son combinadas con SAT Solving para encontrar las valuaciones que hagan verdadera la fórmula. El proceso de SMT Solving es completamente automático, y al igual que SAT Solving, es posible generar modelos, producir pruebas, extraer núcleos insatisfactibles y computar interpolantes.

En este trabajo usaremos el SMT Solver MathSAT [Bruttomesso et al., 2008]. Esta herramienta estará detrás de varias tareas, como el computo de estados abstractos del sistema, verificar si cierta operación está habilitada a ejecutarse en determinado estado, si la ejecución de dicha operación nos conduce a un estado indeseado, etc. Más notablemente, MathSAT es capaz de computar *interpolantes*, los cuales jugaran un rol central a lo largo de este trabajo.

El SMT Solver MathSAT

MathSAT [Bruttomesso et al., 2008] implementa un proceso de decisión, basado en DPLL (Davis-Putnam-Logemann-Loveland), para el problema de SMT sobre varias teorías, incluyendo aquellas con Igualdad y Funciones no Interpretadas (EUF), Lógicas con Diferencia (DL), Aritmética Lineal para Reales (LA(R)) y Aritmética Lineal para Enteros (LA(Z)). MathSAT está basado en un esquema de integración lazy (organizado en jerarquías acorde a la expresividad de la lógica) que usan las mayoría de los SMT Solver del estado del arte, donde el razonamiento proposicional es cuidadosamente integrado con Solvers que incrementan el poder expresivo, de manera tal que los niveles de mayor poder expresivo son invocados con menor frecuencia [Bruttomesso et al., 2007]. Una funcionalidad que destaca a MathSAT es la posibilidad de computar interpolantes para fórmulas SMT (por ejemplo, para las teorías LA(R) y LA(Z)) y combinaciones de teorías usando Delayed Theory Combination (DTC) [Cimatti

et al., 2010]. Además, MathSAT ha sido aplicado sobre diferentes dominios de aplicación del mundo real, como sistemas híbridos y de tiempo real.

MathSAT soporta tres formatos de entrada: un lenguaje nativo MSAT, el lenguaje estándar SMT-LIB, y uno usado por el demostrador de teoremas basado en interpolación FOCI [McMillan, 2005]. Nosotros utilizaremos el lenguaje nativo MSAT, lo suficientemente expresivo para definir variables de tipos simples (`BOOLEAN`, `INTEGER`, `REAL`, etc) y tipos compuestos (funciones no interpretadas), constantes, expresiones lógicas (`!`, `&`, `|`, `->`, `<->`) y expresiones matemáticas (`=`, `!=`, `<`, `>`, `<=` y `>=`). Puede encontrar una descripción detallada del lenguaje en `http://mathsat4.disi.unitn.it/`.

Supongamos que tenemos que modelar el comportamiento de una bomba de agua dentro de una mina (ejemplo extraído de [Kramer et al., 1983]). El estado de la bomba cambia acorde al nivel del agua: la bomba debe encenderse si estaba apagada y el nivel del agua es mayor a High; la bomba debe apagarse si estaba encendida y el nivel del agua es menor a Low. En la Figura 3 especificamos una fórmula SMT que modela todos las posibles transiciones de este sistema. Las variables enteras `waterLevel0` y `waterLevel1` modelan el estado previo y siguiente del nivel de agua, respectivamente, respecto a un incremento (`incWater`) o decremento (`decWater`) del nivel. Las variables booleanas `pumpOn0` y `pumpOn01` representan el estado de la bomba, antes y después de la ejecución de los eventos que la encienden (`switchPumpOn`) y apagan (`switchPumpOff`), respectivamente.

Es posible utilizar MathSAT para analizar si existe alguna asignación de valores a cada una de las variables de manera tal que la fórmula SMT descripta en Figura 3 evalúe a verdadero. Una instancia que MathSAT podría retornar es: `waterLevel0=31, waterLevel1=31, !incWater, !decWater, !pumpOn0, pumpOn01, switchPumpOn, !switchPumpOff`. Esta instancia muestra una situación en la cuál la bomba se enciende porque el nivel del agua es mayor a `High`, provisto de que además estaba apagada.

2.3 MODELOS DE COMPORTAMIENTO

Tradicionalmente, los programas eran pensados de manera *secuencial* como funciones que definen una relación entre valores de entrada (datos) y valores de salida (resultados), tal como se muestra en la Figura 4. Sin embargo, la *concurrencia* comenzó a ser un elemento ampliamente utilizado en el desarrollo de sistemas: múltiples programas interactuando entre si y con el ambiente para lograr un objetivo en común. Surgieron así lo que se conoce como *sistemas reactivos*. Este tipo de sistemas tienen como principal objetivo mantener una

```
1  VAR
2   waterLevel0, waterLevel1: INTEGER
3   incWater, decWater: BOOLEAN
4   pumpOn0, pumpOn01: BOOLEAN
5   switchPumpOn, switchPumpOff: BOOLEAN
6
7  DEFINE
8   Low := 10
9   High := 30
10
11 FORMULA
12
13 # Modelado del cambio del nivel de agua dentro de la mina.
14 ( (incWater & waterLevel1 = waterLevel0 + 1)
15 | (decWater & waterLevel1 = waterLevel0 - 1)
16 | (!incWater & !decWater & waterLevel1 = waterLevel0)
17 )
18 &
19 # Modelado del cambio de estado de la bomba, acorde al nivel de agua.
20 ( (switchPumpOn & !pumpOn0 & waterPres0 > High & pumpOn1)
21 | (switchPumpOff & pumpOn0 & waterPres0 < Low & !pumpOn1)
22 | (!(switchPumpOn & !pumpOn0 & waterPres0 > High)
23    & !(switchPumpOff & pumpOn0 & waterPres0 < Low) & pumpOn1=pumpOn0)
24 )
```

Figura 3: Ejemplo de especificación en MSAT.

interacción con el ambiente en el que se ejecutan, en vez de producir un valor al terminar, como se muestra en Figura 5. Ejemplos de sistemas reactivos pueden ser los sistemas operativos, drivers, controladores de tiempo real (marcapasos, automóviles, etc). Como puede imaginarse, la construcción de sistemas reactivos puede ser una tarea muy compleja. Es por ello, que se han desarrollado numerosos formalismos de modelado para poder razonar y validar estos sistemas, entre los que podemos destacar a los Sistemas de Transición de Estados.

Figura 4: Programa Secuencial

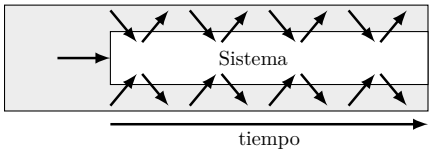

Figura 5: Sistema Reactivo

2.3.1 *Sistemas de Transición de Estados*

Un *Sistema de Transiciones Etiquetadas de Estados*, o simplemente *LTS* (Labelled Transition Systems), es un formalismo utilizado para describir el comportamiento del sistema y la interacción de sus componentes, usualmente caracterizados mediante *estados* y *transiciones* [Keller, 1976]. Formalmente, podemos definir un LTS de la siguiente manera:

Definición 2.14 (Estados y Transiciones)**.** *Definimos* Σ *y Act como el conjunto universal de estados y transiciones, respectivamente, y* τ *como una transición especial que denota eventos internos de un componente que no son observables desde el exterior.*

Definición 2.15 (Labelled Transition Systems)**.** *Un LTS* $\mathcal{P}$ *es una cuádrupla* $\langle Q, A, \delta, q_0 \rangle$*, donde:*

- $Q \subseteq \Sigma$ *es un conjunto finito de estados,*
- $A \subseteq Act$ *es el alfabeto de* $\mathcal{P}$*,*
- $\delta \subseteq Q \times A \cup \{\tau\} \times Q$ *relación de transiciones etiquetadas, y*
- $q_0 \in Q$ *es el estado inicial.*

Continuando con el modelo de la bomba de agua dentro de la mina presentado en la sección anterior, la Figura 6 muestra como podríamos modelar con un LTS el componente que mide el nivel del agua dentro de la mina.

$$\texttt{WaterLevel} = \langle \{0,1,2\}, \{\texttt{aboveLow}, \texttt{belowLow}, \texttt{aboveHigh}, \texttt{belowHigh}\},$$
$$\{(0, \texttt{aboveLow}, 1), (1, \texttt{belowLow}, 0), (1, \texttt{aboveHigh}, 2), (2, \texttt{belowHigh}, 1)\}, 0\rangle$$

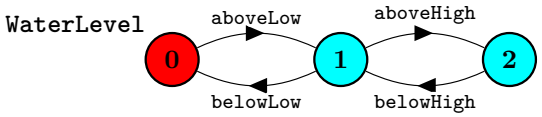

Figura 6: LTS que modela el nivel del agua dentro de la mina.

La semántica de un LTS $\mathcal{P}$ puede ser definida en termino de sus *ejecuciones*, es decir, el conjunto de secuencia de eventos que $\mathcal{P}$ puede ejecutar, comenzando en el estado inicial y siguiendo las transiciones que δ define.

Definición 2.16 (Ejecución)**.** *Sea $\mathcal{P} = \langle Q, A, \delta, q_0\rangle$ un LTS. Diremos que una secuencia (finita o infinita) de estados y eventos $\sigma =< s_0 > e_0 < s_1 > e_1 < s_2 > e_2 < s_3 > \ldots$ es una* ejecución *(o* traza*) de $\mathcal{P}$, si se cumple que: $s_0 = q_0$, y para todo i se cumple que $(s_i, e_i, s_{i+1}) \in \delta$.*

Usaremos además la notación $e_0e_1e_2\ldots$ para referirnos sólo a la secuencia de eventos involucrados en una ejecución del LTS. Si observamos el LTS de `WaterLevel` (Figura 6) vemos que los diferentes cambios en el nivel del agua pueden observarse en las diferentes ejecuciones posibles que define su LTS, como por ejemplo:

- $\sigma_1 =< 0 >$ `aboveLow` $< 1 >$ `belowLow` $< 0 >$ `aboveLow` $< 1 > \ldots$
- $\sigma_2 =< 0 >$ `aboveLow` $< 1 >$ `aboveHigh` $< 2 >$ `belowHigh` $< 1 >$ `belowLow` $< 0 > \ldots$

Los sistemas concurrentes usualmente cuentan con varios procesos o componentes que deben interactuar entre sí para lograr un objetivo común. La comunicación entre los diferentes procesos puede realizarse mediante *memoria compartida* o *envío de mensajes*. En general, esta comunicación debe ser cuidadosamente modelada ya que de ella depende el éxito para alcanzar el objetivo. Sin embargo, a nivel de modelado de máquinas de estado, nos abstraemos del mecanismo elegido para la sincronización de procesos. Sólo va a ser de nuestro interés los eventos compartidos entre los procesos involucrados en el sistema.

El comportamiento del sistema va a ser definido en términos del comportamiento de cada componente y las interacciones que se dan entre ellos. Es por esto que se introduce el concepto de *composición paralela*.

Definición 2.17 (Composición Paralela)**.** *Sean* $\mathcal{P} = \langle Q, A, \delta, q_0 \rangle$ *y* $\mathcal{P}' = \langle Q', A', \delta', q_0' \rangle$ *dos LTSs. La* ***composición paralela*** $\mathcal{P} \parallel \mathcal{P}'$ *define el LTS* $\mathcal{P} \parallel \mathcal{P}' = \langle Q \times Q', A \cup A', \Delta, (q_0, s_0') \rangle$*, tal que* Δ *es la mínima relación que cumple con lo siguiente, donde* $e \in A \cup A'$*:*

$$\frac{(s_1, e, s_2) \in \delta}{((s_1, t), e, (s_2, t)) \in \Delta}\ e \in A \setminus A' \qquad \frac{(t_1, e, t_2) \in \delta'}{((s, t_1), e, (s, t_2)) \in \Delta}\ e \in A' \setminus A$$

$$\frac{(s_1, e, s_2) \in \delta \quad (t_1, e, t_2) \in \delta'}{((s_1, t_1), e, (s_2, t_2)) \in \Delta}\ e \in A \cap A'$$

Intuitivamente, $\mathcal{P} \parallel \mathcal{P}'$ modela la ejecución asíncrona de ambos LTSs, pero sincronizando la ejecución de los eventos (o mensajes) *compartidos*. Es decir que cada componente puede ejecutarse independientemente del resto, mientras que sus transiciones no sean compartidas con algún otro componente.

Continuando con el ejemplo, recordemos que la bomba debe encenderse cuando detecta que el nivel del agua es alto y debe apagarse cuando el nivel es bajo. El simple LTS `Pump` que modela este comportamiento puede visualizarse en Figura 7.

$$\begin{aligned}\texttt{Pump} = \langle &\{0, 1, 2\}, \{\texttt{belowLow}, \texttt{aboveHigh}, \texttt{switchPumpOn}, \texttt{switchPumpOff}\},\\ &\{(0, \texttt{aboveHigh}, 1), (1, \texttt{switchPumpOn}, 0),\\ &(0, \texttt{belowLow}, 2), (2, \texttt{switchPumpOff}, 0)\}, 0\rangle\end{aligned}$$

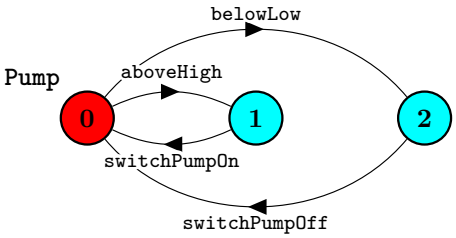

Figura 7: LTS que modela el comportamiento de la bomba en la mina.

La composición paralela `SYS` = `WaterLevel` $\parallel$ `Pump` modela el comportamiento de todo el sistema, es decir, los cambios en el nivel del agua y cómo la bomba reacciona ante esos cambios. La Figura 8 representa el LTS resultante de dicha composición.

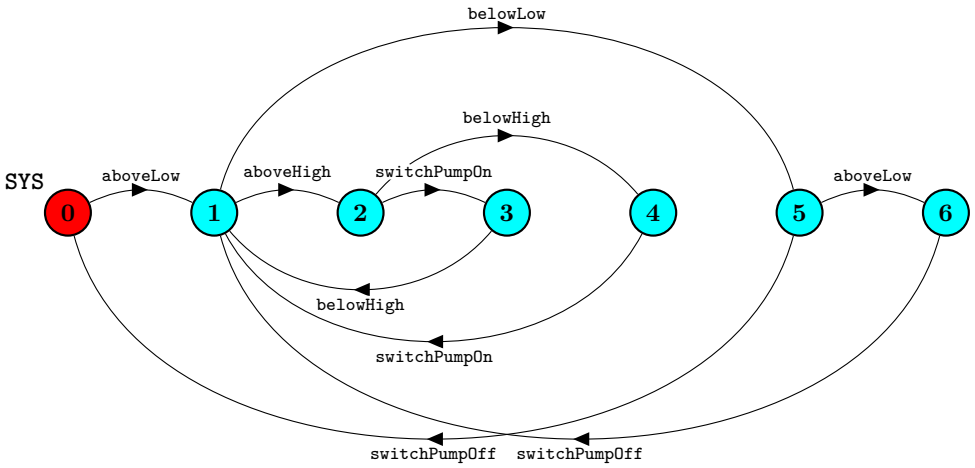

Figura 8: LTS que representa a **SYS** = **WaterLevel** ‖ **Pump**.

2.3.2 *Lenguajes para la descripción de Modelos de Comportamiento*

El modelado de sistemas reactivos mediante la manipulación directa de máquinas de estados, como los LTSs, se vuelve poco práctico a medida que el número de estados y transiciones de los componentes crece (como muestra la Figura 8). Surgieron así numerosos lenguajes para la especificación de sistemas reactivos, que cuentan con una sintaxis simple e intuitiva, además de una semántica formal, lo que los hace adecuados para el análisis automático. Muchas de la metodologías de ingeniería de requisitos (como KAOS y SCR, las utilizadas en esta tesis) adoptaron estos lenguajes para evaluar y mejorar la calidad de sus especificaciones.

En esta sección presentaremos tres lenguajes muy utilizados para modelar y analizar el comportamiento de sistemas reactivos, que además servirán de soporte para muchas de las técnicas desarrolladas en este trabajo: FSP, Promela y Action Language.

FSP*: Procesos de Estados Finitos*

Particularmente, los lenguajes basados en *álgebras de procesos* (o cálculos de procesos) han mostrado ser un formalismo adecuado para la descripción de sistemas que involucran interacción y sincronización de varios procesos. Entre los más destacados podemos mencionar al lenguaje CCS introducido por Robin

Milner [Milner, 1980] y CSP introducido por C. A. R. Hoare [Hoare, 1985], siendo éstos los que sentaron las bases de las álgebras de procesos.

En este trabajo utilizaremos FSP (Finite State Processes) [Magee and Kramer, 2006], un lenguaje introducido por Jeff Kramer y Jeff Magee, ampliamente adoptado en los últimos años para el análisis y diseño de sistemas concurrentes. Tál como lo explican sus creadores, la semántica de FSP se basa en la semántica de CCS, mientras que la sintáxis es muy similar a la del lenguaje CSP. Un punto muy importante es que toda expresión FSP puede ser automáticamente mapeada a un LTS finito y viceversa.

Una especificación FSP contiene básicamente un conjunto de definiciones de procesos *primitivos* y procesos *compuestos*. Los procesos son definidos utilizando las siguientes construcciones: `->` denota un prefijo de acción, `|` denota una elección, y elecciones condicionales son expresadas por medio de clausulas `when`. Además, los procesos pueden ser indexados y parametrizados, y pueden ser compuestos de forma secuencial (`;`) o en paralelo (`||`). En la Figura 9 se puede observar la especificación FSP del sistema que describe el comportamiento de la bomba en la mina.

```
range R = 0..2

WaterLevel = WaterLevel[0],
WaterLevel[i:R] = (when (i==0) aboveLow ->WaterLevel[1]
                 | when (i==1) belowLow ->WaterLevel[0]
                 | when (i==1) aboveHigh ->WaterLevel[2]
                 | when (i==2) belowHigh ->WaterLevel[1]).

Pump = (belowLow -> switchPumpOff ->Pump
       | aboveHigh -> switchPumpOn ->Pump).

||SYS = (WaterLevel || Pump).
```

Figura 9: Especificación FSP de la bomba en la mina.

Un prefijo de acción `x -> P` describe un proceso que ejecuta la acción `x` y luego pasa a comportarse como el proceso `P`. Por ejemplo, `aboveHigh -> switchPumpOn -> Pump` de la Figura 9, indica que al detectar que el nivel del agua es alto (`aboveHigh`), luego se prende la bomba ejecutando la acción `switchPumpOn`, y el proceso pasa a comportarse como `Pump`. Además, puede observarse que en la definición del proceso `Pump` se utiliza el operador de elección (`|`). Por otra parte, vemos que en la definición del proceso `WaterLevel` tenemos elecciones condicionales utilizando la guarda `when`. Más aún, utilizamos elección no determinista en el nivel medio de agua (condición `when (i==1)`) para modelar

que el nivel puede volverse bajo o alto de manera no determinista. `SYS` es el proceso paralelo que compone los procesos `WaterLevel` y `Pump`, mostrado en la Figura 8.

Labeled Transition System Analyzer (LTSA) es la herramienta presentada en [Magee and Kramer, 2006] que permite construir automáticamente LTSs a partir de especificaciones FSP. Además, LTSA facilita el análisis de propiedades que suelen ser de interés verificar sobre sistemas reactivos, como ausencia de *deadlocks* (el sistema no alcanza un estado de bloqueo) y *progreso* de ciertos eventos del sistema. En el Capítulo 4 veremos como podemos utilizar LTSA para analizar especificaciones de requisitos.

PROMELA

En este trabajo también haremos uso del lenguaje de especificación PROMELA (Protocol/Process Meta Language) [Holzmann, 1991], introducido por Gerard Holzmann. Promela provee una sintaxis simple para la descripción de procesos, muy similar a la sintaxis del lenguaje de programación C, por lo que en sus orígines fué rapidamente adoptado como lenguaje de modelado. En particular, Promela fue muy utilizado como una abstracción para el diseño de protocolos de red. Además, su adopción como lenguaje de especificación esta fuertemente asociado al éxito de Spin [Holzmann, 2004], el model checker utilizado para análizar este tipo de especificaciones (presentado a continuación en Sección 2.5).

Una especificación Promela consiste de declaraciones de tipos, canales, variables y procesos. Esta especificación se corresponde a un sistema de transiciones, usualmente *muy grande*, pero *finito*. Los procesos pueden crearse de manera dinámica, pueden comunicarse de manera síncrona o asíncrona, utilizando memoria compartida o por pasaje de mensajes. En la Figura 10 puede encontrar una simple especificación en Promela del sistema que controla la bomba en la mina. Puede observar que el proceso `WaterLevel` modela el ambiente y cómo el nivel del agua dentro de la mina se modifica de manera *no determinista*. El proceso `Pump` monitorea ese nivel de agua y controla el estado de la bomba: la prende cuando `(water > High)` y la apaga cuando `(water < Low)`. `init` es el proceso principal del sistema: a partir de él se pueden generar el resto de los procesos que van a ejecutarse de manera concurrente con la sentencia `run`. Puede encontrar una descripción detallada de la sintaxis Promela en [Holzmann, 1991]. En el Capítulo 5 veremos como es posible codificar una especificación de requisitos SCR en Promela, para luego poder analizarlas utilizando el model checker Spin.

```
#define Low 5
#define High 10
#define true 1
#define false 0

short water;
bool PumpOn;

proctype WaterLevel() {
  do
  :: water++;
  :: water--;
  :: skip;
  od
}

proctype Pump(){
  do
  :: (water < Low) -> PumpOn = false;
  :: (water > High) -> PumpOn = true;
  :: else -> skip;
  od
}

init{
  water = 0;
  run WaterLevel();
  run Pump();
}
```

Figura 10: Especificación PROMELA del comportamiento de la bomba en la mina.

ACTION LANGUAGE

ACTION LANGUAGE es un lenguaje especialmente diseñado para especificar sistemas reactivos [Bultan, 2000]. Una especificación en ACTION LANGUAGE consiste de un conjunto de acciones y módulos (representando componentes) que pueden ser compuestos usando operadores sincronos o asincronos. La semántica de cada módulo es dada en términos del Sistema de Transición que define. Su estado esta dado por los posibles valores de las variables que lo componen (booleanas, enteras o enumeradas) que pueden ser locales o compartidas con otros módulos. Luego, podemos construir expresiones lógicas utilizando las constantes `true` y `false`, y los operadores lógicos `!`, `and`, `or`, `=>` y `<=>`. Las expresiones que involucran variables enteras pueden utilizar además los operadores `+`, `-`, `>`, `<`, `<=`, `>=` y `=`.

Las *acciones* son expresiones que involucran *variables primadas*. Las variables sin primar se refieren al estado corriente, mientras que las variables primadas se refieren al *estado siguiente* de la variable (por ejemplo, `x'= x+1` indica que la variable `x` será incrementada en 1 en el siguiente estado). Las acciones pueden ser combinadas usando los operadores de composición sincrona `&` o asincrona `|`. Su semántica esta dada por la semántica de los operadores lógicos de la conjunción y disyunción, respectivamente.

En la Figura 11 puede observar una simple especificación en ACTION LANGUAGE del controlador de la bomba en la mina. Note que el comando `restrict` permite imponer restricciones sobre los posibles valores que pueden tomar las variables (por ejemplo, `water` tomará valores entre `0` y `5000`), y con `initial` capturamos las condiciones que deben cumplir los estados iniciales del sistema. Luego, definimos el Sistema de Transición para el componente `main`, componiendo las acciones que modifican el valor del nivel de agua y el estado de la bomba. En el Capítulo 5 veremos que una especificación de requisitos SCR puede ser traducida a una especificación en ACTION LANGUAJE, para luego poder analizarla utilizando el model checker ALV.

2.4 LÓGICAS TEMPORALES PARA LA ESPECIFICACIÓN DE LOS OBJETIVOS DEL SISTEMA

Las *Lógicas Modales* son una extensión a la Lógica Proposicional en donde se introducen modalidades, mediante operadores modales ($\Box$ y $\Diamond$), para formalizar las nociones de "necesidad" y "posibilidad" [Lewis, 1918; Blackburn et al., 2006]. A lo largo de su historia se han propuesto diversas variantes, como

```
module main()
  integer water;
  boolean PumpOn;

restrict: (0<=water) and (water<=5000) ;

initial: (water=0) and (!PumpOn) ;

main:
((water'=water) | (water'=water+1) | (water'=water-1))
&
(   (water'<10 & !PumpOn') | (water'>30 & PumpOn')
 | ( !(water'< 10) & !(water'> 30) & (PumpOn'=PumpOn) )
) ;

endmodule
```

Figura 11: Especificación Action Language de la bomba en la mina.

las Lógicas Epistémicas [von Wright, 1951a] y Lógicas Deónticas [von Wright, 1951b], variando la interpretación dada a los operadores modales.

Las *Lógicas Temporales* [Prior, 1957] son una variante de la *Lógica Modal* que permiten razonar sobre la validez de propiedades a lo largo de la ejecución del sistema. En este caso, las modalidades de la Lógica Temporal permiten expresar que una propiedad vale *siempre* ($\Box$) o en *algún momento* ($\Diamond$) de la ejecución del sistema. Notablemente, éstas lógicas ganaron gran aceptación como vehículo para especificar propiedades de sistemas reactivos que involucran paralelismo y concurrencia [Manna and Pnueli, 1992; Manna and Pnueli, 1995], por lo que muchas metodologías de ingeniería de requisitos comenzaron a usarlas para describir formalmente los objetivos del sistema a desarrollar [Dardenne et al., 1993; Heitmeyer et al., 2005].

Existen muchos tipos de lógicas temporales. Se diferencian básicamente por el modelo temporal que utilizan, es decir, como observan el paso del tiempo. Por ejemplo, en *Real-Time Temporal Logic* (RT-LTL) [Alur and Henzinger, 1994] el tiempo ocurrido entre los eventos es observable. Sin embargo, en otras lógicas sólo interesa el orden en que ocurren los eventos: en *Linear-Time Temporal Logic* (LTL) [Pnueli, 1977] la organización temporal es lineal (como una sóla ejecución) por lo que en cada estado tenemos un sólo estado futuro posible, mientras que en *Computational Tree Logic* (CTL) [Lamport, 1980] la organización temporal es ramificada modelando diferentes futuros posibles para cada instante.

A continuación describimos en detalle la Lógica Temporal LTL que será utilizada a lo largo de este trabajo y una extensión de la misma que aplica sobre descripciones de sistemas basados en eventos.

2.4.1 *Linear-Time Temporal Logic (LTL)*

Linear-Time Temporal Logic (LTL) fue introducida en el año 1977 por Amir Pnuli para la verificación de programas secuenciales y paralelos [Pnueli, 1977], y más adelante, junto a Zohar Manna, presentaron una metodología basada en esta lógica para la especificación y verificación de sistemas reactivos [Manna and Pnueli, 1992; Manna and Pnueli, 1995]. LTL cuenta con los operadores temporales $\bigcirc$ (*Next*) y $\mathcal{U}$ (*Until*) que permiten caracterizar propiedades sobre la ejecución del sistema, es decir, podemos referirnos a lo que puede suceder en el tiempo futuro.

Definición 2.18 (Sintaxis de fórmulas LTL)**.** *Sea PA un conjunto de proposiciones atómicas. Las valores de verdad T(verdadero) y F(falso), y toda $p \in PA$ son fórmulas LTL. Sean φ_1, φ_2 dos fórmulas de LTL, entonces las siguientes también son fórmulas LTL: $\neg\varphi_1$, $\varphi_1 \vee \varphi_2$, $\bigcirc\varphi_1$ y $\varphi_1 \mathcal{U} \varphi_2$.*

Como ya mencionamos, la organización temporal de LTL es lineal. Es decir, el tiempo es modelado como una secuencia infinita de estados en donde cada uno tiene un único sucesor, modelando cómo las proposiciones cambian de valor a lo largo de la ejecución del sistema.

Definición 2.19 (Interpretación LTL)**.** *Sea φ una fórmula LTL y PA^{φ} el conjunto de proposiciones incluidas en φ. Una interpretación LTL $\sigma = s_0, s_1, s_2, s_3 \ldots$, con $s_i \subseteq PA^{\varphi}$, es una "secuencia infinita" de modelos de la Lógica Proposicional. Denotaremos $\sigma(i) = s_i$ al i-ésimo estado de σ.*

Definición 2.20 (Semántica de fórmulas LTL)**.** *Sea φ una fórmula LTL y σ una interpretación LTL para φ. Diremos que φ es verdadera bajo σ, si y sólo si, $(\sigma, 0) \models_{LTL} \varphi$. Definimos ésta noción inductivamente de la siguiente manera:*

$$
\begin{array}{ll}
(\sigma, i) \models_{LTL} \mathsf{T} & \\
(\sigma, i) \not\models_{LTL} \mathsf{F} & \\
(\sigma, i) \models_{LTL} p & \textit{iff } p \in \sigma(i) \\
(\sigma, i) \models_{LTL} \neg\phi & \textit{iff } (\sigma, i) \not\models_{LTL} \phi \\
(\sigma, i) \models_{LTL} \phi_1 \vee \phi_2 & \textit{iff } (\sigma, i) \models_{LTL} \phi_1 \textit{ ó } (\sigma, i) \models_{LTL} \phi_2 \\
(\sigma, i) \models_{LTL} \bigcirc\phi & \textit{iff } (\sigma, i+1) \models_{LTL} \phi \\
(\sigma, i) \models_{LTL} \phi_1 \mathcal{U} \phi_2 & \textit{iff } \exists j : j \geq i : (\sigma, j) \models_{LTL} \phi_2 \\
 & \qquad \textit{y } \forall k : i \leq k < j : (\sigma, k) \models_{LTL} \phi_1
\end{array}
$$

Intuitivamente, la fórmula $\bigcirc\phi$ expresa que ϕ es válida en el *siguiente* estado, mientras que la fórmula $\phi_1 \mathcal{U} \phi_2$ indica que vale ϕ_1 *hasta que* vale ϕ_2. A pesar de que los operadores $\bigcirc$ y $\mathcal{U}$ brindan todo el poder expresivo de LTL, existen dos operadores derivados que probablemente sean los más utilizados en la práctica: $\Diamond$ (*Future*) y $\Box$ (*Globally*). La definición de estos operadores es la siguiente:

Sintaxis	Semántica
$\Diamond\varphi \equiv \mathsf{T}\mathcal{U}\varphi$	$(\sigma, i) \models_{LTL} \Diamond\varphi$ iff $\exists j : j \geq i : (\sigma, j) \models_{LTL} \varphi$
$\Box\varphi \equiv \neg\Diamond\neg\varphi$	$(\sigma, i) \models_{LTL} \Box\varphi$ iff $\forall j : j \geq i : (\sigma, j) \models_{LTL} \varphi$

El operador $\Diamond$ es utilizado para indicar que *en algún momento futuro* cierta propiedad va a ser válida, mientras que $\Box$ expresa que la propiedad debe valer *siempre*. En este trabajo también haremos uso de la versión débil del operador until $\mathcal{W}$ (*Weak Until*), definido como sigue: $\varphi_1 \mathcal{W} \varphi_2 \equiv (\Box\varphi_1) \vee (\varphi_1 \mathcal{U} \varphi_2)$. La Figura 12 muestra gráficamente el significado de los operadores temporales presentados anteriormente.

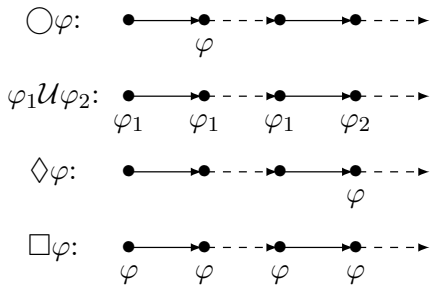

Figura 12: Representación gráfica de los operadores temporales.

2.4.2 *Fluent Linear-Time Temporal Logic (FLTL)*

Las Lógicas Temporales son más directamente aplicables para especificar propiedades de sistemas sobre especificaciones basadas en estados, es decir, cuando podemos referirnos a propiedades del estado del sistema. Dada la importancia de los formalismos basados en eventos, como los ya mencionados CSP, CCS y FSP, se han propuesto varios mecanismos para capturar propiedades de estados sobre sistemas basados en eventos. En particular, los *fluentes* son proposiciones definidas sobre un sistema basado en eventos, que se activan o desactivan acorde a la ejecución de los eventos del sistema [Giannakopoulou and Magee, 2003]. Miller y Shanahan informalmente definen a los fluentes como propiedades del mundo que varían en el tiempo [Miller and Shanahan, 1999]. *Fluent Linear-Time*

Temporal Logic (FLTL) es una variante de LTL especialmente desarrollada para especificar propiedades temporales sobre *sistemas basados en eventos*.

Definición 2.21 (Fluentes). *Sea $\mathcal{P} = \langle Q, A, \delta, q_0 \rangle$ un LTS. Formalmente, $Fl \equiv \langle I_{Fl}, T_{Fl}, B_{Fl} \rangle$ define un fluente Fl, donde $I_{Fl}, T_{Fl} \subseteq A$, $I_{Fl} \cap T_{Fl} = \emptyset$ y $B_{Fl} \in \{\mathsf{T}, \mathsf{F}\}$. B_{Fl} indica el valor inicial del fluente. Cuando un evento de I_{Fl} ocurre, Fl pasa ser verdadero (T), y se vuelve falso (F) cuando se ejecuta un evento de T_{Fl}. Se puede omitir B_{Fl} cuando inicialmente el fluente es falso.*

Definición 2.22 (Fluente asociado a un Evento). *Dado un LTS $\mathcal{P} = \langle Q, A, \delta, q_0 \rangle$, para cada evento $e \in A$ existe un único fluente que caracteriza la ejecución del evento e. Formalmente, el fluente asociado a cada evento $e \in A$ se define como: $Fl_e \equiv \langle \{e\}, A - \{e\}, \mathsf{F} \rangle$. Note que Fl_e se prende cuando se ejecuta e y se apaga cuando se ejecuta cualquier otro evento.*

Veamos algunos ejemplos de fluentes que pueden definirse sobre el LTS $\texttt{SYS} = \texttt{WaterLevel} \parallel \texttt{Pump}$ de la Figura 8.

$$
\begin{aligned}
\mathit{LowWater} &\equiv \langle \{\texttt{belowLow}\}, \{\texttt{aboveLow}\} \rangle \text{ initially } \mathsf{T} \\
\mathit{HighWater} &\equiv \langle \{\texttt{aboveHigh}\}, \{\texttt{belowHigh}\} \rangle \\
\mathit{PumpOn} &\equiv \langle \{\texttt{switchPumpOn}\}, \{\texttt{switchPumpOff}\} \rangle
\end{aligned}
$$

Note por ejemplo, que el fluente *PumpOn* inicialmente es falso indicando que la bomba esta apagada. Cuando se ejecuta el evento `switchPumpOn` el fluente pasa a ser verdadero (la bomba se prende) y vuelve a ser falso cuando ocurre `switchPumpOff` (la bomba se apaga).

Podemos pensar a los fluentes como estados abstractos del sistema, en el sentido de que no forman parte explícita del estado del mismo. Una fórmula FLTL es una fórmula LTL donde las proposiciones son los fluentes definidos sobre el sistema.

Definición 2.23 (Sintaxis de fórmulas FLTL). *Sea $\mathcal{D}$ un conjunto de fluentes. Luego, todo $Fl \in \mathcal{D}$ es una fórmula FLTL; si φ y ψ son fórmulas FLTL, entonces $\neg\varphi$, $\varphi \vee \psi$, $\bigcirc\varphi$ y $\varphi \mathcal{U} \psi$ también son fórmulas FLTL. Los operadores temporales $\Diamond$, $\Box$ y $\mathcal{W}$ se definen de la manera clásica.*

Para poder dar la semántica de fórmulas FLTL, primero definimos formalmente una función de valuación que determina el valor de cada fluente a lo largo de una traza.

Definición 2.24 (Valuación de Fluentes). *Sea $\mathcal{P}$ un LTS, $\mathcal{D}$ un conjunto de fluentes y Σ el conjunto de todas las trazas de $\mathcal{P}$. Luego, $V_{\mathcal{D}} : \Sigma \times \mathbb{N} \to \mathbb{P}(\mathcal{D})$*

es una función de valuación tal que, dado un fluente $Fl \in \mathcal{D}$*, una traza* $\sigma =< e_0, e_1, \ldots, e_i, \ldots >$ *y una posición* i *en la traza,* $Fl \in V(\sigma, i)$ *si y sólo si vale alguna de las siguientes condiciones:*

- $B_{Fl} = T \wedge \forall k \in \mathbb{N} : 0 \leq k \leq i : e_k \notin T_{Fl}$;
- $\exists j \in \mathbb{N} : j \leq i : ((a_j \in I_{Fl}) \wedge (\forall k \in \mathbb{N} : j \leq k \leq i : e_k \notin T_{Fl}))$.

Intuitivamente, la función de valuación indica que un fluente es verdadero en una posición i, si inicialmente era verdadero y no ocurrió ningún evento de terminación; o si en alguna posición j ocurrió un evento de inicialización, y desde j hasta i no ocurrió ningún evento de terminación. Considerando esta definición, podemos ahora definir formalmente la semántica de fórmulas FLTL.

Definición 2.25 (Semántica de fórmulas FLTL)**.** *Sea* $\mathcal{P}$ *un LTS,* D *un conjunto de fluentes,* φ *una fórmula FLTL y* $\sigma =< e_0, e_1, \ldots, e_i, \ldots >$ *una traza de* $\mathcal{P}$*. Diremos que* φ *es verdadera en* σ*, si y sólo si,* $(\sigma, 0) \models_{FLTL} \varphi$*, definida inductivamente de la siguiente manera:*

$$
\begin{array}{ll}
(\sigma, i) \models_{FLTL} Fl & \textit{iff } Fl \in V_D(\sigma, i) \\
(\sigma, i) \models_{FLTL} \neg\phi & \textit{iff } (\sigma, i) \not\models_{FLTL} \phi \\
(\sigma, i) \models_{FLTL} \phi_1 \vee \phi_2 & \textit{iff } (\sigma, i) \models_{FLTL} \phi_1 \textit{ ó } (\sigma, i) \models_{FLTL} \phi_2 \\
(\sigma, i) \models_{FLTL} \bigcirc\phi & \textit{iff } (\sigma, i+1) \models_{FLTL} \phi \\
(\sigma, i) \models_{FLTL} \phi_1 \mathcal{U} \phi_2 & \textit{iff } \exists j : j \geq i : (\sigma, j) \models_{FLTL} \phi_2 \\
 & \qquad y\ \forall k : i \leq k < j : (\sigma, k) \models_{FLTL} \phi_1
\end{array}
$$

2.4.3 *Especificación de los Objetivos del Sistema*

Notablemente, las lógicas temporales han sido exitosamente aplicadas a la especificación y verificación de propiedades sobre sistemas reactivos [Manna and Pnueli, 1992; Manna and Pnueli, 1995]. Esto motivó a que muchas metodologías de ingeniería de requisitos las adoptaran como un formalismo para especificar de manera precisa los objetivos que el sistema debe alcanzar. Se han utilizado variados tipos de lógicas dependiendo de las características de los requisitos a especificar: de tiempo real [Dardenne et al., 1993; Letier and van Lamsweerde, 2002b], de tiempo discreto [Letier et al., 2005], y de tiempo lineal como LTL [Kazhamiakin et al., 2004; Heitmeyer et al., 2005] y FLTL [Uchitel et al., 2007; Alrajeh et al., 2009].

En general, los objetivos especificados mediante lógicas temporales pueden clasificarse en dos categorías: *safety* y *liveness*. Los objetivos de *safety* expresan la idea de que "nada malo va a pasar". Usualmente, este tipo de objetivos tienen

la forma canónica $\Box\varphi$, para indicar que en toda ejecución del sistema vale la propiedad φ. Por otro lado, los objetivos de *liveness* expresan de que "algo bueno va a ocurrir". Este tipo de objetivos suelen representarse con la forma canónica $\Diamond\varphi$, indicando que en algún momento la propiedad φ va a ser válida. Puede encontrar una clasificación más detallada de las propiedades que podemos expresar en LTL en [Manna and Pnueli, 1992].

Continuando con el ejemplo de la bomba en la mina, veamos como podríamos formalizar en FLTL los objetivos que debe satisfacer el sistema:

$$PumpOnWhenHighWater = \Box(HighWater \rightarrow \bigcirc PumpOn)$$
$$PumpOffWhenLowWater = \Box(LowWater \rightarrow \bigcirc \neg PumpOn)$$

La propiedad $PumpOnWhenHighWater$ expresa que cada vez que el sistema detecte un nivel alto de agua, entonces en el siguiente estado la bomba debe encenderse. Por otro lado, la propiedad $PumpOffWhenLowWater$ indica que si el sistema detecta que el nivel del agua es bajo, entonces en el siguiente estado la bomba debe apagarse. Note que estos dos objetivos son propiedades de safety. Adicionalmente, estamos interesados en que frecuentemente el sistema mantenga el nivel del agua en niveles aceptables. Para lograr esto, entonces debe ocurrir que al encender la bomba, en algún momento futuro el nivel del agua baje. Este objetivo adicional se corresponde a una propiedad de liveness especificada de la siguiente manera:

$$ResponseNoHighWaterWhenPumpOn = \Box(PumpOn \rightarrow \Diamond\neg HighWater)$$

A continuación presentamos una técnica automática para verificar la validez de propiedades temporales especificadas sobre modelos de comportamiento. En el próximo capítulo veremos que las lógicas temporal son ampliamente utilizadas por las metodologías formales de ingeniería de requisitos para capturar los objetivos del sistema.

2.5 MODEL CHECKING

A fines de los 70s y principios de los 80s, trabajos como [Pnueli, 1977; Owicki and Lamport, 1982] mostraron exitosamente que las lógicas temporales son un buen formalismo para especificar propiedades de interés en programas concurrentes (por ejemplo, exclusión mutua, ausencia de deadlocks, etc). Sin embargo, hasta ese momento debían de realizarse las pruebas manualmente para verificar si el sistema satisfacía las propiedades. Edmund M. Clarke y Allen Emerson fueron los primeros en introducir el concepto de *Model Checking* [Clarke and Emerson, 1981; Clarke et al., 1986; Clarke et al., 1999; Clarke, 2008].

Precisamente, dado un modelo M y una propiedad φ especificada en alguna lógica temporal (por ejemplo, LTL o FLTL), el problema de *Model Checking* consiste en verificar automáticamente si el modelo M satisface la propiedad φ.

El Algoritmo 2.2 muestra una versión simple del algoritmo clásico de Model Checking definido para la lógica temporal LTL. Inicialmente, el algoritmo comienza por construir un autómata de Büchi $\mathcal{B}(\neg\varphi)$ que representa la negación de la propiedad φ ([Kesten et al., 1993] presenta una técnica automática para construir dicho autómata). Luego, se chequea si hay intersección entre los comportamientos que define el modelo M y los definidos por el autómata $\mathcal{B}(\neg\varphi)$. Finalmente, si la intersección no es vacía, entonces hemos encontramos un *contraejemplo* para φ, es decir, una ejecución del modelo que viola la propiedad. En caso de que la intersección sea vacía, entonces no hay violaciones a la propiedad, por lo tanto decimos que el modelo M satisface φ ($M \models \varphi$). A lo largo de los años, este algoritmo ha sufrido diversas modificaciones para ajustarse a las diversas representaciones de los modelos (por ejemplo, estados explícitos o simbólicos) y las nuevas lógicas que fueron surgiendo.

Algorithm 2.2 Model Checking

```
1: function ModelChecking(M: Modelo, φ: formula LTL)
2:     B(¬φ) ←LTLtoBuchi(¬φ)
3:     X ← Trazas(M)∩ Trazas(B(¬φ))
4:     if X ≠ ∅ then
5:         σ ←ExtraerContraejemplo(X)
6:         return Propiedad Inválida (σ)
7:     else
8:         return Propiedad Verificada
```

Las herramientas que implementan este algoritmo (o alguna variante) automáticamente se conocen como *Model Checkers*. Existen una gran cantidad de Model Checkers diferentes que varían en la forma en que representan el modelo del sistema (estados concretos o abstractos), el lenguaje para describir el sistema (lenguajes de programación, lenguajes de modelado como FSP y Promela) y la lógica empleada para la especificación de la propiedades esperadas del sistema (LTL, FLTL, CTL, etc.). En este trabajo utilizaremos el model checker LTSA para analizar sistemas descritos en el lenguaje de modelado FSP, Spin para analizar modelos Promela, y ALV para verificar especificaciones descriptas en Action Language.

2.5.1 *LTSA: Labelled Transition System Analyser*

LTSA es una herramienta que permite realizar diferente tipos de análisis automáticos sobre sistemas descriptos en el lenguaje de modelado FSP [Magee and Kramer, 2006]. En particular, LTSA nos permite realizar animaciones y verificar la ausencia de *deadlocks* en el sistema, así como también el *progreso* de ciertos eventos. Sin embargo, la principal característica de LTSA es su poderoso algoritmo de Model Checking. LTSA provee una sintaxis simple para definir fluentes, por lo que podemos especificar propiedades en FLTL y LTSA se encarga de verificar su validez sobre el modelo descripto en FSP. En nuestro caso, describiremos modelos operacionales de los requisitos en FSP, y especificaremos los objetivos en FLTL para luego poder analizarlos automáticamente utilizando LTSA.

El algoritmo de model checking que implementa LTSA es similar al Algoritmo 2.2. Primero obtiene un LTS por cada proceso FSP definido. Luego, los compone para obtener el LTS que representa el comportamiento completo del sistema. Este LTS contiene todos los estados y transiciones alcanzables que el sistema puede ejecutar. Dada la propiedad a analizar especificada en FLTL, LTSA genera un LTS que representa la *negación* de la propiedad y lo compone con el LTS del sistema. Finalmente, LTSA verifica si el *estado de error* es alcanzable desde el estado inicial. En caso de serlo, LTSA retorna un contrajemplo, es decir, una ejecución posible del sistema que viola la propiedad. En caso de que el estado de error no sea alcanzable, entonces decimos que la propiedad es válida.

Es posible especificar propiedades de safety en forma de procesos mediante el comando `property`. Básicamente, un proceso anotado con `property` describe el orden en que ciertos eventos deben de ejecutarse para que una traza sea considerada segura. Si LTSA encuentra una traza que viola ese orden, será reportada como un contraejemplo a la propiedad. Por otro lado, el comando `progress` nos permite especificar que cierto conjunto de eventos deben ejecutarse *infinitas* veces en toda ejecución del sistema.

En lo que respecta a este trabajo, para especificar propiedades de safety y liveness utilizaremos la lógica temporal FLTL mediante el comando `assert` que brinda LTSA. La sintaxis que provee LTSA para especificar cada operador lógico y temporal es la siguiente: `!` (negación), `&&` (conjunción), `||` (disyunción), `->` (implicación), `X` (*Next*), `[]` (*Globally*), `<>` (*Future*), `U` (*Until*) y `W` (*Weak Until*).

Considerando el modelo FSP de la Figura 9 que describe el comportamiento de la bomba dentro de la mina, las propiedades FLTL presentadas en la sección anterior, pueden ser especificadas en LTSA de la siguiente manera:

```
fluent PumpOn = <switchPumpOn, switchPumpOff>
fluent LowWater = <belowLow, aboveLow> initially 1
fluent HighWater = <aboveHigh, belowHigh>

assert PumpOnWhenHighWater = [](HighWater ->X PumpOn)
assert PumpOffWhenLowWater = [](LowWater ->X !PumpOn)

assert ResponseNoHighWaterWhenPumpOn = [](PumpOn -><>!HighWater)
```

2.5.2 *Spin: Simple Promela INterpreter*

Spin [Holzmann, 2004] es un model checker de *estados explícitos* con el cuál podemos analizar diversas propiedades sobre sistemas descriptos en PROMELA, como ausencia de deadlocks, validez de invariantes (aserciones) o cualquier fórmula LTL (safety y liveness). Cabe destacar que Spin implementa el algorírmo clásico de model checking descripto anteriormente en Algoritmo 2.2.

En particular, `assert(P)` es una sentencia de Promela que al ejecutarse Spin verifica que la expresión `P` evalúe a verdadero. En caso de evaluar a falso, Spin finaliza el análisis y retorna un contraejemplo. Si Spin logra construir todo el espacio de estados y la expresión `P` nunca evaluó a falso, entonces la propiedad ha sido verificada. Usualmente la sentencia `assert` se usa para especificar invariantes del sistema, es decir, propiedades que queremos que valgan en todos los estados alcanzables.

Promela además nos permite especificar propiedades LTL que luego serán analizadas por Spin. La sintaxis textual para especificar la fórmula LTL es la misma que provee LTSA. Para incluir la propiedad LTL al modelo Promela, debemos seguir la siguiente notación: `ltl [ name ] ‘{’ formula_LTL ‘}’`.

Considerando el modelo PROMELA de la Figura 10, veamos como podemos especificar en fórmulas LTL las propiedades que debe cumplir el controlador de la bomba dentro de la mina:

```
ltl PumpOnWhenHighWater { [] ((water > High) -> PumpOn) }
ltl PumpOffWhenLowWater { [] ((water < Low) -> !PumpOn) }
ltl ResponseNoHighWaterWhenPumpOn =
                    { [](PumpOn -> <> (water < High)) }
```

2.5.3 *ALV: Action Language Verifier*

ALV [Bultan and Yavuz-Kahveci, 2001; Yavuz-Kahveci et al., 2005] es un model checker *simbólico* de estados infinitos que permite analizar especificaciones descriptas en ACTION LANGUAGE. ALV utiliza técnicas de model checking simbólico para verificar propiedades especificadas en CTL [Lamport, 1980] (aunque en este trabajo sólo usaremos la directiva `invariant(`φ`)` que se corresponde a la fórmula CTL $AG(\varphi)$). Debido a que nuestra especificación puede tener variables enteras sin límites, no hay garantías de que el computo de punto fijo converja, por lo que ALV implementa diferentes heurísticas para tratar de lograr convergencia. Para las expresiones aritméticas, ALV soporta una representación Poliedra y otra basada en Autómatas, y para las expresiones booleanas y enumeradas utiliza BDDs. Estas representaciones son combinadas para caracterizar simbólicamente el estado del sistema.

Considere la especificación en ACTION LANGUAGE de la Figura 11, podemos agregarle las propiedades que queremos verificar con ALV utilizando la directiva `spec` de la siguiente manera:

```
// PumpOnWhenHighWater
spec: invariant((water > 30) => PumpOn);

// PumpOffWhenLowWater
spec: invariant((water < 10) => !PumpOn) ;

// ResponseNoHighWaterWhenPumpOn
spec: AG(PumpOn -> AF(water < 30)) ;
```

2.6 RESUMEN

En este capítulo presentamos, principalmente, la sintaxis y semántica de las diferentes lógicas que vamos a utilizar en esta tesis para especificar formalmente los requisitos de software. Introducimos además, los mecanismos automáticos de análisis para estas lógicas, que serán combinados por nuestras técnicas para resolver problemas específicos de la Ingeniería de Requisitos. Por último, presentamos brevemente la sintaxis de las herramientas subyacentes que utilizaran las técnicas aquí desarrolladas.

3

Fundamentos y Lenguajes para la Especificación de Requisitos de Software

3.1 Introducción

Previamente, en el Capítulo 1, presentamos y motivamos la necesidad de contar con metodologías sistemáticas para capturar, elaborar, especificar y validar requisitos de software. Más aún, mencionamos diversas técnicas informales y (semi-)formales que buscan extraer los requisitos desde la descripción del cliente, y mostramos las ventajas de elegir un lenguaje formal para la especificación de los mismos. Luego, en el Capítulo 2, introducimos los formalismos básicos que serán utilizados por los lenguajes de requisitos que utilizaremos en esta tesis. Más precisamente, presentamos las lógicas temporales como el lenguaje declarativo para la especificación formal de requisitos, y los LTSs para el modelado operacional del comportamiento del sistema, junto a poderosos mecanismos de análisis automáticos asociados a estos formalismos.

A continuación en la Sección 3.2, presentamos dos modelos conceptuales que nos permiten formular claramente las nociones fundamentales de la Ingeniería de Requisitos: *El Mundo y La Máquina* y *El Modelo de Cuatro Variables*. Luego, en la Sección 3.3 motivamos brevemente la necesidad la utilizar un lenguaje formal para especificar requisitos de software. Finalmente, presentamos dos metodologías para el proceso de ingeniería de requisitos muy utilizadas en el ámbito académico y aplicadas exitosamente en la práctica: el *Método KAOS* en la Sección 3.4, y el *Método SCR* en la Sección 3.5.

3.2 Fundamentos de la Ingeniería de Requisitos

Antes de introducir los lenguajes de especificación y metodologías utilizadas en esta tesis, es esencial comprender las nociones básicas de la Ingeniería de Requisitos. Para esto, nos centraremos en dos modelos conceptuales muy simples

que permiten caracterizar los requisitos del sistema: *El Mundo y La Máquina* presentado por Jackson y Zave [Jackson, 1995; Zave and Jackson, 1997] y *El Modelo de Cuatro Variables* introducido por Parnas y Madey en [Parnas and Madey, 1995]. Estos enfoques no sólo permiten estructurar y organizar los requisitos de manera clara, sino que además brindan los fundamentos necesarios para validación de los requisitos especificados. Presentaremos además la terminología empleada a lo largo de este trabajo.

Jackson y Parnas establecieron una importante distinción entre las *propiedades de dominio* (llamadas indicativas en [Jackson, 1995] y NAT en [Parnas and Madey, 1995]) y los *objetivos del sistema* (llamados optativos en [Jackson, 1995] y REQ en [Parnas and Madey, 1995]). Las *propiedades de dominio* son sentencias *descriptivas* que se asumen verdaderas sin importar el comportamiento del sistema. Usualmente, capturan restricciones naturales dadas por leyes físicas y políticas organizacionales. Por otro lado, los *objetivos del sistema*, o simplemente *objetivos*, son sentencias *prescriptivas* que expresan propiedades deseadas del sistema, cuyos componentes buscarán forzar su validez. Llamaremos *requerimiento* a un tipo especial de objetivo que debe ser satisfecho pura y exclusivamente por un componente de software. La distinción entre sentencias descriptivas y prescriptivas es fundamental en el proceso de ingeniería de requisitos. Los requerimientos y objetivos pueden ser modificados, fortalecidos o debilitados, mientras que las propiedades del dominio permanecen estáticas. Note que para mantener una terminología uniforme a lo largo de la tesis, realizamos unos pequeños cambios a la terminología de Jackson y Zave. Por ejemplo, en su terminología, los objetivos son llamados requerimientos, y un requerimiento es llamado especificación.

La Figura 13 muestra el modelo de *El Mundo y La Máquina* presentado por Jackson y Zave [Jackson, 1995; Zave and Jackson, 1997], formalizado más adelante en [Gunter et al., 2000]. En este enfoque, el *mundo* caracteriza la parte problemática del mundo real que deseamos mejorar, mientras que la *máquina* denota el software/hardware que deberemos desarrollar para solucionar dicho problema. Los fenómenos compartidos representan la interacción entre el mundo y la máquina. La máquina *monitorea* algunos eventos compartidos con el mundo y *controla* otros, lo que nos permite definir un límite sobre cuál parte será automatizada y cuál no. En general, los objetivos se formulan en términos que los stakeholders pueden comprender (los fenómenos del mundo), mientras que los requerimientos se formulan en un vocabulario mas comprensible por los desarrolladores del software.

El modelo de Jackson y Zave nos permite formular dos criterios de validación. Por un lado, los requerimientos R de la máquina deben satisfacer los objetivos O, dadas ciertas asunciones D sobre el dominio ($R, D \models O$). Por otro lado,

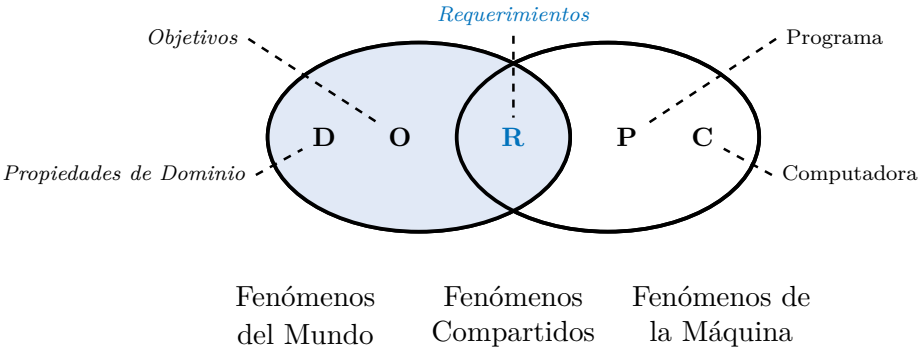

Figura 13: El Mundo y La Máquina.

debe ocurrir que el programa P ejecutando sobre una computadora C debe satisfacer los requerimientos R ($P, C \models R$). Además, debemos garantizar que los requerimientos sean consistentes con las propiedades del dominio, es decir, $R, D \not\models False$.

El *Modelo de Cuatro Variables* de Parnas y Madey [Parnas and Madey, 1995] describe el comportamiento esperado del sistema en términos de relaciones sobre cuatro conjuntos de variables: variables *monitoreadas* y *controladas*, y los datos de entrada y salida (ver Figura 14). Similar al modelo de Jackson y Zave, las variables monitoreadas son valores del estado del ambiente que influyen en el comportamiento del sistema, mientras que las controladas son valores que el sistema pretende controlar. Este enfoque introduce dos conjuntos adicionales de variables ambientales. Los datos de entrada representan los valores reales que toma el software, leídos por los dispositivos de entrada. Los datos de salida en cambio, representan los valores computados por el software, que serán plasmados en el ambiente por medio de los dispositivos de salida.

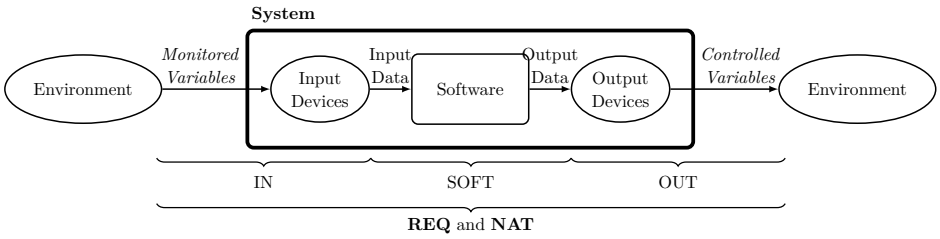

Figura 14: El Modelo de Cuatro Variables.

En el Modelo de Cuatro Variables, la especificación del comportamiento esperado del sistema es dado en términos de dos relaciones, REQ y NAT, desde las variables monitoreadas a las controladas. La relación NAT describe las restricciones naturales impuestas sobre el sistema y posibles asunciones, similar a

las propiedades de dominio mencionadas anteriormente. Por otro lado, la relación REQ describe las restricciones entre variables monitoreadas y controladas que el sistema debería inducir. Podemos pensar que REQ representa los objetivos y requerimientos del enfoque anterior. En [Gunter et al., 2000] puede encontrar una comparación más detallada entre los dos enfoques presentados.

La relación IN entre variables monitoreadas y datos de entrada describe el comportamiento de los dispositivos de entrada, mientras que la relación OUT entre datos de salida y variables controladas describe el comportamiento de los dispositivos de salida. Mientras que, el comportamiento del software es descripto por la relación SOFT entre los datos de entrada y salida. Para validar y verificar que el software implementa correctamente los requerimientos, debemos chequear si se cumple que: $\text{NAT} \wedge \text{IN} \wedge \text{SOFT} \wedge \text{OUT} \Rightarrow \text{REQ}$.

Ambos enfoques proveen modelos simples para explicar en que consiste en proceso de ingeniería de requisitos, por lo que fueron adoptados por muchos lenguajes formales para la especificación de requisitos, entre los cuales se encuentran los métodos KAOS y SCR utilizados en esta tesis.

3.3 LENGUAJES FORMALES PARA LA ESPECIFICACIÓN DE REQUISITOS

Es ampliamente aceptado que la depuración es una de las etapas más costosas en el desarrollo de software, y que los errores son más fáciles (y menos costosos) de corregir si se capturan lo más temprano posible en el proceso de desarrollo [Boehm and Papaccio, 1988]. Luego, el descubrir inconsistencias, errores de comprensión e imprecisiones durante el proceso de ingeniería de requisitos es de fundamental importancia práctica y económica en la actividad de producción de software [Ghezzi et al., 2002; Sommerville, 2006; Jalote, 2005]. Para esto es necesario contar con técnicas de análisis rigurosas que nos brinden ciertas garantías sobre la calidad de la especificación de requisitos.

Los lenguajes informales, en particular, las notaciones diagramáticas como los DFDs [Stevens et al., 1974; DeMarco, 1979], los casos de uso [Cockburn, 2000; Maiden and Alexander, 2004], y los diagramas de secuencias de mensajes [Rumbaugh et al., 1999] han tenido gran aceptación para la descripción de requisitos. Estas notaciones han demostrado ser de gran utilidad para interactuar con los stakeholders en la etapa de captura de requisitos, y además nos brindan los primeros pasos de la documentación. Sin embargo, debido a la informalidad de este tipo de notaciones, no son en general analizables automáticamente más allá de cuestiones sintácticas propias de cada notación.

En la sección anterior presentamos dos enfoques conceptuales que resaltan la importancia de identificar y distinguir entre las asunciones que hacemos sobre el ambiente y aquellas propiedades que queremos alcanzar, para lo cuál desarrollaremos el software/harware necesario. Estos enfoques proveen mecanismos de validación y verificación que nos permiten evaluar la calidad de nuestra especificación de requisitos. Las metodologías formales de ingeniería de requisitos utilizan los lenguajes formales, como las lógicas temporales presentadas en Capítulo 2, para formalmente especificar las *propiedades del dominio* y los *objetivos* esperados del sistema. Contar con un lenguaje con semántica formal, nos permite alcanzar una mayor precisión en la descripción de los requisitos y eliminar absolutamente las ambigüedades propias de la notación. Más aún, podemos utilizar los diferentes mecanismos de análisis automáticos asociados a estos lenguajes formales, para desarrollar formas de validación y verificación más sofisticadas que pueden ser completamente automatizadas por herramientas. Sin embargo, sólo una semántica formal no es suficiente: las complejidades y sutilezas de los requisitos, y el grado de detalle propio de las notaciones formales demandan formas de modularizar especificaciones, de manera de colaborar con los desarrolladores en la comprensión y la manipulación para el análisis. Esto es crucial para que una notación formal sea prácticamente utilizable.

A continuación, presentamos dos metodologías que se centran en diferentes etapas del proceso de ingeniería de requisitos, que brindan los mecanismos necesarios para modularizar, organizar y analizar especificaciones de requisitos. En la Sección 3.4, presentamos el *Método KAOS* [Dardenne et al., 1993], una de las metodologías orientadas a *objetivos* que se enfoca en la *etapa temprana* de la ingeniería de requisitos donde se exploran los objetivos deseados del cliente, y se diseñan y evalúan diferentes posibilidades para satisfacerlos. En la Sección 3.5 presentamos el *Método SCR*, una metodología aplicada típicamente sobre sistemás de seguridad crítica, por lo que en general se utiliza en la *etapa tardía* de la ingeniería de requisitos para realizar estrictos análisis de ambigüedad, inconsistencia e incompletitud a las especificaciones de requisitos.

3.4 EL MÉTODO KAOS

La sigla *KAOS* proviene de Knowledge Acquisition in autOmated Specifications. En general, el método KAOS es utilizado en la *etapa temprana* de la ingeniería de requisitos, por lo que provee no sólo un lenguaje de especificación, sino también un conjunto de estrategias para elaborar estos requerimientos y asistencia automática para guiar el proceso de captura de requisitos [Dardenne et al., 1993]. En KAOS los *objetivos* juegan un rol fundamental: son muy útiles para negociación con los stakeholders, facilitan una organización y diseño de

la especificación, son adecuados para el análisis, documentación y evolución del sistema [van Lamsweerde, 2001]. En esta sección introduciremos algunos conceptos referentes al lenguaje de especificación que KAOS provee, el cuál será utilizado en este trabajo como formalismo para la especificación de requisitos de software. Para una lectura más detallada sobre los restantes características del método, referirse a [van Lamsweerde, 2009].

Una especificación KAOS captura múltiples vistas del sistema, tal como puede apreciarse en la Figura 15. El *Modelo de Objetivos* define una descomposición de los objetivos, que indica básicamente como los objetivos de alto nivel son alcanzados en términos de sub-objetivos de más bajo nivel, y los conflictos que pueden existir entre estos. El *Modelo de Objetos* define las entidades, relaciones y atributos que son relevantes a cada objetivo. En el *Modelo de Responsabilidad* se identifican los agentes involucrados, sus interfaces y sus responsabilidades con respecto a los objetivos. Finalmente, el *Modelo Operacional* describe las operaciones que son provistas por los agentes, para garantizar la satisfacción del objetivo del que son responsables.

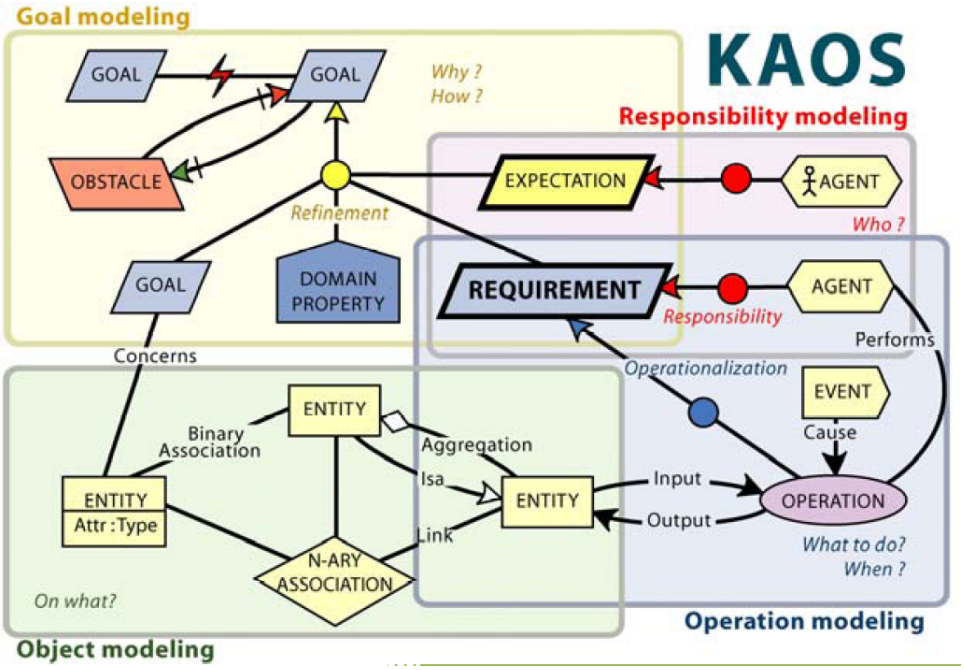

Figura 15: Múltiple vistas de una especificación KAOS.

3.4.1 *Modelo de Objetivos*

Como explicamos en la sección Sección 3.2, los *objetivos* son propiedades prescriptivas que esperamos el sistema satisfaga, requiriendo usualmente la cooperación de múltiples agentes para su satisfacción. Los objetivos pueden referirse desde *requisitos funcionales* ("la bomba debe controlar el nivel del agua en la mina") a *requisitos no funcionales*, como seguridad, calidad o costos del servicio ("reducir los costos en seguridad"). A su vez, estos pueden ser más abstractos, considerados como *objetivos de alto nivel* ("prevenir de una inundación en la mina") o más técnicos y precisos, es decir, *objetivos de bajo nivel* ("cuando el nivel del agua sea alto y no haya metano en el ambiente, entonces la bomba debe encenderse").

El Modelo de Objetivos organiza y relaciona los objetivos mediante refinamientos AND/OR, en el cuál se muestra como los objetivos de alto nivel pueden alcanzarse en base a otros más simples (ver Figura 16). Un refinamiento AND relaciona un objetivo padre con sub-objetivos, expresando que la satisfacción de los sub-objetivos garantiza la satisfacción del objetivo padre. Un refinamiento OR modela diferentes alternativas para satisfacer el objetivo padre, mostrando varios refinamientos posibles. El refinamiento de objetivos termina cuando los objetivos de bajo nivel pueden ser alcanzados por un único agente, el cual debe tener la capacidad de monitorear y controlar lo necesario para satisfacer el objetivo del cuál es responsable [Letier and van Lamsweerde, 2002b]. En particular, un *requerimiento* es un objetivo de bajo nivel que debe ser garantizado por un único agente de software, mientras que una *expectativa* es un objetivo bajo la responsabilidad de un único agente (no controlable) del ambiente.

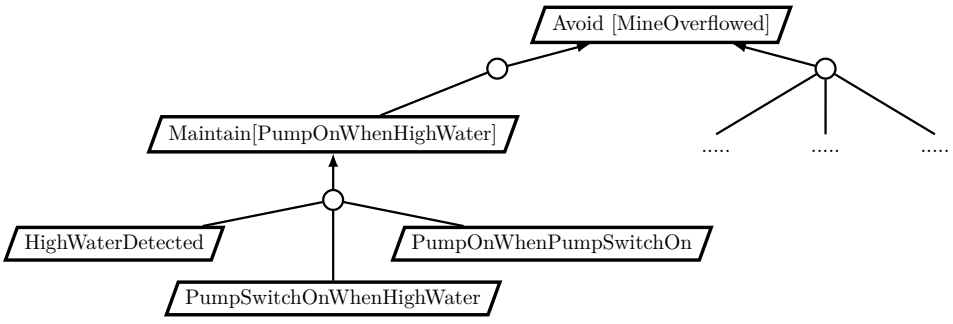

Figura 16: Modelo de Objetivos.

La Figura 16 muestra que una posibilidad es refinar el objetivo de alto nivel `Avoid[MineOverflowed]` en uno más preciso `Maintain[PumpOnWhenHighWater]`. Esto indica que para prevenir la inundación de la mina, lo que el sistema debe hacer es mantener la bomba encendida mientras el nivel del agua sea

alto. Esto puede lograrse si se satisfacen los siguientes tres objetivos de bajo nivel: `HighWaterDetected`, `PumpSwitchOnWhenHighWater` y `PumpOnWhenPumpSwitchOn`. Intuitivamente, si el dispositivo ambiental detecta que el nivel de agua es alto, entonces nuestro controlador debe reaccionar y tratar de encender la bomba utilizando el interruptor, para que luego la bomba (dispositivo ambiental) efectivamente se encienda. Note que los objetivos `HighWaterDetected` y `PumpOnWhenPumpSwitchOn` son expectativas sobre el ambiente, ya que capturan el comportamiento esperado de los dispositivos ambientales. Por otro lado, el objetivo `PumpSwitchOnWhenHighWater` es un requerimiento, es decir, el objetivo que debe ser satisfecho por nuestro software a construir.

En KAOS, cada objetivo cuenta con un nombre, una definición en lenguaje natural, y de manera opcional, una definición formal utilizando lógicas temporales, como es el caso de LTL y FLTL (ver Sección 2.4). Por ejemplo, asumamos que la proposición *HighWater* indica que "el nivel del agua en la mina es alto" y *PumpOn* que "la bomba está encendida". Luego, podemos especificar el objetivo `Maintain[PumpOnWhenHighWater]` de la siguiente manera:

Goal Maintain [PumpOnWhenHighWater]
InformalDef The pump shall be on when the water level is too high
FormalDef $\Box(HighWater \rightarrow PumpOn)$
Refines `Avoid [MineOverflowed]`
RefinedTo
`HighWaterDetected`,
`PumpSwitchOnWhenHighWaterDetected`,
`PumpOnWhenPumpSwitchOn`

Para aquellos usuarios que no poseen un buen manejo de lógicas temporales, KAOS ofrece *patrones de especificación*. Estos patrones son *syntactic sugar* para expresar fórmulas LTL, es decir, cada patrón puede ser traducido a una fórmula LTL para su análisis. A continuación mostramos los patrones principales propuestos en [Dardenne et al., 1993], algunos de los cuales aparecerán a lo largo de este trabajo:

- Achieve$[C \Rightarrow \Diamond P]$: Siempre que la condición corriente C valga, entonces en algún momento futuro valdrá la propiedad P.
- Cease$[C \Rightarrow \Diamond \neg P]$: Siempre que la condición corriente C valga, entonces en algún momento futuro no valdrá la propiedad P.
- Maintain$[C \Rightarrow P]$: Siempre que la condición corriente C valga, también debe valer la propiedad P.
- Avoid$[C \Rightarrow \neg P]$: Siempre que la condición corriente C valga, no debe valer la propiedad P.

La notación $A \Rightarrow B$ equivale a la fórmula LTL $\Box(A \rightarrow B)$. Existen variantes de estos patrones que tratan de dar más precisiones sobre en qué momento futúro debe valer de la propiedad P:

- ImmediateAchieve$[C \Rightarrow \bigcirc P]$: La propiedad P debe valer inmediatamente después de que C vale.
- BoundedAchieve$[C \Rightarrow \Diamond_{\leq d} P]$: Desde el momento en que la condición C vale, la propiedad P debe valer en menos de d unidades de tiempo.

Contar con una descripción formal de los objetivos, nos brinda la posibilidad de realizar diversos análisis cobre el Modelo de Objetivos. Por ejemplo, podemos chequear *consistencia* (los sub-objetivos de un mismo padre no pueden ser contradictorios), *completitud* (la satisfacción de los sub-objetivos debe ser suficiente para satisfacer el objetivo padre), y *minimalidad* (si uno de los sub-objetivos no es considerado, luego la satisfacción del objetivo padre no puede ser garantizada) [Dardenne et al., 1993]. En este trabajo consideraremos objetivos especificados en LTL, pero nos centraremos en otro tipo de análisis. En particular, los objetivos guiarán a nuestra técnica de refinamiento para la operacionalización de objetivos.

3.4.2 *Modelo de Objetos*

Un Modelo de Objetos define las entidades del dominio, las relaciones y atributos que son relevantes para especificar los objetivos a alcanzar. Este modelo incluye un glosario de términos que puede ser acordado junto a los diferentes stakeholders, y permitirá eliminar posibles ambigüedades referenciales. Además, durante la etapa de diseño, el Modelo de Objeto proveerá las bases para generar los esquemas de las bases de datos (si son necesarias) y para elaborar la arquitectura del sistema.

Hay varios tipos de objetos que pueden ser capturados en un Modelo de Objetos. Una *entidad* es un objeto pasivo autónomo, cuya existencia no depende de otros objeto, pero no puede modificar el estado de otros objetos. Una *asociación* es una relación de dependencia entre dos objetos, en la cuál cada objeto tiene un rol particular. Un *agente* es un objeto *activo* capaz de ejecutar operaciones. Un agente puede ser un componente de software, hardware, una persona, etc. Su comportamiento es definido en términos de los objetos que monitorean y controlan sus operaciones, y de los objetivos de los cuales es responsable. Un *evento* puede verse como un objeto instantáneo que relaciona

dos estados, especificando sus atributos y las asociaciones en las que puede ocurrir dicho evento.

Por ejemplo, respecto al objetivo `PumpSwitchOnWhenHighWater` especificado anteriormente, `HighWater` es un atributo de la entidad `Mine`, mientras que `Switch` es un objeto de la entidad `Pump`. Además, la relación entre las entidades `Mine` y `Pump` está dada por la asociación *HasPump(m, p)*. Este simple Modelo de Objetos puede observarse en la Figura 17. Además, KAOS permite especificar formalmente invariantes sobre el dominio de los objetos. Como puede observarse, indicamos que no puede ocurrir que el nivel del agua sea bajo y alto al mismo tiempo.

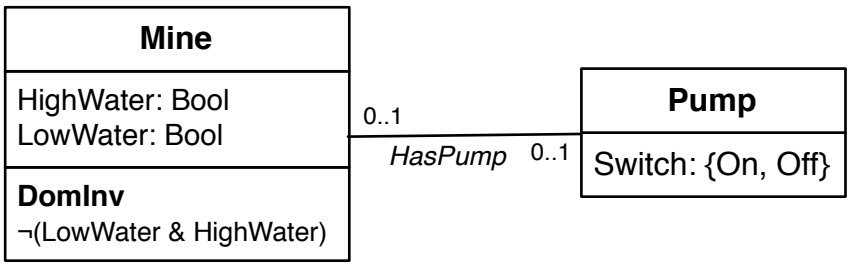

Figura 17: Modelo de Objetos.

3.4.3 *Modelo de Responsabilidad*

Este modelo define las responsabilidades e interfaces de los agentes que van a intervenir en el sistema. Un *agente* es un componente activo (humano, software, hardware, etc.) que cumple un rol particular en la satisfacción de los objetivos. Podemos pensar a un agente como un componente que ejecuta acciones bajo ciertas condiciones para poder garantizar la satisfacción de los objetivos de los cuáles es responsable. Tal como mencionamos anteriormente, el refinamiento de objetivos finaliza cuando cada objetivo de bajo nivel es asignado a un único agente responsable. Los objetivos asignados a agentes ambientales se llaman *expectativas*, los asignados a componentes de software se llaman *requerimientos*.

Continuando con el ejemplo, la Figura 18 muestra los agentes y sus responsabilidades en pos de lograr la satisfacción del objetivo de alto nivel `Avoid [MineOverflowed]`. En particular, el agente de software `MinePumpController` será responsable de satisfacer el objetivo `PumpSwitchOnWhenHighWater`.

Para poder satisfacer el objetivo, el agente de software debe contar con las capacidades necesarias para monitorear y controlar los valores de los cuáles

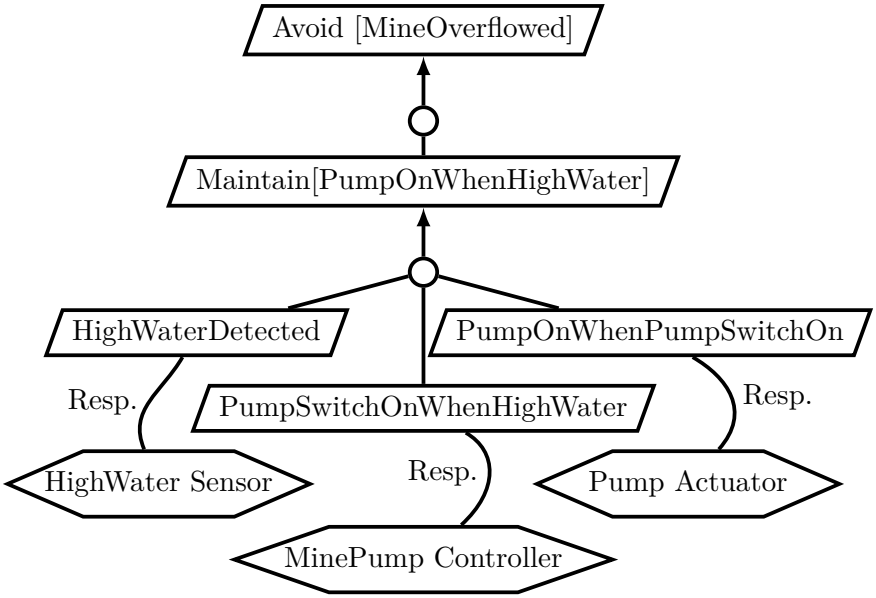

Figura 18: Modelo de Responsabilidades.

depende el objetivo [Letier and van Lamsweerde, 2002b]. Por ejemplo, el agente de software `MinePumpController` deberá de poder monitorear el nivel del agua (mediante las señales de `HighWater` y `LowWater`) y controlar el estado de la bomba mediante el interruptor (`Switch`) del objeto `Pump`.

3.4.4 *Modelo Operacional*

Los agentes ejecutan operaciones para controlar los valores de los que son responsables, cuya combinación debería de ser tal que los objetivos se satisfagan. Una *operación* es una relación entrada-salida entre componentes del modelo de objetos, caracterizando una transición de estados en el sistema. Para cada operación se especifica su signatura (variables de entrada y salida) y su *precondición*, *postcondición* y condición de *triggering* (es decir, condición de disparo). Se hace una importante distinción entre las condiciones de *dominio* (*DomPre* y *DomPost*), y las condiciones *requeridas*, que capturan condiciones extras para lograr satisfacer los objetivos. Las condiciones de dominio son *descriptivas* (capturan el comportamiento natural de la operación), mientras que las condiciones requeridas son *prescriptivas* [Zave and Jackson, 1997; Dardenne et al., 1993]. Existen tres tipos de condiciones requeridas:

- una precondición *requerida* (*ReqPre*) captura *permiso*: bajo esta condición la operación *puede* ser ejecutada,

- una condición de *triggering* requerida (*ReqTrig*) expresa *obligación*: bajo esta condición la operación *debe* ejecutarse,
- y una postcondición requerida (*ReqPost*) captura efectos adicionales que produce la ejecución de la operación.

La precondición de dominio es un predicado de estado sobre el cuál se ejecuta la operación; las postcondiciones de dominio y requeridas predican sobre el estado siguiente alcanzado luego de ejecutar la operación, aunque también pueden referirse al estado corriente en el que se ejecuta; y las pre/triggering condiciones requeridas predican sobre el estado corriente con posibles referencias a estados anteriores (usando operadores LTL de pasado).

El proceso de *operacionalización de objetivos* consiste en *prescribir* las condiciones pre/triggering/post requeridas de manera tal que los objetivos sean satisfechos (más detalles en la Sección 4.3). Por ejemplo, supongamos que nuestro agente de software `MinePumpController` debe operacionalizar el objetivo previamente especificado `PumpSwitchOnWhenHighWater`. Luego, en la Figura 19, mostramos el Modelo Operacional necesario para garantizar la satisfacción de este objetivo.

```
Operation switchPumpOn
  Input   c: MinePumpController
  Output  c: MinePumpController/PumpOn
  DomPre  ¬ c.PumpOn
  DomPost c.PumpOn
  ReqTrig for PumpSwitchOnWhenHighWater
     c.HighWater

Operation switchPumpOff
  Input   c: MinePumpController
  Output  c: MinePumpController/PumpOn
  DomPre  c.PumpOn
  DomPost ¬ c.PumpOn
  ReqPre for PumpSwitchOnWhenHighWater
     ¬ c.HighWater
```

Figura 19: Modelo Operacional del `MinePumpController` para satisfacer el objetivo `PumpSwitchOnWhenHighWater`.

La Figura 19 muestra que la signatura de estas operaciones se corresponden al objeto `MinePumpController`. En particular, la salida es restringida al atributo `PumpOn`, es decir que las operaciones sólo pueden *controlar* el valor del

interruptor de la bomba. Cuando la signatura sea simple y esté entendida en el contexto (como en este caso que tenemos sólo el objeto `MinePumpController`), evitaremos mencionar el objeto de la signatura en las condiciones. Por ejemplo, usaremos la condición `PumpOn` para referirnos al atributo `c.PumpOn` del objeto `c:MinePumpController`.

Note que la operación `switchPumpOn` posee una condición de triggering requerida, indicando que cuando el controlador detecta un nivel alto de agua, debe obligadamente ejecutarse para encender la bomba. Por otro lado, la operación `switchPumpOff` tiene una precondición requerida, prohibiendo su ejecución cuando el nivel del agua es alto. Ambas condiciones requeridas son necesarias para garantizar el objetivo `PumpSwitchOnWhenHighWater`.

Semántica del Modelo Operacional

Es posible dar una semántica formal del Modelo Operacional, en términos de aserciones LTL, tal como fue presentado en [Letier and van Lamsweerde, 2002b]. Básicamente, las aserciones LTL indican cuales son las ejecuciones permitas y obligadas de cada operación del Modelo Operacional, a lo largo de la ejecución del sistema.

Definición 3.1 (Semántica de Condiciones de Dominio)**.** *Sea op una operación del Modelo Operacional,* $i_1, \ldots, i_n$ *sus argumentos de entrada y* $o_1, \ldots, o_m$ *los argumentos de salida de op. El predicado* $[\![op]\!]$ *representa la ejecución de la operación en el estado corriente, es decir, relaciona un par de estados que satisfacen la precondición y postcondición de dominio de la operación.*

$$[\![op]\!](i_1, \ldots, i_n, o_1, \ldots, o_m) \Leftrightarrow DomPre(op) \wedge \bigcirc DomPost(op)$$

Recuerde que $A \Leftrightarrow B \equiv \Box(A \leftrightarrow B)$ y $A \Rightarrow B \equiv \Box(A \rightarrow B)$. Note que la ejecución de la operación *op* requiere la validez de la precondición de dominio en el estado corriente, y luego de ejecutarse, la postcondición de dominio debe valer en el estado siguiente. De manera recíproca, la validez de estas condiciones en una transición de estados, se corresponde a una ejecución de la operación *op*. KAOS además garantiza que cualquier argumento diferente a los declarados $(i_1, \ldots, i_n, o_1, \ldots, o_m)$ permanecen sin modificaciones luego de su ejecución. Cuando las entradas y salidas $i_1, \ldots, i_n, o_1, \ldots, o_m$ estén dadas por entendidas, no serán mencionadas en las condiciones para simplificar la explicación.

Veamos cuáles son las aserciones LTL que definen la ejecución de las operaciones que hemos introducido anteriormente:

$$[\![\texttt{switchPumpOn}]\!] \Leftrightarrow \neg\texttt{PumpOn} \wedge \bigcirc\texttt{PumpOn} \quad (1)$$

$$[\![\texttt{switchPumpOff}]\!] \Leftrightarrow \texttt{PumpOn} \wedge \bigcirc\neg\texttt{PumpOn} \quad (2)$$

Note que ésta semántica permite que varias operaciones puedan ejecutarse de manera *concurrente* entre dos estados, siempre y cuando satisfagan sus pre/post condiciones de dominio (no es el caso de las operaciones `switchPumpOn` y `switchPumpOff`, ya que sus pre/post condiciones son contradictorias). Esto demuestra que KAOS posee una **semántica síncrona**. La elección de este tipo de semántica facilita la definición de la semántica de las condiciones de triggering, debido a su obligación inmediata de ser ejecutadas [Letier and van Lamsweerde, 2002b].

Definición 3.2 (Semántica de Condiciones Requeridas)**.** *Sea op una operación del Modelo Operacional. Definimos la semántica de las condiciones requeridas de la siguiente manera:*

(i) $[\![ReqPre(op)]\!] \stackrel{def}{=} [\![op]\!] \Rightarrow ReqPre(op)$,

(ii) $[\![ReqTrig(op)]\!] \stackrel{def}{=} ReqTrig(op) \wedge DomPre(op) \Rightarrow [\![op]\!]$

(iii) $[\![ReqPost(op)]\!] \stackrel{def}{=} [\![op]\!] \Rightarrow \bigcirc ReqPost(op)$

Intuitivamente, la condición *(i)* expresa que si la operación se ejecutó, es porque su precondición requerida era válida. La condición *(ii)* captura la obligación de ejecutar la operación cuando la condición de triggering y la precondición de dominio son válidas. Y la condición *(iii)* indica los efectos de la postcondición requerida después de ejecutar la operación.

La semántica de la condición de triggering requerida que agregamos a la operación `switchPumpOn` anteriormente, es la siguiente:

$$\texttt{HighWater} \wedge \neg\texttt{PumpOn} \Rightarrow [\![\texttt{switchPumpOn}]\!]$$

Esta aserción LTL asegura que cuando el nivel del agua sea alto y la bomba este apagada, entonces la operación `switchPumpOn` debe ejecutarse, así en el siguiente estado la bomba estará encendida. Por otro lado, veamos la semántica de la precondición requerida de la operación `switchPumpOff`:

$$[\![\texttt{switchPumpOff}]\!] \Rightarrow \neg\texttt{HighWater}$$

Esta aserción LTL asegura que si se ejecuta `switchPumpOff`, entonces el nivel del agua no puede ser alto.

En caso de que la operación *op* tenga múltiples condiciones requeridas, consideraremos que:

- $ReqPre(op) \equiv R_1 \wedge \ldots \wedge R_n$, para toda precondición requerida R_i,
- $ReqTrig(op) \equiv T_1 \vee \ldots \vee T_m$, para toda condición de triggering requerida T_i,
- $ReqPost(op) \equiv P_1 \wedge \ldots \wedge P_k$, para toda post condición requerida P_i.

Note que para que una operación pueda ejecutarse debe satisfacer todas sus precondiciones requeridas; mientras que la validez de alguna de las condiciones de triggering, fuerza la ejecución de la operación.

Para que una operación sea *consistente*, debemos evitar contradicciones entre sus condiciones de triggering requeridas y sus precondiciones requeridas. Es decir, queremos que siempre que la operación esté obligada a ejecutarse, entonces también debe estar habilitada a hacerlo. Para esto, cada operación del Modelo Operacional debe garantizar la siguiente meta-regla de KAOS:

$$ReqTrig(op) \wedge DomPre(op) \Rightarrow ReqPre(op)$$

Finalmente, la semántica del Modelo Operacional se define en términos de las condiciones de dominio y requeridas de cada operación.

Definición 3.3 (Semántica Modelo Operacional)**.** *Sea $O = \{op_1, \ldots, op_n\}$ un Modelo Operacional. Su semántica esta dada por la semántica de cada una de sus operaciones.*

$$[\![O]\!] = \bigwedge_{op_i \in O} ([\![ReqPre(op_i)]\!] \wedge [\![ReqTrig(op_i)]\!] \wedge [\![ReqPost(op_i)]\!])$$

Dada ésta caracterización formal en aserciones LTL, veremos más adelante en la Sección 4.3, como podemos automáticamente verificar si un Modelo Operacional operacionaliza correctamente un conjunto de objetivos.

3.4.5 *Modelo de Comportamiento*

Los modelos presentados anteriormente modelan de manera *implícita* el comportamiento del sistema: el Modelo de Objetivos provee una descripción declarativa del comportamiento esperado; el Modelo Operacional se enfoca en transiciones de un paso entre entradas-salidas; el Modelo de Objetos estructura las variables bajo estas transiciones; mientras que el Modelo de Responsabilidades indica qué agente es responsable de ejecutar dichas operaciones. El *Modelo de Comportamiento* complementa estas nociones haciendo *explícito* el comportamiento dinámico del sistema mediante una representación basada en

eventos. Utilizando alguna notación basada en escenarios, es posible dar una descripción del comportamiento específico de un agente en ciertas circunstancias, o podemos caracterizar el comportamiento del sistema como un todo utilizando máquinas de estado, como los LTSs [van Lamsweerde, 2009].

En esta tesis estamos particularmente interesados en el Modelo de Comportamiento que se obtiene a partir de un conjunto de objetivos operacionalizados. Básicamente, a partir de un Modelo Operacional, es posible construir de forma sistemática un Modelo de Comportamiento siguiendo [Letier et al., 2008]. Los requerimientos pueden expresarse utilizando la lógica temporal FLTL (Subsección 2.4.2), combinando los operadores $\Box$ y $\bigcirc$, para capturar las condiciones requeridas. Luego, se puede construir automáticamente un LTS (Subsección 2.3.1) que caracteriza el comportamiento del sistema. Usando esta caracterización, se puede utilizar un model checker (por ejemplo, LTSA introducido en la Subsección 2.5.1) para verificar si el modelo operacional satisface los objetivos, si son expresados como fórmulas FLTL. Veremos los detalles de este proceso en la Subsección 4.3.1.

3.5 EL MÉTODO SOFTWARE COST REDUCTION (SCR)

Cuando el sistema a desarrollar contiene componentes lógicos complejos, como los sistemas de seguridad crítica o tolerantes a fallas, es necesario brindar mayores garantías de calidad sobre la especificación de requisitos. Para estos casos, no sólo es necesario contar con metodologías rigurosas para la elaboración y especificación de requisitos, sino que además necesitamos poderosos mecanismos de análisis asociados a estas metodologías. En particular, el Método *Software Cost Reduction* (SCR) [Heitmeyer et al., 1996] fue desarrollado y aplicado exitosamente por el Naval Research Laboratory de los Estados Unidos para la elaboración, especificación y análisis de varios sistemas críticos, incluyendo el programa operacional de vuelo del *A-7 aircraft* [K. Heninger and Shore, 1978], un sistema de comunicaciones submarinas [Heitmeyer and McLean, 1983], y componentes de seguridad críticos de la planta nuclear de Darlington en Canada [van Schouwen et al., 1993].

El Método SCR está basado en el Modelo de Cuatro Variables [Parnas and Madey, 1995], por lo que la especificación del comportamiento esperado del sistema es dado en términos de las relaciones NAT (propiedades del dominio) y REQ (objetivos del sistema) desde las variables monitoreadas a las controladas (ver Sección 3.2). SCR provee un lenguaje *formal* para especificar los requisitos de software mediante *notaciones tabulares*. Las notaciones tabulares han probado ser un medio útil para describir de manera concisa y formal expresiones que

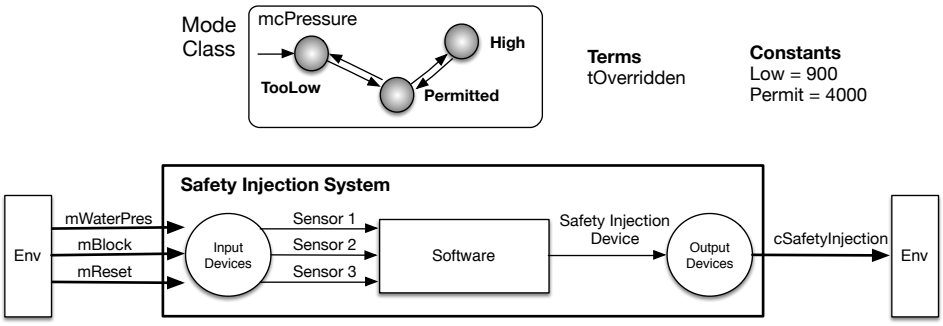

Figura 20: Modelo de Cuatro Variables para el *Safety Injection System*.

caracterizan requerimientos complejos. Las tablas imponen una *estructura* sobre las especificaciones, que ayudan a organizar fórmulas grandes y complejas en fórmulas mas chicas y bien estructuradas, que resultan más fáciles de comprender.

Para presentar la notación SCR, veamos a continuación una versión simple del *Safety Injection System* (SIS) [Courtois and Parnas, 1993], cuyo principal tarea es controlar el nivel de presión del agua en el sistema de refrigeración de una planta nuclear, tal como se muestra en Figura 20. El sistema es capaz de monitorear la presión del agua y el estado de dos interruptores manejados por el usuario: uno para bloquear la inyección del líquido refrigerante, y el otro para reactivar el sistema (si previamente había sido bloqueado). Estas mediciones ambientales son representadas en SCR mediante las variables monitoreadas `mWaterPres`, `mBlock` y `mReset`, respectivamente. Por otro lado, cuando la presión del agua es "demasiado baja" (nivel de gran peligrosidad), el sistema debe activar el sistema de inyección segura para restablecer el nivel normal de presión. Esto puede ser modelado en SCR utilizando una variable controlada `cSafetyInjection` que indica si el sistema de inyección de líquido refrigerante esta encendido (`On`) o apagado (`Off`).

Los estados de una especificación SCR son clasificados en *modos* pertenecientes a una *clase de modo* (en SCR pueden haber más de una clase de modo, indicando diferentes clasificaciones de los estados). Intuitivamente, los modos definen una partición de los estados del sistema, que típicamente se utilizan para capturar información histórica de la ejecución del sistema. Para el caso del SIS, la especificación incluye una única clase de modo `mcPressure`, que contiene tres modos: `TooLow`, `Permitted` y `High`. Cada modo es una representación abstracta de la variable monitoreada `mWaterPres`: `TooLow` indica que la presión del agua esta por debajo de la constante `Low`, `High` si la presión esta por encima de la constante `Permit`, y el modo `Permitted` indica que la presión esta en el medio de las dos constantes. En todo momento, el sistema debe estar en alguno de

estos modos. El término `tOverridden` es una variable auxiliar que indica si el sistema de inyección ha sido bloqueado.

Es importante mencionar que en SCR todas las variables son tipadas. Una variable de modo solo puede tomar valores de su clase de modo, mientras que las variables monitoreadas, controladas y los términos pueden ser de tipo numérico, booleano o enumerado. Por ejemplo, las variables monitoreadas `mBlock` y `mReset` son de tipo enumeradas con posibles valores `On` y `Off`, mientras que `mWaterPres` es una variable entera que puede asumir los valores entre `[0..5000]`. El término `tOverridden` es de tipo booleano, mientras que la variable controlada `cSafetyInjection` es de tipo enumerada con posibles valores `On` y `Off`. En SCR también es posible definir constantes. En el SIS, las constantes `Low`(900) y `Permit`(4000) representan valores particulares de la variable `mWaterPres`, que indican los límites para ingresar al modo `TooLow` (si `mWaterPres < Low`), y al modo `High` (si `mWaterPres > Permit`), respectivamente.

En SCR, el sistema reacciona ante *eventos de entrada* (cambio detectado en alguna variable monitoreada) que ocurren no-determinísticamente en el ambiente, y propaga los cambios a los términos y variables controladas acorde al comportamiento del sistema definido mediante *tablas* (que describen la relación REQ del Modelo de Cuatro Variables). A continuación, presentamos la sintaxis y semántica formal de los componentes que forman una especificación SCR [Heitmeyer et al., 1996], utilizando el Safety Injection System como ejemplo introductorio.

3.5.1 *Sintaxis y Semántica de SCR*

Sistema

Un sistema $\Sigma = (S, S_0, E^m, T)$ es una 4-upla, donde S es el conjunto de estados, $S_0 \subseteq S$ es el conjunto de estados iniciales, E^m es el conjunto de *eventos de entrada* (cambio en una variable monitoreada), y $T : E^m \times S \to S$ es una función parcial llamada función de *transformación* del sistema. SCR hace una asunción importante sobre este sistema llamada *One-Input Assumption*, que expresa que en cada transición de estado del sistema sólo ocurre un evento de entrada, es decir, sólo puede cambiar una variable monitoreada a la vez.

Estados del sistema

En SCR hay cuatro tipos de entidades básicas: las clases de modos, los términos, y las variables monitoreadas y controladas, como ya vimos en el

ejemplo del SIS. Una especificación SCR define *clases de modos* $M_1, \ldots, M_N$, donde cada M_i representa un conjunto de elementos llamados *modos*. Un modo no puede a pertenecer a más de una clase, es decir, los conjuntos $M_1, \ldots, M_N$ deben ser disjuntos. Como las variables en SCR son tipadas, llamaremos TY a la función que asigna a cada variable su tipo. Así, si m es una clase de modo, luego $TY(m)$ retorna el conjunto de modos posibles representados por m (M_m la clase de modo de m). Si v es un término o variable monitoreada/controlada, entonces $TY(v)$ retorna el tipo de v. Por ejemplo, en la especificación para el SIS:

$$
\begin{aligned}
&TY(\texttt{mcPressure}) = \{\texttt{TooLow}, \texttt{Permitted}, \texttt{High}\},\\
&TY(\texttt{mWaterPres}) = \texttt{[0..5000]},\\
&TY(\texttt{mBlock}) = \{\texttt{On}, \texttt{Off}\},\\
&TY(\texttt{mReset}) = \{\texttt{On}, \texttt{Off}\},\\
&TY(\texttt{tOverridden}) = \{\texttt{true}, \texttt{false}\},\\
&TY(\texttt{cSafetyInjection}) = \{\texttt{On}, \texttt{Off}\}
\end{aligned}
$$

Luego, un estado s de una especificación SCR es una función total que mapea cada variable v a un valor en $TY(v)$. Denotaremos $s(v)$ al valor de la variable v en el estado s. Así, en cualquier estado s, el sistema está exactamente en un un modo de cada clase de modo, y cada variable tiene un único valor. Podemos entonces, definir una función $MD : S \to (M_1 \times ... \times M_n)$ que retorna el modo del estado s. Por ejemplo, considere el siguiente estado del SIS:

$$
\begin{aligned}
s = \{&\texttt{mcPressure} \mapsto \texttt{High}, \texttt{mWaterPres} \mapsto 4000, \texttt{mBlock} \mapsto \texttt{On}, \texttt{mReset} \mapsto \texttt{Off},\\
&\texttt{tOverridden} \mapsto \texttt{True}, \texttt{cSafetyInjection} \mapsto \texttt{Off}\}
\end{aligned}
$$

Luego, para este estado s, como SIS sólo tiene la clase de modo `mcPressure`, $MD(s) = \texttt{High}$.

Condiciones

Las condiciones son expresiones lógicas definidas sobre las variables y se refieren a un sólo estado del sistema. Las condiciones simples pueden ser *true*, *false*, ó $v \sim c$, donde v es una variable, $\sim \in \{=, \neq, >, \geq, <, \leq\}$, y $c \in TY(v)$ un valor constante. Además, podemos combinar condiciones simples usando operadores lógicos: $\wedge$ (`AND`), $\vee$ (`OR`) and $\neg$ (`NOT`).

Eventos

En SCR, la noción de cambio de estado está dada por medio de *eventos*. En particular, `@T(c)` denota un *evento*, donde `c` es una condición simple. Intuitivamente, este evento indica que la expresión `c` es falsa en el estado corriente y se vuelve verdadera en el estado siguiente. Un *evento condicionado* `@T(c) WHEN d` es similar al evento anterior, pero además requiere que la condición `c` sea verdadera en el estado corriente sobre el cuál se va a ejecutar dicho evento. La notación `@F(c)` es usada para denotar el evento `@T(¬c)`. Además, es posible indicar que cierta variable `v` a cambiado su valor mediante el evento `@C(v)`. Considere los siguientes dos ejemplos de eventos sobre el SIS: el evento condicionado `@T(mBlock=On) WHEN mReset=Off` indica que `mBlock` cambia de `Off` a `On` cuando `mReset` es `Off` en el estado corriente; y el evento `@C(mWaterPres)` expresa que la presión del agua cambia de valor.

Para un estado s, denotaremos $\mathtt{c}[s]$ a la expresión que indica que la condición `c` vale en el estado s. Luego, dados el estado corriente s y el estado siguiente s', podemos definir la semántica de los eventos mediante las siguientes expresiones lógicas:

$$\mathtt{@T(c)\ WHEN\ d} = \neg \mathtt{c}[s] \wedge \mathtt{c}[s'] \wedge \mathtt{d}[s] \tag{3}$$

$$\mathtt{@C(v)} = s'(\mathtt{v}) \neq s(\mathtt{v}) \tag{4}$$

Diremos que un evento ocurre si la condición lógica que éste evento representa evalúa a verdadero. Además, diremos que un evento `@T(c) WHEN d` está *habilitado* a ser ejecutado en s, si $(\neg \mathtt{c} \wedge \mathtt{d})[s]$ evalúa a verdadero ($\neg \mathtt{c}[s]$ en caso del evento `@T(c)`).

El sistema modelado en SCR reacciona antes los cambios detectados en el ambiente utilizando las variables monitoreadas. Los cambios de las variables monitoreadas son capturados mediante un tipo especial de eventos, llamados *eventos de entrada*. Básicamente, un evento de entrada tiene la forma `@T(mV = c)`, donde `mV` es una variable monitoreada y `c` es un valor del tipo de `mV`. Luego, en cualquier punto de la ejecución, cualquier evento de entrada *habilitado* puede ser ejecutado. Es decir, el comportamiento de los eventos de entrada es *no-determinista*. Luego, las tablas de la especificación SCR indican como estos cambios se propagan a las variables restantes (modos, términos y controladas) para obtener un nuevo estado.

El comportamiento de las variables monitoreadas en SCR pueden ser pensadas como un sistema de transición de estados, definiendo el rango de los valores permitidos para la variable, su valor inicial, y una relación de transición que describe las modificaciones permitidas sobre la variable. Por ejemplo, el rango de la variable monitoreada `mBlock` del SIS es $\{\mathtt{On}, \mathtt{Off}\}$, inicialmente es `Off`, y

su relación de transición es definida como $\{(\texttt{On}, \texttt{Off}), (\texttt{Off}, \texttt{On})\}$. Esta última indica que `mBlock` puede cambiar de `On` a `Off`, y viceversa. En SCR es muy común que la relación NAT -describiendo restricciones ambientales- limite los cambios permitidos sobre las variables monitoreadas (sobre todo las variables numéricas). Por ejemplo, para el SIS, la relación de transición de la variable monitoreada `mWaterPres` es:

$$\{(x, x') | x' \neq x \wedge -10 \leq x' - x \leq 10 \wedge x, x' \in \{0 \dots 5000\}\}$$

donde $\{0 \dots 5000\}$ en el rango de `mWaterPres`, y la restricción $-10 \leq x' - x \leq 10$ es impuesta por NAT, estableciendo que `mWaterPres` no puede variar en más de 10 unidades entre dos observaciones consecutivas de la variable.

Si $\Sigma = (S, S_0, E^m, T)$ es un sistema SCR, definamos ahora formalmente el conjunto E^m, que contiene todos los eventos de entrada de la especificación SCR. E^m es el conjunto de todas las tuplas $(\texttt{mV}, v, v')$, tal que `mV` es una variable monitoreada y (v, v') pertenecen a la relación de transición definida sobre `mV`. Intuitivamente, el evento $(\texttt{mV}, v, v')$ representa un cambio de la variable `mV` desde el valor v al valor nuevo v'. Note que para que este evento este habilitado en un estado s, debe ocurrir que $s(\texttt{mV}) = v$.

Dependencias de variables

Para computar el nuevo estado, las tablas que definen una variable pueden depender del valor (corriente o siguiente) de otras variables. Decimos que una entidad r_i depende de otra r_j, si el valor de r_j es utilizado para computar el valor de r_i. Dependiendo del tipo de tabla utilizado para definir la variable r_i (ver la siguiente sección), puede que r_i utilice los valores corrientes o siguientes de r_j (por ejemplo, las tablas que involucran eventos consideran ambos valores).

Sea r_i, r_i' el valor corriente y siguiente, respectivamente, de la variable r_i. Luego, toda especificación SCR tiene dos relaciones de dependencias D_{new} y D_{old} entre las variables, definidas como: $(r_i, r_j) \in D_{new}$ si r_i' depende de r_j', y $(r_i, r_j) \in D_{old}$ si r_i' depende de r_j. La clausura transitiva de la relación D_{new}, denotada como D_{new}^+, debe ser necesariamente un orden parcial, para evitar dependencias circulares. Sea $D = D_{new} \cup D_{old}$, llamaremos a D la relación de dependencia de la especificación SCR. A modo de ejemplo, veamos la relación de dependencia del SIS:

$D =$ { (`mcPressure`, `mWaterPres`), (`mcPressure`, `mWaterPres'`),
(`tOverridden`, `mcPressure`), (`tOverridden`, `mcPressure'`), (`tOverridden`, `mBlock`),
(`tOverridden`, `mBlock'`),(`tOverridden`, `mReset`), (`tOverridden`, `mReset'`),
(`cSafetyInjection`, `mcPressure`), (`cSafetyInjection`, `tOverridden`) }

Tal como se puede observar, la variable de modo `mcPressure` depende del valor corriente y siguiente de la variable monitoreada `mWaterPres`. El término `tOverridden` depende de los interruptores manejados por el usuario y del modo del actual y siguiente del sistema. Mientras que, la variable controlada `cSafetyInjection` depende del modo corriente del sistema y del término `tOverridden` que indica si el sistema esta bloqueado o no.

Tablas SCR

Como hemos mencionado, en SCR las tablas definen los valores de las variables controladas, términos y clases de modo, en términos de las variables de las que dependen. Sea T_i la tabla que define la entidad r_i, y $r_1, \ldots, r_k$ las entidades de las que depende r_i, es decir, $(r_i, r_j) \in D$ para $1 \leq j \leq k$. La semántica de T_i se define como una *función total* $F_i : TY(r_1), \ldots, TY(r_k) \rightarrow TY(r_i)$. Intuitivamente, cada vez que un evento de entrada ocurre en un estado s, la tabla T_i define exactamente qué valor le corresponde a la entidad controlada r_i en el nuevo estado s'. Como todas las tablas son funciones totales, aseguran que para cada evento de entrada y estado, exista un sólo estado siguiente. La asignación de valores a la entidad r_i en la tabla T_i depende de una única clase de modo M. Durante el resto de esta sección, usaremos M para referirnos a la clase de modo asociada a una entidad (definida mediante tablas). Por ejemplo, para el caso del SIS, `mcPressure` es la clase de modo asociada a la variable controlada `cSafetyInjection`.

Básicamente, las tablas SCR pueden ser de dos tipos: las *tablas de condiciones* definen variables en términos del valor de otras variables, y las *tablas de eventos* se utilizan para definir variables que dependen de la ocurrencia de ciertos eventos.

TABLAS DE CONDICIÓN Las tablas de condición definen el valor de una entidad r usando los valores actuales de otras entidades. La Tabla 1 muestra la forma general de una tabla de condición T_c, donde m_i son modos de M -la clase de modo asociada con r-, $c_{i,j}$ son condiciones, y v_i son valores del tipo de r. T_c debe satisfacer las siguientes condiciones:

1. $m_i \neq m_j$, para cada $1 \leq i, j \leq n$ tales que $i \neq j$.
2. $v_i \neq v_j$, para cada $1 \leq i, j \leq p$ tales que $i \neq j$.
3. $\bigcup_{i=1}^{n} m_i = M$.
4. *Completitud*: La disyunción de todas las condiciones de cada fila de la tabla debe evaluar a verdadero. Esto quiere decir, que debemos de considerar

Modos	Condiciones			
m_1	$c_{1,1}$	$c_{1,2}$	$\dots$	$c_{1,p}$
m_2	$c_{2,1}$	$c_{2,2}$	$\dots$	$c_{2,p}$
$\dots$	$\dots$	$\dots$	$\dots$	$\dots$
m_n	$c_{n,1}$	$c_{n,2}$	$\dots$	$c_{n,p}$
r	v_1	v_2	$\dots$	v_p

Tabla 1: Tabla de Condición

Modos	Condiciones	
High, Permitted	True	False
TooLow	tOverridden	NOT tOverridden
cSafetyInjection	Off	On

Tabla 2: SIS: Tabla de Condición para `cSafetyInjection`.

todos los posibles casos (esto indica una función total). Formalmente, para todo $1 \leq i \leq n$, $(\vee_{k=1}^{p} c_{i,k}) = true$.

5. *Disyunción*: las condiciones en la misma fila no pueden superponerse. Formalmente, para todo $1 \leq i \leq n, 1 \leq j, k \leq p, j \neq k$, $c_{i,j} \wedge c_{i,k} = false$.

Asumamos que la entidad r depende de las variables $x_1, x_2, \dots, x_l$ (con $(r, x_i) \in D$), donde distinguimos a x_1 como la clase de modo M, y las restantes $x_2, \dots, x_l$ son entidades que aparecen en las condiciones que definen la tabla. Luego, T_c define una función total F_c de la siguiente manera:

$$r = F_c(x_1, x_2, \dots, x_l) = \begin{cases} v_1 & \text{if } \bigvee_{i=1}^{n}(x_1 = m_i \wedge c_{i,1}) \\ v_2 & \text{if } \bigvee_{i=1}^{n}(x_1 = m_i \wedge c_{i,2}) \\ \vdots & \\ v_p & \text{if } \bigvee_{i=1}^{n}(x_1 = m_i \wedge c_{i,p}) \end{cases}$$

Las condiciones anteriores 1 y 5 aseguran que F_c es una función, mientras que la 3 y 4 aseguran que F_c es total.

Continuando con el SIS, la variable controlada `cSafetyInjection` es definida mediante la tabla de condición mostrada en Tabla 2. Podemos observar, por ejemplo, que cuando el modo es `TooLow` y el sistema no esta bloqueado, entonces la inyección de líquido debe encenderse (`cSafetyInjection = On`). Luego, la función que define la Tabla 2 es la siguiente:

$$\texttt{cSafetyInjection} =$$
$$F_c(\texttt{mcPressure}, \texttt{tOverridden}) =$$
$$\begin{cases} \texttt{Off} & \text{if } \texttt{mcPressure} = \texttt{High} \vee \texttt{mcPressure} = \texttt{Permitted} \vee \\ & \quad (\texttt{mcPressure} = \texttt{TooLow} \wedge \texttt{tOverrridden}) \\ \texttt{On} & \text{if } \texttt{mcPressure} = \texttt{TooLow} \wedge \neg\texttt{tOverrridden} \end{cases}$$

TABLA DE EVENTOS Una *tabla de eventos* asigna valores a una entidad r acorde a la ocurrencia de ciertos eventos en el sistema, por lo que su definición depende del valor actual y siguiente de las variables de las que depende. La forma general de la tabla de eventos T_e es dada en la Tabla 3, donde m_i son

Modes	Events			
m_1	$e_{1,1}$	$e_{1,2}$	...	$e_{1,p}$
m_2	$e_{2,1}$	$e_{2,2}$	...	$e_{2,p}$
...	...	...	...	...
m_n	$e_{n,1}$	$e_{n,2}$	...	$e_{n,p}$
r	v_1	v_2	...	v_p

Tabla 3: Tabla de Eventos.

Modes	Events	
High	Never	@F(mcPressure=High)
TooLow, Permitted	@T(mBlockOn) when mReset=Off	@T(mcPressure=High) OR @T(mReset=On)
tOverridden	True	False

Tabla 4: SIS: Tabla de Eventos para `tOverridden`.

modos de M, $e_{i,j}$ son eventos, y v_i son valores del tipo de r. T_e debe cumplir con las siguientes condiciones:

1. $m_i \neq m_j$, para cada $1 \leq i, j \leq n$ tales que $i \neq j$.
2. $v_i \neq v_j$, para cada $1 \leq i, j \leq p$ tales que $i \neq j$.
3. *Disyunción*: los eventos en la misma fila deben ser mutuamente disjuntos. Formalmente, para todo $1 \leq i \leq n, 1 \leq j, k \leq p, j \neq k$, $e_{i,j} \wedge e_{i,k} = false$.

Supongamos qur r depende del valor actual de las entidades $x_1, x_2, \ldots, x_l$ y de los valores nuevos de $y'_1, y'_2, \ldots, y'_q$, donde nuevamente asumimos que x_1 es la clase de modo de M, y $x_2, \ldots, x_l, y'_1, y'_2, \ldots, y'_q$ son entidades que aparecen en los eventos de la tabla. Luego, T_e define la función F_e como:

$$r' = F_e(x_1, \ldots, x_l, y'_1, \ldots, y'_q) = \begin{cases} v_1 & \text{if} \bigvee_{i=1}^{n}(x_1 = m_i \wedge e_{i,1}) \\ v_2 & \text{if} \bigvee_{i=1}^{n}(x_1 = m_i \wedge e_{i,2}) \\ \vdots & \\ v_p & \text{if} \bigvee_{i=1}^{n}(x_1 = m_i \wedge e_{i,p}) \\ r & \text{otherwise} \end{cases}$$

F_e es una función debido a las condiciones 1 y 3 anteriores, y a la *One Input Assumption* que se asume sobre las especificaciones SCR (sólo un evento de entrada puede ser ejecutado a la vez, ver Subsección 3.5.1). Además, note que si no ocurre ninguno de los eventos que aparecen en la definición de r, entonces mantiene su valor actual, asegurando que F_e es una función total.

Considere el término `tOverridden` del SIS, definido mediante la tabla de eventos en Tabla 4. Esta tabla indica exactamente cuando las acciones del sistema están bloqueadas, por ejemplo, cuando el interruptor block fue presionado en

cualquier modo diferente a `High` y el sistema no estaba reseteado. La función F_e que define la Tabla 4 es dada a continuación:

$$\texttt{tOverridden'} =$$

$$F_e(\texttt{mBlock}, \texttt{mReset}, \texttt{mcPressure}, \texttt{tOverridden}, \texttt{mBlock'}, \texttt{mReset'}, \texttt{mcPressure'}) =$$

$$\begin{cases} \texttt{True} & \text{if } (\texttt{mcPressure} = \texttt{TooLow} \wedge \texttt{mBlock'} = \texttt{On} \wedge \texttt{mBlock} = \texttt{Off} \wedge \\ & \quad \texttt{mReset} = \texttt{Off}) \vee (\texttt{mcPressure} = \texttt{Permitted} \wedge \texttt{mBlock'} = \texttt{On} \wedge \\ & \quad \texttt{mBlock} = \texttt{Off} \wedge \texttt{mReset} = \texttt{Off}) \\ \texttt{False} & \text{if } (\texttt{mcPressure} = \texttt{TooLow} \wedge \texttt{mReset'} = \texttt{On} \wedge \texttt{mReset} = \texttt{Off}) \vee \\ & \quad (\texttt{mcPressure} = \texttt{Permitted} \wedge \texttt{Reset'} = \texttt{On} \wedge \texttt{mReset} = \texttt{Off}) \vee \\ & \quad (\texttt{mcPressure'} \neq \texttt{High} \wedge \texttt{mcPressure} = \texttt{High}) \vee \\ & \quad (\texttt{mcPressure'} = \texttt{High} \wedge \texttt{mcPressure} \neq \texttt{High}) \\ \texttt{tOverridden} & \text{otherwise} \end{cases}$$

TABLAS DE TRANSICIÓN DE MODOS Los cambios de modos dependen de eventos, por lo que son descritos mediante un tipo especial de tablas de eventos: las *tablas de transición de modos*. Luego, la forma general de una tabla de transición de modos T_m, dada en la Tabla 5, puede ser traducida a una tabla de eventos dada en la Tabla 3. Además, la función F_m asociada a la tabla T_m puede ser definida como mostramos anteriormente. Las condiciones que debe cumplir una tabla de transición de modos son las mismas que las que debe cumplir una tabla de eventos, pero además debe cumplir una propiedad extra: todos los modos de la clase de modos M deben poder ser alcanzados desde el estado inicial del sistema.

4. Alcanzabilidad en M: Sea ${F_m}^q$ denotando q aplicaciones sucesivas de la función F_m (por ejemplo, ${F_m}^2(x) = F(F(x))$), y sea $m_i \in M$ el modo inicial. Luego, para cada modo $m_j \in M$ existe $p \geq 0$ tal que ${F_m}^p(m_i) = m_j$.

Old mode	Event	New mode
m_1	$e_{1,1}$ $e_{1,2}$ ... e_{1,k_1}	$m_{1,1}$ $m_{1,2}$ ... m_{1,k_1}
...	...	...
m_n	$e_{n,1}$ $e_{n,2}$ ... e_{n,k_n}	$m_{1,1}$ $m_{1,2}$ ... m_{n,k_n}

Tabla 5: Tabla de Transición de Modos

Old mode	Event	New mode
TooLow	@T(mWaterPres >= Low)	Permitted
Permitted	@T(mWaterPres < Low)	TooLow
Permitted	@T(mWaterPres >= Permit)	High
High	@T(mWaterPres < Permit)	Permitted

Tabla 6: SIS: Tabla de Transición de Modos para `mcPressure`.

La Tabla 6 modela el cambio de modos del SIS. Por ejemplo, cuando el sistema esta en modo `TooLow` y la presión del agua se vuelve mayor o igual a la

constante Low, entonces el modo cambia a Permitted (ver la primer fila de la tabla de transición de modos). La función F_m definida mediante la Tabla 6 es la siguiente:

$$\texttt{mcPressure'} =$$
$$F_m(\texttt{mWaterPres}, \texttt{mcPressure}, \texttt{mWaterPres'}) =$$
$$\begin{cases} \texttt{TooLow} & \text{if } \texttt{mcPressure} = \texttt{Permitted} \wedge \texttt{mWaterPres'} < \texttt{Low} \wedge \\ & \quad \neg\texttt{mWaterPres} < \texttt{Low} \\ \texttt{Permitted} & \text{if } (\texttt{mcPressure} = \texttt{TooLow} \wedge \texttt{mWaterPres'} \geq \texttt{Low} \wedge \\ & \quad \neg\texttt{mWaterPres} \geq \texttt{Low}) \vee (\texttt{mcPressure} = \texttt{High} \wedge \\ & \quad \texttt{mWaterPres'} < \texttt{Permit} \wedge \neg\texttt{mWaterPres} < \texttt{Permit}) \\ \texttt{High} & \text{if } \texttt{mcPressure} = \texttt{Permitted} \wedge \texttt{mWaterPres'} \geq \texttt{Permit} \wedge \\ & \quad \neg\texttt{mWaterPres} \geq \texttt{Permit} \\ \texttt{mcPressure} & \text{otherwise} \end{cases}$$

Ejecuciones de Especificaciones SCR

La especificación SCR detecta cambios en las variables monitoreas, y reacciona propagando los cambios a las variables dependientes (términos, clases de modos, variables controladas) acorde lo indican las tablas. Más formalmente, dado un sistema SCR $\Sigma = (S, S_0, E^m, T)$, una *ejecución* es una secuencia infinita de estados y eventos de entrada $\sigma =< s_0 > e_0 < s_1 > \ldots < s_k > e_k < s_{k+1} > \ldots$ con las siguientes propiedades:

1. $s_0 \in S_0$ es un estado inicial del sistema,
2. $e_0, e_1, \ldots$ es la secuencia de *eventos de entrada* para los cuales el sistema reacciona, es decir, que cada evento $e_i = (\texttt{mV}_i, v, v')$ describe el cambio en una variable monitoreada $\texttt{mV}_i$,
3. $T(s_i, e_i) = s_{i+1}$, es decir, a partir del estado corriente s_i y el evento de entrada e_i, se obtiene el nuevo estado s_{i+1} utilizando la función de transformación del sistema T descrita por medio de tablas SCR. Básicamente, T propaga las modificaciones sobre todas las variables que dependen de $\texttt{mV}_i$. Aquellas variables que no dependen de $\texttt{mV}_i$, mantienen el mismo valor que el estado corriente.

Por ejemplo, considere el siguiente estado $s = \langle \texttt{High}, 4000, \texttt{On}, \texttt{Off}, \mathit{True}, \texttt{Off} \rangle$, correspondiente a los valores de las variables mcPressure, mWaterPres, mBlock, mReset, tOverridden and cSafetyInjection. respectivamente. Si el evento

de entrada es `mWaterPres' = mWaterPres-1`, luego el siguiente estado alcanzado es $\langle$`Permitted`, 3999, `On`, `Off`, *False*, `Off`$\rangle$, si asumimos que la constante `Permit` = 4000.

3.6 RESUMEN

Es este capítulo presentamos dos modelos conceptuales en los que se basan las metodologías de requisitos: el Mundo y la Máquina, y el Modelo de Cuatro Variables. Mostramos algunas semejanzas y diferencias entre ambos enfoques, en [Gunter et al., 2000] puede encontrar una comparación más detallada. Además fijamos la nomenclatura que vamos a utilizar a lo largo de la tesis, basándonos en la utilizada por estos enfoques (salvo mínimas diferencias).

Principalmente, introducimos las metodologías formales de requisitos sobre las que se basan nuestra técnicas: el método KAOS y el método SCR. Presentamos la sintaxis y semántica de sus especificaciones, además de algunos detalles respecto como proceder metodologícamente en cada formalismo. A lo largo del capítulo mencionamos numerosas citas a las cuales puede referirse para una comprensión más en detalle de cada método.

4

Elaboración Formal de Requisitos de Software

4.1 INTRODUCCIÓN

El proceso de ingeniería de requisitos tiene como principal objetivo producir un documento de requisitos de calidad, que oriente a los desarrolladores a construir un sistema que se adecue a las expectativas del cliente. En pos de mejorar la calidad de sus especificaciones, la mayoría de los lenguajes y metodologías formales de la ingeniería de requisitos, brindan poderosos mecanismos de análisis para evaluar, por ejemplo, la completitud y consistencia de los requisitos. Típicamente, las técnicas que automáticamente realizan este tipo de análisis, apuntan a la *etapa tardía* del proceso de ingeniería de requisitos, es decir, requieren una especificación de requisitos bastante avanzada y detallada, sin enfocarse en cómo fué elaborada dicha especificación. Sin embargo, trabajos como [Yu, 1997] han mostrado que la *etapa temprana* de requisitos, es tán importante como la tardía por varias razones. La etapa temprana del proceso de ingeniería de requisitos involucra, en general, tares específicas para la *captura* y *elaboración* de requisitos. Es aquí donde se trata de comprender las necesidades del cliente, pensar y modelar como podrían solventarse, e identificar los actores necesarios para la solución (software y agentes ambientales). Debido a la *incertidumbre* e *imparcialidad* de muchos requisitos en estas etapas, las especificaciones elaboradas deberán considerar (y seleccionar) entre varias posibilidades, identificar potenciales riesgos, y adaptarse de la mejor manera a cambios en los requisitos. Es generalmente reconocido que muchas de estas tareas, cruciales para el éxito del sistema resultante, se realizan de manera informal.

En las últimas décadas, las metodologías *orientadas a objetivos* (como KAOS [Dardenne et al., 1993] e I* [Yu, 1997]) han sido desarrolladas y exitosamente aplicadas al problema de captura, elaboración, especificación y validación de requisitos de software, mostrando ser particulamente apropiadas para muchas tareas de la etapa temprana de la ingeniería de requisitos. El modelado guiado por *objetivos* ha mostrado ser una forma adecuada para explorar diseños alternativos del sistema, e identificar situaciones excepcionales (obstáculos) que

podrían interferir en la satisfacción de los objetivos. Típicamente, estos métodos demandan el refinamiento de objetivos de alto nivel, los cuales requieren la cooperación de varios agentes para ser satisfechos, en objetivos que pueden ser realizados por agentes individuales [Dardenne et al., 1993]. Un objetivo asignado a un agente debe ser *operacionalizado*, es decir, mapeado en operaciones que serán provistas y ejecutadas por el agente en pos de garantizar el objetivo del cuál es responsable.

La *operacionalización de objetivos* consiste en sintetizar un modelo operacional para el agente de software, de manera tal que su comportamiento satisfaga los objetivos del sistema (más detalles en Sección 4.3). El modelo operacional generado, luego puede ser sometido a diversos tipos de análisis (como completitud, consistencia y minimalidad) para mejorar la calidad de la especificación de requisitos que será provista a los desarrolladores. Note que esta tarea, de mapear objetivos declarativos en requisitos operacionales, nos permite acercarnos a la etapa tardía del proceso de ingeniería de requisitos.

El problema de operacionalización de objetivos ha sido estudiado por varios investigadores. En el trabajo de Letier y van Lamsweerde [Letier and van Lamsweerde, 2002b], se propone una técnica basada en patrones para *derivar* un modelo operacional desde objetivos declarativos del sistema. Esta técnica provee algunas plantillas para derivar, desde objetivos expresados en Linear-Time Temporal Logic (LTL), un conjunto de pre/triggering condiciones requeridas para operaciones, de manera tal que los objetivos sean satisfechos. Sin embargo, este enfoque está limitado a tipos particulares de objetivos LTL (impone restricciones sintácticas) y requiere un Modelo de Objetivos completamente refinado. Más recientemente, Alrajeh et al. propusieron un enfoque semi automático para *aprender* requerimientos operacionales desde un conjunto de objetivos [Alrajeh et al., 2009]. Esta técnica usa model checking para verificar si el modelo operacional dado satisface los objetivos. Si la verificación falla, el usuario debe examinar el contraejemplo generado por el model checker, para identificar la ejecución incorrecta de una operación, y proveer escenarios positivos mostrando "buenas" ejecuciones de esa operación. Estos escenarios son utilizados por un motor de aprendizaje inductivo para computar automáticamente nuevas pre/triggering condiciones requeridas para la operación seleccionada. La condición obtenida asegura que el contraejemplo es removido y el comportamiento descripto por los escenarios positivos es preservado. El enfoque de Alrajeh et al. es semi automático debido a que requiere la intervención del ingeniero para proveer los escenarios positivos.

En este capítulo presentamos una técnica para la operacionalización de objetivos, que *automáticamente* computa pre/triggering condiciones requeridas para operaciones, a fin de satisfacer un conjunto de objetivos [Degiovanni et al.,

2014]. Esta técnica no depende de los escenarios provistos por el usuario ni de sus características, como por ejemplo, riqueza y corrección, como es el caso de [Alrajeh et al., 2009]. El proceso de refinamiento esta basado en el uso de *interpolación* y SAT Solving. Similar a los enfoques previos, nuestra técnica aplica a objetivos de *safety* y *time progress* (progreso del tiempo). Más aún, podremos lidiar con un amplio rango de objetivos de *liveness*, en particular, aquellos capturados por el *patrón de reactividad* [Manna and Pnueli, 1992].

Nuestro enfoque comienza con una fase de model checking, para verificar si la especificación operacional de requisitos satisface los objetivos o no. Si la verificación es exitosa, luego el modelo operacional no necesita más refinamientos. Si el model checker produce un contraejemplo, entonces computamos un interpolante para este contraejemplo y el objetivo violado, el cuál será explotado para fortalecer o debilitar precondiciones o condiciones de triggering requeridas, respectivamente, para remover el contraejemplo. La técnica realiza varios chequeos lógicos para garantizar que las condiciones requeridas refinadas sean consistentes con la actual especificación operacional. Estos chequeos son realizados usando SAT Solving.

Tal como explicamos en Subsección 2.2.2, un interpolante para dos fórmulas A y B cuya conjunción es inconsistente, es una fórmula implicada por A, inconsistente con B, y expresada en el lenguaje común de A y B. Interpolación ha sido ampliamente utilizado en verificación basada en abstracción [Mcmillan, 2005], notablemente para el refinamiento automático de modelos abstractos usando el modelo concreto y contraejemplos abstractos espurios. En este trabajo, proponemos utilizar interpolación con un propósito, relacionado, pero diferente. Esencialmente, un interpolante para una fórmula capturando una traza y el objetivo violado, es una propiedad sobre la traza expresada en el lenguaje común de la traza y el objetivo. Este interpolante puede ser pensado como una *explicación* de porqué la traza viola el objetivo. Tal como veremos, los interpolantes pueden ser utilizados para refinar condiciones requeridas, en el proceso de operacionalización de objetivos.

El resto de este capítulo se organiza de la siguiente manera: la Sección 4.2 introduce el ejemplo motivador usado para presentar la técnica; la Sección 4.3 describe formalmente el problema de operacionalización de objetivos; la Sección 4.4 presenta nuestro enfoque en detalle, para operacionalizar objetivos de safety; la Sección 4.5 extiende la técnica para poder lidiar con objetivos de liveness; y en la Sección 4.6 proponemos la metodología a seguir para lograr exitosamente operacionalizar los objetivos. En la Sección 4.7 evaluamos nuestra técnica, y la comparamos con otras relacionadas, sobre varios casos de estudio. Finalmente, discutimos los trabajos relacionados en Sección 4.8 y concluimos en la Sección 4.9.

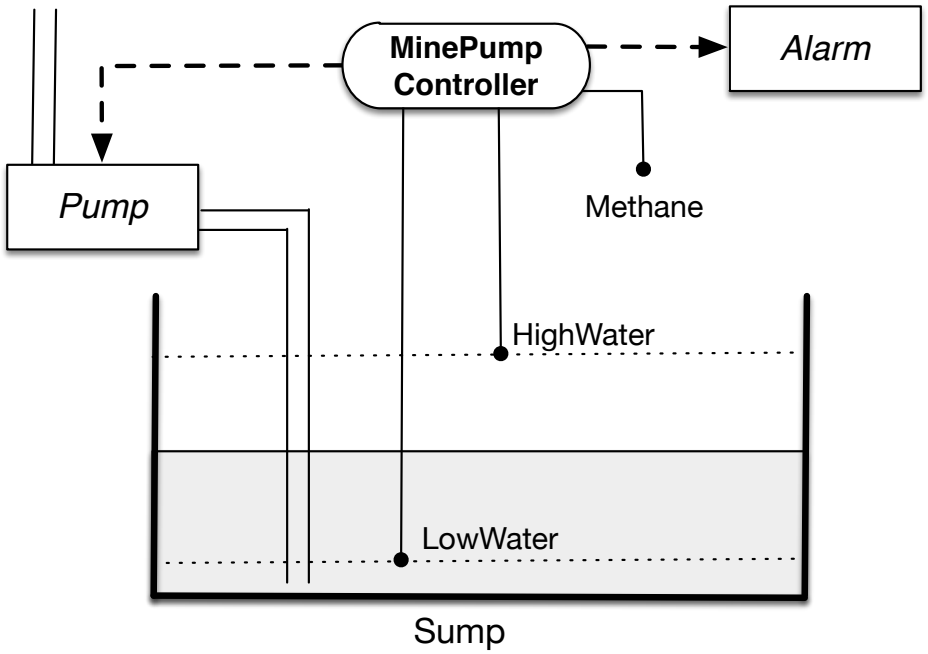

Figura 21: Ejemplo Motivador: `MinePumpController`.

4.2 EJEMPLO MOTIVADOR

Durante la Sección 3.4 utilizamos un simple modelo operacional del `MinePumpController` para introducir el método KAOS. En esta sección presentaremos una versión más sofisticada de este ejemplo, tomada de [Kramer et al., 1983], en la cuál el sistema debe considerar más variables ambientales y controladas para satisfacer los objetivos. Este modelo será utilizado como ejemplo motivador para presentar nuestra técnica de operacionalización de objetivos.

La Figura 21 muestra que el agua que se filtra en la mina es almacenado en un tanque (`Sump`). El agente de software `MinePumpController` puede detectar si el nivel del agua está demasiado bajo o alto, utilizando los sensores `LowWater` y `HighWater`, respectivamente. `MinePumpController` controla el estado de la bomba mediante un interruptor (`PumpOn`), debiendo encenderlo cuando el nivel de agua es alto, y apagarlo cuando el nivel de agua es bajo. Además, el sistema cuenta con un sensor para detectar niveles críticos de metano en el ambiente (`Methane`). En caso de que el sensor se active, el controlador debe disparar una alarma (`Alarm`) para informar a los operarios que deben evacuar la mina. Para evitar una explosión dentro de la mina, la bomba debe operar sólo cuando los niveles de metano no sean críticos.

```
Operation switchPumpOn          Operation switchPumpOff
  DomPre  ¬PumpOn                 DomPre  PumpOn
  DomPost PumpOn                  DomPost ¬PumpOn
  ReqPre  True                    ReqPre  True
  ReqTrig False                   ReqTrig False

Operation raiseAlarm            Operation stopAlarm
  DomPre  ¬Alarm                  DomPre  Alarm
  DomPost Alarm                   DomPost ¬Alarm
  ReqPre  True                    ReqPre  True
  ReqTrig False                   ReqTrig False
```

Figura 22: Modelo Operacional inicial para el `MinePumpController`.

Las variables monitoreadas por el `MinePumpController`, son modificadas por eventos ambientales: `belowLow`, `aboveLow`, `aboveHigh` y `belowHigh` modifican el nivel del agua en la mina, mientras que `signalMethane` y `signalNoMethane` ocurren acorde a la presencia de niveles críticos de metano en el ambiente. El sistema puede activamente controlar el estado de la bomba y la alarma con los eventos `switchPumpOn`, `switchPumpOff`, `raiseAlarm` y `stopAlarm`. El Modelo Operacional inicial para el `MinePumpController` es presentado en Figura 22. Como puede observar, inicialmente las operaciones estarán habilitadas a ejecutarse mientras lo permita la precondición de dominio (es decir, la precondición requerida es True), y no estarán obligadas a ejecutarse en ningún estado (es decir, la condición de triggering requerida es False).

Supongamos que `MinePumpController` debe cumplir con el objetivo `PumpOff-WhenMethane` que establece que *"cuando hay presencia de metano en el ambiente, la bomba debe ser apagada"*. Para verificar si el modelo operacional satisface este objetivo, utilizaremos una técnica tomada de [Letier et al., 2008], que presentaremos en detalle en la Subsección 4.3.1. Básicamente, la especificación operacional es mapeada a un sistema de transición basado en eventos (LTS), los objetivos son expresados en la lógica temporal FLTL (ver Subsección 2.4.2), y el model checker LTSA [Magee and Kramer, 2006] verifica la validez del objetivo sobre el LTS. El anterior objetivo puede expresarse en FLTL como sigue:

```
assert PumpOffWhenMethane =
     □(tick → (Methane → ◯(¬tick 𝒲 (tick ∧ ¬PumpOn))))
```

Tal como veremos en Subsección 4.3.1, el evento `tick` es un medio para capturar la interpretación *síncrona* del Modelo Operacional KAOS bajo una

interpretación *asíncrona*, que es la usada por LTSA [Letier et al., 2008]. La fórmula anterior indica que, si `Methane` es verdadero cuando `tick` ocurre, entonces antes del siguiente `tick` la bomba debe ser apagada (¬`PumpOn`). Teniendo el Modelo Operacional capturado por un LTS y los objetivos especificados en fórmulas FLTL, es posible verificar si la especificación satisface los objetivos o no. En este caso, los objetivos pueden ser violados, y el model checker LTSA produce el siguiente contraejemplo:

```
Trace to property violation in PumpOffWhenMethane:
  tick  (s0)
  signalMethane    Methane
  tick             Methane  (s1)
  switchPumpOn     Methane ∧ PumpOn
  tick             Methane ∧ PumpOn  (s2)
```

Al examinar este contraejemplo, podemos observar que en el estado `(s1)` el nivel de metano es crítico, pero en el estado `(s2)` la bomba está encendida, por lo que no satisface el objetivo `PumpOffWhenMethane`. El problema en esta ejecución se debe a que la operación `switchPumpOn` está habilitada a ejecutarse cuando hay metano en el ambiente (note que su precondición requerida lo permite).

A continuación explicamos brevemente cómo podemos utilizar interpolación para refinar el modelo operacional, y remover así esta violación retornada por el model checker.

4.2.1 *Usando Interpolación para Refinar Modelos Operacionales*

El anterior contraejemplo puede ser expresado como una fórmula proposicional A, donde los estados del sistema son representados por variables booleanas (una por cada fluente en la especificación), y estas variables son replicadas 3 veces, para los 3 estados diferentes de la traza (es decir, `s0`, `s1` y `s2`), relacionando cada una con la siguiente acorde indica cada evento que se ejecuta (ver por ejemplo, Figura 3). Supongamos que el objetivo puede ser expresado como una *propiedad de estado* B (por ejemplo, es directo si la propiedad es de safety), refiriendose a los últimos estados de la traza. Como la traza es una violación a B, claramente $A \wedge B$ es insatisfactible. Podemos entonces usar interpolación, para ver que interpolante obtenemos. En este caso, el interpolante es `Methane` ∧ `PumpOn`. Este interpolante indica que el problema es que en el último estado hay metano en el ambiente y la bomba esta encendida, lo cuál causa una violación al objetivo `PumpOffWhenMethane`. Comencemos entonces desde atrás hacia adelante, desde el estado anterior al último `tick`, tratanto

de encontrar el primer lugar donde la violación pueda ser evitada. Podemos computar la *weakest precondition (WP)* entre el interpolante con respecto a la última acción, es decir, `switchPumpOn`, obteniendo:
WP(`switchPumpOn`, `Methane` $\wedge$ `PumpOn`) = `Methane`.

Note que esta última operación tiene la habilidad de "cambiar" el valor de verdad del interpolante (el interpolante se hace válido después de ejecutar `switchPumpOn`). Así, previniendo su ejecución, podremos remover este contraejemplo. Sin embargo, primero tenemos que verificar si la acción `switchPumpOn` está o no obligada a ejecutarse cuando `Methane` esta activado. Esto puede chequearse usando algún mecanismo de decisión (por ejemplo, SAT Solving), para decidir la fórmula `Methane` $\Rightarrow$ *ReqTrig*(`switchPumpOn`). Si este es el caso, entonces la acción no puede ser prevenida, y tenemos que seguir buscado otra forma de remover el contraejemplo. Si no es el caso, podemos quitar el contraejemplo agregando $\neg$`Methane` a la precondición requerida de `switchPumpOn`.

4.3 OPERACIONALIZACIÓN DE OBJETIVOS

En términos generales, el proceso de *operacionalización de objetivos* consiste en *prescribir* condiciones requeridas para operaciones de un agente de software, de manera tal que el Modelo Operacional obtenido satisfaga los objetivos [Letier and van Lamsweerde, 2002b]. Por ejemplo, como vimos anteriormente, para lograr operacionalizar el objetivo `PumpOffWhenMethane` será necesario agregar la siguiente precondición requerida a la operación `switchPumpOn`:

```
Operation switchPumpOn
  ...
  ReqPre for PumpOffWhenMethane
      ¬ Methane
```

Note que la precondición requerida agregada nos garantiza que la operación `switchPumpOn` no puede ejecutarse cuando haya nivel crítico de metano, lo que produciría una violación al objetivo `PumpOffWhenMethane`, similar a la mostrada anteriormente.

Usualmente, un objetivo será operacionalizado por múltiples operaciones, considerando varias condiciones requeridas. Por ejemplo, para operacionalizar el objetivo `PumpOffWhenMethane`, es necesario además una condición de triggering requerida para la operación `switchPumpOff`:

```
Operation switchPumpOff
  ...
  ReqTrig for PumpOffWhenMethane
      Methane
```

Esta condición de triggering indica que cuando el controlador detecte niveles críticos de metano en el ambiente, entonces deberá apagar la bomba ejecutando la operación `switchPumpOff`.

Cuando el agente de software es responsable de operacionalizar varios objetivos, puede ocurrir que una misma operación necesite múltiples pre/triggering/-post condiciones requeridas para operacionalizar cada objetivo. Por ejemplo, en la Subsección 3.4.4 para operacionalizar el objetivo `PumpSwitchOnWhenHighWater`, habíamos introducido `HighWater` como condición de triggering requerida para la operación `switchPumpOn`, y ¬`HighWater` como precondición requerida para `switchPumpOff` (ver Figura 19).

La operacionalización de objetivos es considerado un proceso incremental. Si nuevos objetivos surgen en el Modelo de Objetivos, entonces nuevas condiciones requeridas deberán ser agregadas al Modelo Operacional para satisfacerlo. Para evitar inconsistencias en el Modelo Operacional, cada vez que una nueva condición requerida es agregada, debemos garantizar que la siguiente meta-regla de KAOS no es violada:

$$ReqTrig(op) \wedge DomPre(op) \Rightarrow ReqPre(op) \qquad \text{(KAOS meta-rule)}$$

La satisfacción de esta regla garantiza que la condición de triggering de una operación, no se contradice con su precondición requerida. Es decir, cada vez que la operación este obligada a ejecutarse, entonces además debería estar habilitada a hacerlo. En particular, nuestra técnica automática de operacionalización de objetivos, utilizará SAT Solving para chequear la validez de esta regla en cada paso de refinamiento.

Diremos que un objetivo es *correctamente operacionalizado* por un conjunto de operaciones, si la satisfacción de todas sus condiciones requeridas garantiza la satisfacción del objetivo. Podemos definir formalmente esta noción utilizando la semántica para Modelos Operacionales, presentada en la Subsección 3.4.4.

Definición 4.1 (Correcta Operacionalización de Objetivos)**.** *Sea G un objetivo, y $\{R_1, \ldots, R_n\}$ el conjunto de las condiciones pre/triggering/post requeridas de las operaciones del Modelo Operacional. Diremos que el objetivo G es correctamente operacionalizado por el Modelo Operacional, si y sólo si, las siguientes condiciones valen:*

- (completitud) $[\![R_1]\!] \wedge \ldots \wedge [\![R_n]\!] \models G$

- (consistencia) $[\![R_1]\!] \wedge \ldots \wedge [\![R_n]\!] \not\models \boldsymbol{false}$

Note que la semántica de operacionalización de objetivos recae sobre las condiciones que podemos prescribir, es decir las condiciones requeridas. Las condiciones de dominio no aparecen explícitamente, pero están incluidas en la semántica de las condiciones requeridas del modelo operacional.

La *minimalidad* del Modelo Operacional suele ser también una propiedad deseable. Es decir, el Modelo Operacional no debería restringir el comportamiento del sistema más de lo necesario para satisfacer los objetivos. Esta propiedad puede expresarse formalmente como:

- (minimalidad) $G \models [\![R_1]\!] \wedge \ldots \wedge [\![R_n]\!]$

Garantizar esta propiedad puede ser muy complicado, sobre todo cuando se trata de operacionalizar múltiples objetivos. Más adelante discutiremos si nuestra técnica puede garantizar esta propiedad sobre los modelo operacionales que sintetiza.

4.3.1 *Modelos de Comportamiento a partir de Operaciones*

Veamos a continuación como podemos utilizar un model checker, en particular LTSA (Subsección 2.5.1), para verificar automáticamente si un Modelo Operacional satisface un Modelo de Objetivos.

Siguiendo [Letier et al., 2008], es posible obtener sistemáticamente un Modelo de Comportamiento (un LTS) a partir de un Modelo Operacional. Las condiciones requeridas pueden ser capturadas utilizando la lógica temporal FLTL (Subsección 2.4.2), combinando los operadores $\Box$ y $\bigcirc$, e introduciendo algunas restricciones FLTL extras para lidiar con la semántica sincrona del método KAOS. Luego, se puede construir automáticamente un LTS que caracteriza el comportamiento del sistema. Si los objetivos son expresados como fórmulas FLTL, podemos entonces utilizar el model checker LTSA para verificar si el LTS, que caracteriza al Modelo Operacional, satisface los objetivos.

Sin embargo, como KAOS posee una semántica *síncrona*, no podemos utilizar directamente la caracterización en aserciones LTL del Modelo Operacional presentada en Subsección 3.4.4, ya que LTSA posee semántica *asíncrona*. En una semántica *síncrona* los estados son observados por unidad de tiempo fijo (es decir, que entre dos estados pueden ejecutarse varias operaciones), mientras que en una semántica *asíncrona* los estados son observados después de la ejecución de cada evento. Esto hace que los operadores temporales no tengan el mismo

significado en lógicas con semántica síncrona y asíncrona. Por ejemplo, $\bigcirc P$ significa "P vale en la siguiente unidad de tiempo" si la semántica es síncrona, y "P vale después del siguiente evento" si la semántica es asíncrona. Para lidiar con este problema, vamos a utilizar la técnica presentada en [Letier et al., 2008], que pasamos a explicar a continuación.

Consideremos `tick` como un evento especial que modela el paso del tiempo del reloj del sistema [Magee and Kramer, 2006; Letier et al., 2005]. Luego, podemos utilizar este evento para clasificar los estados de un LTS como *observables* o *no-observables*. Básicamente, un estado de un LTS es *observable*, si existe una transición etiquetada con el evento `tick` que llega a él, mientras que los estados que no cumplen con esta propiedad son considerados *no-observables*.

Definición 4.2 (Estados Observables de un LTS)**.** *Sea* $\mathcal{P} = \langle Q, A \cup \{\texttt{tick}\}, \delta, q_0 \rangle$ *un LTS y* $s \in Q$ *un estado de* $\mathcal{P}$*. Diremos que* s *es un estado observable, si y sólo si, cumple con la siguiente condición:*

$$\exists s' \in Q \cdot \delta(s', \texttt{tick}) = s$$

Basados en este concepto de observabilidad, [Letier et al., 2005] propone una traducción de aserciones escritas en FLTL con semántica síncrona, a aserciones FLTL pero con semántica asíncrona, modelando las unidades de tiempo mediante el evento `tick`.

Definición 4.3 (Traducción de $FLTL_{Sync}$ a $FLTL_{Async}$)**.** *A continuación definimos* $Tr : FLTL_{Sync} \rightarrow FLTL_{Async}$ *una traducción de aserciones FLTL con semántica síncrona a aserciones FLTL pero con semántica asíncrona.*

$$\begin{array}{lcl}
Tr(\varphi \; \mathcal{U} \; \phi) & = & (\texttt{tick} \rightarrow Tr(\varphi)) \; \mathcal{U} \; (\texttt{tick} \wedge Tr(\phi)) \\
Tr(\Box\varphi) & = & \Box(\texttt{tick} \rightarrow Tr(\varphi)) \\
Tr(\Diamond\varphi) & = & \Diamond(\texttt{tick} \wedge Tr(\varphi)) \\
Tr(\bigcirc\varphi) & = & \bigcirc(\neg\texttt{tick} \; \mathcal{W} \; (\texttt{tick} \wedge Tr(\varphi))) \\
Tr(\varphi \rightarrow \phi) & = & Tr(\varphi) \rightarrow Tr(\phi) \\
Tr(\varphi \wedge \phi) & = & Tr(\varphi) \wedge Tr(\phi) \\
Tr(\varphi \vee \phi) & = & Tr(\varphi) \vee Tr(\phi) \\
Tr(\neg\varphi) & = & \neg Tr(\varphi)
\end{array}$$

La traducción del operador $\bigcirc\varphi$ de la Definición 4.3 muestra claramente la diferencia entre las semánticas síncrona/asíncrona, expresando que φ debe valer en la siguiente unidad de tiempo (el siguiente `tick`). Esta traducción asume que la aserción se evalúa inicialmente en la primer unidad de tiempo, es decir, el primer evento que se ejecuta es `tick` para observar el estado inicial.

A modo de ejemplo, veamos como obtenemos el objetivo `PumpOffWhenMethane` FLTL asíncrono a partir de su definición síncrona:

$$
\begin{aligned}
&Tr(\Box(\texttt{Methane} \rightarrow \bigcirc\neg\texttt{PumpOn}))\\
&= \Box(\texttt{tick} \rightarrow Tr(\texttt{Methane} \rightarrow \bigcirc\neg\texttt{PumpOn}))\\
&= \Box(\texttt{tick} \rightarrow (Tr(\texttt{Methane}) \rightarrow Tr(\bigcirc\neg\texttt{PumpOn})))\\
&= \Box(\texttt{tick} \rightarrow (\texttt{Methane} \rightarrow \bigcirc(\neg\texttt{tick}\ \mathcal{W}\ (\texttt{tick} \wedge Tr(\neg\texttt{PumpOn})))))\\
&= \Box(\texttt{tick} \rightarrow (\texttt{Methane} \rightarrow \bigcirc(\neg\texttt{tick}\ \mathcal{W}\ (\texttt{tick} \wedge \neg Tr(\texttt{PumpOn})))))\\
&= \Box(\texttt{tick} \rightarrow (\texttt{Methane} \rightarrow \bigcirc(\neg\texttt{tick}\ \mathcal{W}\ (\texttt{tick} \wedge \neg\texttt{PumpOn}))))
\end{aligned}
$$

Esta traducción es útil para codificar los objetivos en aserciones asíncronas FLTL. En [Letier et al., 2008] se presenta además la codificación del Modelo Operacional en términos de fluentes y aserciones FLTL, lo que nos permitirá sintetizar un LTS. Básicamente, cada operación del sistema es representada por un evento y las variables del sistema mediante fluentes.

Definición 4.4 (Caracterización de Operaciones)**.** *Sean* $\{op_1, \ldots, op_n\}$ *el conjunto de operaciones del Modelo Operacional. Haremos corresponder cada operación* op_i *a un evento, de igual nombre, para dar la caracterización FLTL del Modelo Operacional.*

Definición 4.5 (Caracterización de Variables)**.** *Cada variable* f *del Modelo Operacional (por ejemplo,* `PumpOn`*) será caracterizada mediante un fluente* $Fl_f \equiv \langle I_{Fl_f}, T_{Fl_f}, B_{Fl_f}\rangle$*, que definimos como sigue:*

- $op_i \in I_{Fl_f}$ *si y sólo si* f *aparece en la* $DomPost(op_i)$*;*
- $op_i \in T_{Fl_f}$ *si y sólo si* $\neg f$ *aparece en la* $DomPost(op_i)$*;*
- B_{Fl_f} *depende del valor inicial de la variable* f *en el Modelo Operacional.*

Note que la definición de estos fluentes captura la semántica de las postcondiciones de dominio (*DomPost*), ya que la ejecución de cada operación modificará el valor de las variables que controla. Veamos por ejemplo la definición de los fluentes que se corresponden a las variables del Modelo Operacional de la Figura 22:

$$
\begin{aligned}
\texttt{PumpOn} &\equiv \langle\{\texttt{switchPumpOn}\}, \{\texttt{switchPumpOff}\}, \mathit{False}\rangle\\
\texttt{Alarm} &\equiv \langle\{\texttt{raiseAlarm}\}, \{\texttt{stopAlarm}\}, \mathit{False}\rangle\\
\texttt{Methane} &\equiv \langle\{\texttt{signalMethane}\}, \{\texttt{signalNoMethane}\}, \mathit{False}\rangle\\
\texttt{HighWater} &\equiv \langle\{\texttt{aboveHigh}\}, \{\texttt{belowHigh}\}, \mathit{False}\rangle\\
\texttt{LowWater} &\equiv \langle\{\texttt{belowLow}\}, \{\texttt{aboveLow}\}, \mathit{True}\rangle
\end{aligned}
$$

La herramienta LTSA provee un comando `constraint` que permite generar un LTS a partir de una aserción FLTL de safety. Si la aserción no es de safety, LTSA retorna un error. El LTS derivado de la fórmula FLTL, puede ser compuesto con cualquier otro proceso.

Anteriormente, en la Subsección 3.4.4, presentamos una semántica formal del Modelo Operacional en términos de aserciones LTL síncronas. Por lo tanto, para construir el LTS que caracterice al Modelo Operacional, primero vamos a traducir las condiciones requeridas de cada operación a aserciones FLTL asíncronas. Luego, con el comando `constraint`, vamos a generar el LTS que caracteriza cada condición requerida, para finalmente componerlos y obtener así el LTS que representa el Modelo Operacional.

Definición 4.6 (Caracterización de Condiciones Requeridas)**.** *Para cada operación op_i del Modelo Operacional, definimos a continuación las aserciones asíncronas FLTL que capturan la semántica de la precondición de dominio y las condiciones requeridas de op_i.*

- *si la precondición no vale en el estado corriente, entonces la operación no puede ejecutarse en esta unidad de tiempo:*

$$\textbf{constraint}\ DomPre\_Op_i = \Box(\mathit{tick} \rightarrow (\neg \mathit{DomPre}(op_i) \rightarrow \bigcirc(\neg op_i\ \mathcal{W}\ \mathit{tick})))$$
$$\textbf{constraint}\ ReqPre\_Op_i = \Box(\mathit{tick} \rightarrow (\neg \mathit{ReqPre}(op_i) \rightarrow \bigcirc(\neg op_i\ \mathcal{W}\ \mathit{tick})))$$

- *si la condición de triggering vale, y además también es válida la precondición de dominio, entonces la operación debe ser ejecutada antes del siguiente* `tick`*:*

$$\textbf{constraint}\ ReqTrig\_Op_i = \Box((\mathit{tick} \wedge \mathit{DomPre}(op_i) \wedge \mathit{ReqTrig}(op_i)) \rightarrow \bigcirc(\neg \mathit{tick}\ \mathcal{W}\ op_i))$$

- *si la operación op_i se ejecuta, entonces en el siguiente estado observable debe valer $ReqPost(op_i)$:*

$$\textbf{constraint}\ ReqPost\_Op_i = \Box(op_i \rightarrow \bigcirc(\neg \mathit{tick}\ \mathcal{W}\ (\mathit{tick} \wedge \mathit{ReqPost}(op_i))))$$

Considerando el Modelo Operacional del `MinePumpController` de la Figura 22, y las condiciones requeridas para las operaciones `switchPumpOn` y `switchPumpOff` mencionadas anteriormente, la Figura 23 presenta una versión simplificada de la especificación FSP que podrá ser analizada con LTSA.

En [Letier et al., 2008] se muestra que si P y Q son propiedades de safety, ocurre que $\texttt{constraint}(P) \parallel \texttt{constraint}(Q) = \texttt{constraint}(P \wedge Q)$. Dada la caracterización presentada del Modelo Operacional, LTSA puede generar un LTS

```
// Variables del Modelo Operacional
fluent PumpOn = <switchPumpOn, switchPumpOff>
fluent HighWater = <aboveHigh, belowHigh>
fluent LowWater = < belowLow, aboveLow> initially 1
fluent Methane = <signalMethane, signalNoMethane>
fluent Alarm =< {raiseAlarm}, {stopAlarm} >

// Condiciones de DOMINIO
constraint DOMPre_switchPumpOn =
    [] ((tick && !PumpOn)-> X (! switchPumpOff W tick))
constraint DOMPre_switchPumpOff =
    [] ((tick && PumpOn) ->X (! switchPumpOn W tick))
constraint DOMPre_raiseAlarm =
    [](tick && ! ! Alarm) ->X(! raiseAlarm W tick))
constraint DOMPre_stopAlarm =
    [](tick && ! Alarm) ->X(! stopAlarm W tick))
||DOM = ( DOMPre_switchPumpOn || DOMPre_switchPumpOff
        || DOMPre_raiseAlarm ||DOMPre_stopAlarm
    ).

// Condiciones REQUERIDAS
constraint REQPre_switchPumpOn =
    [](tick && ! (!Methane)) ->X(! switchPumpOn W tick))
constraint REQTrig_switchPumpOff =
    [](tick && Methane && PumpOn ->X(! tick W switchPumpOff))
||REQ = (REQPre_switchPumpOn || REQTrig_switchPumpOff).

StartWithTick = (tick ->S0), S0 = ({AllActions} ->S0).
||MinePump = (StartWithTick || DOM || REQ).

// Objetivos a satisfacer por el MinePumpController
assert PumpOffWhenMethane =
  [](tick -> (Methane -> X (!tick W (tick && ! PumpOn))))
```

Figura 23: Especificación FSP para el `MinePumpController`.

para cada aserción FLTL, y luego componerlos en paralelo, para así finalmente obtener un LTS que representa el Modelo de Comportamiento del sistema (ver los detalles en la Figura 23). El LTS que se obtiene preserva la semántica asíncrona de la especificación y puede ser analizado utilizando el model checker LTSA para verificar, por ejemplo, si satisface los objetivos. Para más detalles, referirse a [Letier et al., 2008].

En la siguiente sección, presentaremos una técnica automática que detecta si el Modelo Operacional no satisface sus objetivos (utilizando esta caracterización), y mediante el uso de Interpolación y SAT solving resuelve la incompletitud del Modelo Operacional, computando condiciones requeridas adecuadas que correctamente operacionalizan los objetivos.

4.4 ELABORACIÓN DE REQUISITOS OPERACIONALES

En esta sección describiremos en detalle nuestra técnica para elaborar modelos operacionales de requisitos. Para simplificar la explicación, asumamos por ahora que los objetivos son propiedades de *safety*. Más adelante extenderemos el enfoque para lidiar con la propiedad de *Time Progress* (TP) y propiedades de liveness expresadas con el patrón de *reactividad* [Manna and Pnueli, 1992].

Dado un conjunto de objetivos $G = \{G_1, \ldots, G_n\}$, y una especificación del modelo operacional *Spec* que puede no satisfacer G, proponemos un enfoque iterativo que refina *Spec* mediante la construcción de nuevas pre/triggering condiciones requeridas para las operaciones, de manera tal que la especificación resultante refinada satisface G. Nuestro enfoque sólo puede refinar *Spec* mediante el debilitamiento de precondiciones o fortalecimiento de las condiciones de triggering requeridas de las operaciones de *Spec*.

La técnica consiste de dos fases que pueden observarse en la Figura 24. La fase *Model Checking* se encarga de verificar si *Spec* satisface o no los objetivos en G. Si la verificación tiene éxito, entonces la especificación no necesita refinamiento alguno. Por otro lado, si los objetivos no son satisfechos en *Spec*, el model checker detectará la violación para al menos un objetivo G_i, y producirá un contraejemplo como testigo de esta violación. La fase de *Refinamiento*, completamente automática, refina el modelo operacional de tal manera que la misma violación no podrá ocurrir nuevamente. Para lograr esto, nuestra técnica usa *interpolación* y comienza computando un interpolante (I) para el contraejemplo y el objetivo violado G_i. Este interpolante es explotado, usando *weakest precondition* (WP), para identificar operaciones cuyas condiciones puedan ser alteradas para remover el contraejemplo. Más precisamente, el interpolante obtenido, con la ayuda de *weakest precondition*, produce una fórmula (R) que

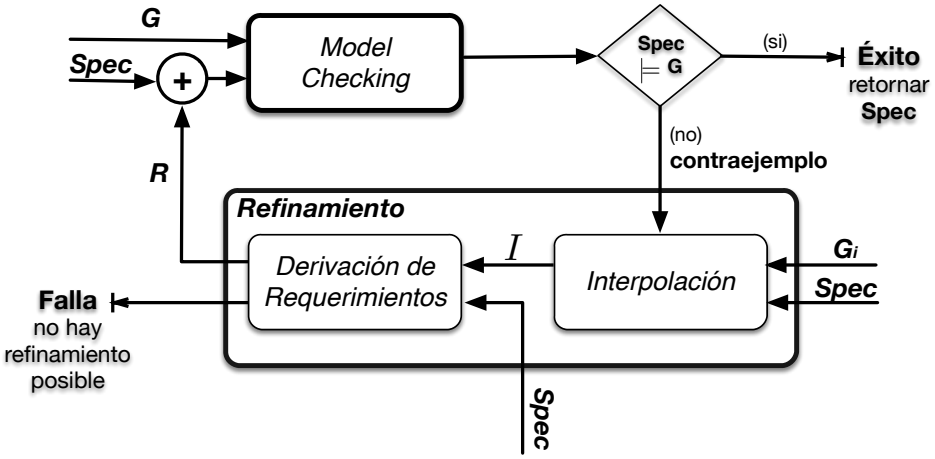

Figura 24: Resumen del enfoque iterativo.

será utilizada para fortalecer o debilitar pre/triggering condiciones requeridas, respectivamente, a fin de remover contraejemplos.

La especificación refinada, que incorpora nuevas y *más precisas* condiciones requeridas para algunas operaciones, remueve de su comportamiento el contraejemplo encontrado previamente. El proceso se repite hasta que no haya más violaciones a los objetivos, retornando una operacionalización correcta para los objetivos G (ver Sección 4.3), o hasta que alcanza un punto en el cuál no puede remover la violación encontrada, es decir, la técnica no puede refinar condiciones requeridas que remuevan el contraejemplo de la especificación.

4.4.1 *Fase de Model Checking*

El model checker LTSA ha sido utilizado exitosamente para verificar si un Modelo Operacional de requisitos satisface un Modelo de Objetivos expresados en FLTL [Letier et al., 2008]. Tal como mostramos en la Subsección 4.3.1, la descripción del estado del sistema es dada en términos de fluentes, y para capturar la semántica del Modelo Operacional KAOS en LTSA, usaremos el evento `tick` que representa de manera explícita el comienzo y fin de las unidades de tiempo en FLTL asíncrona. A modo de ejemplo, considere el siguiente objetivo `PumpOnWhenHighWaterANDNoMethane` para el modelo de la Figura 23 en donde los ticks ya fueron incorporados:

```
assert PumpOnWhenHighWaterANDNoMethane =
     □(tick→((HighWater∧¬Methane)→ ◯(¬tick 𝒲 (tick ∧ PumpOn))))
```

Esta fórmula expresa que, si en alguna unidad de tiempo el nivel del agua es alto y no hay nivel crítico de metano, entonces la bomba debe ser encendida en la siguiente unidad de tiempo. Note que este objetivo es similar, pero más fuerte, que aquel utilizado en Sección 3.4 para introducir el método KAOS. Podemos utilizar LTSA para chequear si el Modelo Operacional especificado para el `MinePumpController` de la Figura 22 satisface el objetivo `PumpOnWhenHighWaterANDNoMethane`. Recuerde que inicialmente consideramos la precondición requerida más débil (True) y la condición de triggering más fuerte (False) para cada operación, por lo que inicialmente no tenemos ninguna de las condiciones requeridas de la Figura 23. En este caso, LTSA retorna el siguiente contraejemplo:

```
Trace to property violation in PumpOnWhenHighWaterANDNoMethane:
  tick  (s0)
  aboveLow
  tick  (s1)
  aboveHigh
  tick        HighWater (s2)
  tick        HighWater (s3)
```

Debido a la forma en que las propiedades son expresadas (en relación al evento `tick`), los estados importantes son aquellos que siguen inmediatamente el evento tick. Este contraejemplo corresponde a un caso en el cual `(s2)` comienza con el nivel del agua alto, sin metano en el ambiente y la bomba apagada, y en la siguiente unidad de tiempo `(s3)` la situación no cambia y la bomba continúa apagada.

4.4.2 *Fase de Refinamiento*

Los contrajemplos generados por el model checker son utilizados para automática e iterativamente computar refinamientos durante el proceso de operacionalización de objetivos, los cuales garantizan remover los contraejemplos detectados.

Supongamos que obtenemos una traza T como contraejemplo, violando un objetivo particular G_i. Si construimos una fórmula F_T capturando la traza T, y una fórmula F_{G_i} capturando el hecho de que el objetivo G_i vale en el último estado, claramente $F_T \wedge F_{G_i}$ es insatisfactible. Podemos luego producir un interpolante desde estas fórmulas, es decir, una sentencia I, en la intersección de los lenguajes de la traza y el objetivo, tal que $F_T \Rightarrow I$ e $I \wedge F_{G_i}$ es insatisfactible. Intuitivamente, el interpolante I representa un contraejemplo "más débil" que

T, una condición alcanzable desde el estado inicial, el cuál nos conduce a la violación del objetivo. Así, mientras no removamos la propiedad I, no vamos a dejar de violar el objetivo G_i. Sin embargo, sólo con remover I, no podemos garantizar la satisfacción del objetivo G_i, pero no removerlo nos garantiza su violación. Note que el cómputo del interpolante I es, de alguna manera, una forma de generalización.

Un contraejemplo puede ser removido por alguna de las dos formas: *prohibiendo* la ocurrencia de una operación en ciertos estados (es decir, fortaleciendo su precondición requerida), ó *forzando* una operación a que ocurra en ciertos estados (es decir, debilitando su condición de triggering requerida). La técnica se ocupa de detectar automáticamente cuál de los casos es necesario, y usando el interpolante produce un cambio en la condición correspondiente. En este proceso de refinamiento de condiciones requeridas utilizando interpolantes, *weakest precondition* juegan un rol muy importante.

A continuación explicamos los detalles del proceso de Interpolación que realiza nuestra técnica. Luego explicaremos en que consiste el cómputo de weakest precondition, y finalmente describiremos como derivar una condición requerida a partir del interpolante computado.

Interpolación

El primer paso de la fase de refinamiento es computar un interpolante para el contraejemplo y el objetivo violado. Para esta tarea, vamos a utilizar MathSAT, un SMT Solver con muchas características, que en particular, cuenta con mecanismos eficientes para computar interpolantes. A pesar de ser un SMT Solver, en los casos de estudio de este capítulo no haremos uso de las teorías con las que esta equipado MathSAT. Es decir, que usaremos la herramienta sólo como un SAT Solver, con las capacidades de computar interpolantes. Puede encontrar una descripción de MathSAT y el algoritmo de interpolación que usaremos en las Subsecciones 2.2.2 y 2.2.3, respectivamente.

CODIFICACIÓN DEL CONTRAEJEMPLO. Veamos primero la codificación del contraejemplo en una fórmula proposicional. Como ya mencionamos, debido a la forma en que el Modelo Operacional es capturado en una especificación FSP (en relación al evento `tick`), los estados que nos interesan de la traza son los observables, es decir, los que siguen inmediatamente al evento `tick`.

Sean T la traza del contraejemplo y $Count(T, \texttt{tick}) = (N + 1)$ la cantidad de `tick` que ocurren en T. Note que, acorde a la codificación usada, siempre debe ocurrir que T comienza y termina con el evento `tick` (es decir, $T =$

`tick`; . . . ; `tick`), denotando al estado inicial (s_0) y final (s_N), respectivamente. Luego, los estados del sistema son representados por variables booleanas (una por cada fluente en la especificación), y estas variables son replicadas $N+1$ veces, para representar los $N+1$ estados diferentes de la traza $(s_0, \ldots, s_N)$. Por ejemplo, considerando el contraejemplo obtenido en la fase de Model Checking, cada variable debe ser replicada 4 veces, es decir, `PumpOn0`, ..., `PumpOn3` representarán el valor del fluente `PumpOn` en cada estado del contraejemplo.

Entre dos ticks consecutivos (digamos $\texttt{tick}_i$ y $\texttt{tick}_{i+1}$) pueden ejecutarse varios eventos que modifican el valor de las variables desde el estado s_i al s_{i+1}. Para capturar esta noción, vamos a definir una fórmula $R_{\texttt{F}}(i, i+1)$ que indica como se actualiza el valor de cada variable `F` del Modelo Operacional en s_{i+1}, dependiendo de los eventos que se ejecuten.

Definición 4.7 (Actualización de Variables)**.** *Sea* F *una variable del Modelo Operacional. Para el conjunto de operaciones* $\{op_1, \ldots, op_k\}$ *que controlan la variable* F *(es decir,* $(\neg)F \in (DomPost(op) \cup ReqPost(op))$*, para* $op \in \{op_1, \ldots, op_k\}$*), definimos la siguiente fórmula:*

$$\begin{aligned} R_F(i, i+1) = &((op_1^i \wedge (\neg)F_{i+1}) | (op_2^i \wedge (\neg)F_{i+1}) | \ldots | (op_k^i \wedge (\neg)F_{i+1}) \\ &| (\neg op_1^i \wedge \neg op_2^i \wedge \ldots \wedge \neg op_k^i \wedge F_{i+1} = F_i) \\ &) \end{aligned}$$

La notación op_j^i expresa que la operación op_j se ejecuta entre el $\texttt{tick}_i$ y $\texttt{tick}_{i+1}$. Luego, cuando esta operación se ejecuta (op_j^i es verdadero), entonces la variable `F` será verdadera en s_{i+1} si $\texttt{F} \in (DomPost(op) \cup ReqPost(op))$, o falsa en caso de que $\neg\texttt{F} \in (DomPost(op) \cup ReqPost(op))$. Si ninguna de las operaciones que controlan el valor de `F` se ejecutan (es decir, todas $op_1^i, \ldots, op_k^i$ son falsas), entonces `F` preserva el valor que tenía en el estado s_i ($\texttt{F}_{i+1} = \texttt{F}_i$).

Además, podemos definir $Init(0)$ que captura el valor inicial de cada variable `F`, acorde al valor inicial de cada fluente en su definición. Finalmente, debemos indicar los eventos que se ejecutan (y los que no) en el contraejemeplo, y en qué posición respecto a los ticks. Por ejemplo, `switchPumpOn0` indicará que la operación `switchPumpOn` se ejecuta entre el $\texttt{tick}_0$ y $\texttt{tick}_1$, mientras que `!belowLow2` indica que la operación ambiental `belowLow` no se ejecuta entre los $\texttt{tick}_2$ y $\texttt{tick}_3$. Note que esta codificación no requiere introducir las condiciones de dominio y requeridas restantes, ya que el contraejemplo retornado por LTSA es una ejecución factible del Modelo Operacional actual.

Utilizando esta codificación, veamos en la Figura 25 una especificación simplificada, en MSAT el lenguaje de MathSAT, del contraejemplo previo retornado en la fase de Model Checking.

```
VAR
#variables
LowWater0,...,LowWater3: BOOLEAN; HighWater0,...,HighWater3:BOOLEAN;
Methane0,...,Methane3: BOOLEAN;
PumpOn0,...,PumpOn3: BOOLEAN;    Alarm0,...,Alarm3: BOOLEAN;
#operaciones
signalMethane0,signalNoMethane0: BOOLEAN
aboveHigh0, belowHigh0, belowLow0, aboveLow0: BOOLEAN
switchPumpOn0, switchPumpOff0, raiseAlarm0, stopAlarm0: BOOLEAN
....
signalMethane2,signalNoMethane2: BOOLEAN
aboveHigh2, belowHigh2, belowLow2, aboveLow2: BOOLEAN
switchPumpOn2, switchPumpOff2, raiseAlarm2, stopAlarm2: BOOLEAN

FORMULA
#Init(0)
LowWater0 & !HighWater0 & !Methane0 & !PumpOn0 & !Alarm0
# R(0,1)
& ((aboveHigh0 & HighWater1) | (belowHigh0 & !HighWater1)
  | (!aboveHigh0 & !belowHigh0 & HighWater1=HighWater0))
& ((belowLow0 & LowWater1) | (aboveLow0 & !LowWater1)
  | (!belowLow0 & !aboveLow0 & LowWater1=LowWater0))
& ((signalMethane0 & Methane1) | (signalNoMethane0 & !Methane1)
  | (!signalMethane0 & !signalNoMethane0 & Methane1=Methane0))
& ((switchPumpOn0 & PumpOn1) | (switchPumpOff0 & !PumpOn1)
  | (!switchPumpOn0 &!switchPumpOff0 & PumpOn1=PumpOn0))
& ((raiseAlarm0 & Alarm1) | (stopAlarm0 & !Alarm1)
  | (!raiseAlarm0 & !stopAlarm0 & Alarm1=Alarm0))
# R(1,2)
...
# R(2,3)
#contraejemplo
& !signalMethane0 & !signalNoMethane0
& !belowLow0 & aboveLow0 & !aboveHigh0 & !belowHigh0
& !switchPumpOn0 & !switchPumpOff0 & !raiseAlarm0 & !stopAlarm0

& !signalMethane1 & !signalNoMethane1
& !belowLow1 & !aboveLow1 & aboveHigh1 & !belowHigh1
& !switchPumpOn1 & !switchPumpOff1 & !raiseAlarm1 & !stopAlarm1

& !signalMethane2 & !signalNoMethane2
& !belowLow2 & !aboveLow2 & !aboveHigh2 & !belowHigh2
& !switchPumpOn2 & !switchPumpOff2 & !raiseAlarm2 & !stopAlarm2
```

Figura 25: Especificación MSAT del Modelo Operacional del `MipePumpControler`

CODIFICACIÓN DE OBJETIVOS DE SAFETY. Recordemos que, durante esta sección, los objetivos de G son propiedades de safety. En [Sistla, 1999] se presenta una *sintaxis LTL para fórmulas de safety*. Es decir, cualquier propiedad de safety debe poder ser expresada utilizando este subconjunto sintáctico de las reglas de formación de fórmulas LTL, al que denotaremos como LTL_{safety}.

Definición 4.8 (Sintaxis LTL para Safety)**.** *Para toda proposición $p \in PA$. El conjunto de fórmulas LTL sintaćticas de* safety LTL_{safety} *es dado por:*

$$\varphi ::= \mathsf{T} \mid \mathsf{F} \mid p \mid \neg p \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \vee \varphi_2 \mid \bigcirc\varphi \mid \varphi_1 \mathcal{W} \varphi_2 \mid \Box\varphi$$

Respecto a la nomenclatura utilizada en esta tesis, en el paper [Sistla, 1999] se utiliza el símbolo $\mathcal{U}$ para referirse a nuestro operador $\mathcal{W}$ (weak until). Note que estas fórmulas de safety sólo pueden tener negación ($\neg$) sobre las proposiciones, es decir, están en *forma normal positiva* (PNF) (CNF -Definición 2.2- es un tipo particular de PNF). Cualquier fórmula LTL puede ser codificada en PNF [Baier and Katoen, 2008] (considerando además el operador temporal until $\mathcal{U}$). Sin embargo, aquellas que sólo utilicen los operadores temporales $\bigcirc$, $\mathcal{W}$ y $\Box$ serán de LTL_{safety}.

Por lo tanto, consideremos que cada objetivo $G_i \in G$ es expresado en LTL_{safety}. Por ejemplo, el objetivo `PumpOnWhenHighWaterANDNoMethane` de la fase de Model Checking puede ser expresado en LTL_{safety} como:

$$\Box((\texttt{HighWater} \wedge \neg\texttt{Methane}) \rightarrow \bigcirc\texttt{PumpOn}) \equiv \Box(\neg\texttt{HighWater} \vee \texttt{Methane} \vee \bigcirc\texttt{PumpOn})$$

Anteriormente mostramos como codificamos la traza del contraejemplo en una fórmula proposicional. Lo que necesitamos hacer ahora es capturar con una fórmula proposicional el hecho de que el objetivos G_i es válido en el último estado s_N. Para el objetivo anterior, la fórmula proposicional sería: $\neg\texttt{HighWater}_{N-1} \vee \texttt{Methane}_{N-1} \vee \texttt{PumpOn}_N$. Veamos a continuación como podemos construir dichas fórmulas automáticamente, cualquiera sea el objetivo de safety G_i.

Primero vamos a definir la noción de *prefijo-válido* para una propiedad de safety $\varphi \in LTL_{safety}$ y una cota $k \in \mathbb{N}$. Intuitivamente, un prefijo-válido para φ y k, es una fórmula proposicional capturando las condiciones que deben cumplir los primeros k estados de una traza, para no violar φ (provisto que φ es una propiedad de safety).

Definición 4.9 (Prefijo Válido)**.** *Sean $\varphi \in LTL_{safety}$, σ una interpretación LTL (una traza) y $k \in \mathbb{N}$. Diremos que σ es un* k-prefijo válido *de φ, si y sólo si, $(\sigma, 0) \models_k \varphi$, donde:*

$$
\begin{array}{ll}
(\sigma,i) \models_k \mathit{T} & (\sigma,i) \models_k p \ \mathit{iff}\ p \in \sigma(i) \\
(\sigma,i) \not\models_k \mathit{F} & (\sigma,i) \models_k \neg p \ \mathit{iff}\ p \notin \sigma(i) \\
(\sigma,i) \models_k \varphi_1 \vee \varphi_2 & \mathit{iff}\ (\sigma,i) \models_k \varphi_1 \ \acute{o}\ (\sigma,i) \models_k \varphi_2 \\
(\sigma,i) \models_k \varphi_1 \wedge \varphi_2 & \mathit{iff}\ (\sigma,i) \models_k \varphi_1 \ y\ (\sigma,i) \models_k \varphi_2 \\
(\sigma,i) \models_k \bigcirc\varphi & \mathit{iff} \begin{cases} (\sigma,i+1) \models_k \varphi & si\ i < k \\ \mathit{T} & si\ i \geq k \end{cases} \\
(\sigma,i) \models_k \Box\varphi & \mathit{iff}\ \forall j \cdot i \leq j \leq k \cdot (\sigma,j) \models_k \varphi \\
(\sigma,i) \models_k \varphi_1 \mathcal{W} \varphi_2 & \mathit{iff}\ \forall j \cdot i \leq j \leq k \cdot (\sigma,j) \models_k \varphi_1 \ \acute{o} \\
 & \quad \exists j \cdot i \leq j \leq k \cdot (\sigma,j) \models_k \varphi_2 \ y\ \forall l : i \leq l \leq j : (\sigma,l) \models_k \varphi_1
\end{array}
$$

Note que, para el caso particular de $\bigcirc\varphi$, cuando es evaluada en una posición $i \geq k$, la fórmula no puede ser violada dentro del prefijo k, ya que el operador $\bigcirc$ nos obligaría a evaluar φ fuera del prefijo de los primeros k estados.

Nuestra codificación de prefijos válidos en fórmulas proposicionales se basa fuertemente en la idea de *Bounded Model Checking* (BMC) [Biere et al., 1999]. Sin embargo, en BMC, la fórmula proposicional que construyen a partir de una propiedad LTL, captura la noción de *prefijos inválidos* (cuando la fórmula proposicional es SAT, entonces se encontró un contraejemplo de longitud k).

A continuación definimos $[\![\cdot]\!]_k^i$ una tranducción de fórmulas LTL de safety a fórmulas proposicionales. El parámetro k determina la longitud del prefijo e i la posición corriente en la traza. La fórmula proposicional obtenida por $[\![\varphi]\!]_k^0$, representara al conjunto de prefijos válidos de φ de longitud k.

Definición 4.10 (Traducción de Prefijos Válidos)**.** *Para una fórmula $\varphi \in LTL_{safety}$, $k, i \in \mathbb{N}$ con $i \leq k$, definimos:*

$$
\begin{array}{ll}
[\![\mathit{T}]\!]_k^i := \mathit{T} & [\![\mathit{F}]\!]_k^i := \mathit{F} \\
[\![p]\!]_k^i := p_i & [\![\neg p]\!]_k^i := \neg p_i \\
[\![\varphi_1 \vee \varphi_2]\!]_k^i := [\![\varphi_1]\!]_k^i \vee [\![\varphi_2]\!]_k^i & [\![\varphi_1 \wedge \varphi_2]\!]_k^i := [\![\varphi_1]\!]_k^i \wedge [\![\varphi_2]\!]_k^i \\
[\![\bigcirc\varphi]\!]_k^i := \begin{cases} [\![\varphi]\!]_k^{i+1} & si\ i < k \\ \mathit{T} & si\ i \geq k \end{cases} & [\![\Box\varphi]\!]_k^i := \bigwedge_{j=i}^{k} ([\![\varphi]\!]_k^j)
\end{array}
$$

$$
[\![\varphi_1 \mathcal{W} \varphi_2]\!]_k^i := \bigwedge_{j=i}^{k} ([\![\varphi_1]\!]_k^j) \vee \bigvee_{j=i}^{k} ([\![\varphi_2]\!]_k^j \wedge \bigwedge_{l=i}^{j} ([\![\varphi_1]\!]_k^l))
$$

Lema 4.1. *Sean $\varphi \in LTL_{safety}$, σ una interpretación LTL (una traza) y $k \in \mathbb{N}$. Si $(\sigma,0) \models_k \varphi$, entonces $\sigma[..k] \models_{LP} [\![\varphi]\!]_k^0$, donde $\sigma[..k] = \bigwedge_{i=0}^{k} \sigma(i)$.*

Demostración. La justificación de este lema procede directamente de que todos lo cuantificadores (universales y existenciales) de la semántica de prefijos-válidos están acotados por k. Luego, la función de traducción es exactamente idéntica a la semántica de prefijos-válidos, sin el "syntactic sugar" de los cuantificadores. □

Nuestra intención es capturar con una fórmula proposicional el hecho de que el objetivo G_i es válido en el último estado. Para esto, primero definamos una función $\delta : LTL_{safety} \rightarrow \mathbb{N}$ que retorna la profundidad de una fórmula (es decir, el número del máximo anidamiento del operador $\bigcirc$):

$$\begin{aligned}
\delta(\cdot) &= 0, \text{ con } \cdot \in \{\mathsf{T}, \mathsf{F}, p, \neg p\} \\
\delta(\varphi_1 \wedge \varphi_2) &= max\{\delta(\varphi_1), \delta(\varphi_2)\} \\
\delta(\varphi_1 \vee \varphi_2) &= max\{\delta(\varphi_1), \delta(\varphi_2)\} \\
\delta(\bigcirc\varphi) &= \delta(\varphi) + 1 \\
\delta(\varphi_1 \mathcal{W} \varphi_2) &= max\{\delta(\varphi_1), \delta(\varphi_2)\} \\
\delta(\Box\varphi) &= \delta(\varphi)
\end{aligned}$$

Finalmente, si N es la cantidad de estados del contraejemplo y $\delta(G_i) = M$, la fórmula proposicional $[\![G_i]\!]_N^{N-M}$ captura el hecho de que el objetivo G_i es válido en el último estado de la traza del contraejemplo (s_N).

Continuando con el objetivo `PumpOnWhenHighWaterANDNoMethane` de la fase de Model Checking, $\delta(\Box(\neg \texttt{HighWater} \vee \texttt{Methane} \vee \bigcirc \texttt{PumpOn})) = 1$. Luego, como el contraejemplo retornado por LTSA es de longitud 4, N debe ser 3 (para capturar los estados s_0, s_1, s_2, s_3). Finalmente, la fórmula proposicional que nos indica que el objetivo `PumpOnWhenHighWaterANDNoMethane` vale en el último estado es la siguiente:

$$\begin{aligned}
F_{\texttt{PumpOnWhenHighWaterANDNoMethane}} &= [\![\Box(\neg \texttt{HighWater} \vee \texttt{Methane} \vee \bigcirc \texttt{PumpOn})]\!]_3^2 \\
&= (\neg \texttt{HighWater2} \vee \texttt{Methane2} \vee \texttt{PumpOn3}) \\
&= (\texttt{HighWater2} \wedge \neg \texttt{Methane2} \rightarrow \texttt{PumpOn3})
\end{aligned}$$

COMPUTO DE INTERPOLANTES. Sea T la traza de contraejemplo que viola un objetivo de safety G_i. Como mostramos anteriormente, podemos construir una fórmula proposicional F_T que captura la traza T, y una fórmula proposicional F_{G_i} que captura el hecho de que el objetivo G_i vale en el último estado. Como $F_T \wedge F_{G_i}$ es insatisfactible, luego podemos computar automáticamente un interpolante I. Recuerde que, por definición, I debe ser una fórmula expresada en el lenguaje común de F_T y F_{G_i}, tal que $F_T \Rightarrow I$ e $I \wedge F_{G_i}$ es insatisfactible.

En nuestro caso, para el cómputo de interpolantes utilizaremos MathSAT, que implementa de manera automática y eficiente, el algoritmo de interpolación pre-

sentado en Subsección 2.2.2. Por ejemplo, el interpolante para el contraejemplo obtenido en la fase de Model Checking y la fórmula $F_{\texttt{PumpOnWhenHighWaterANDNoMethane}}$ es: $(\texttt{HighWater2} \wedge \neg\texttt{Methane2} \wedge \neg\texttt{PumpOn3})$

Intuitivamente, podemos pensar al interpolante I como una representación más débil del contraejemplo T, por lo tanto una propiedad alcanzable desde el estado inicial, cuya validez nos conduce a una violación del objetivo G_i. Así, mientras no removamos la propiedad I, no vamos a dejar de violar el objetivo G_i. Sin embargo, sólo con remover I no podemos garantizar la satisfacción de G_i, pero no removerlo nos garantiza su violación.

Weakest Precondition

Un contraejemplo puede ser removido por alguna de las dos formas: *prohibiendo* la ocurrencia de una operación en ciertos estados (es decir, fortaleciendo su precondición requerida), ó *forzando* una operación a que ocurra en ciertos estados (es decir, debilitando su condición de triggering requerida). Nuestra técnica automáticamente se encarga de detectar cuál de los casos es necesario utilizando *weakest precondition*.

Definición 4.11 (Weakest Precondition)**.** *Sea* $Trans(\overline{x}, \overline{x}')$ *una relación de transición y* $F \in LP$ *una fórmula proposicional. La* weakest precondition *(WP) de* F *sobre la relación de transición* $Trans(\overline{x}, \overline{x}')$ *es dada por:*

$$WP(F, Trans(\overline{x}, \overline{x}')) \stackrel{def}{=} \forall \overline{x}' \cdot Trans(\overline{x}, \overline{x}') \rightarrow F(\overline{x}')$$

Esta definición fue tomada de [Bradley and Manna, 2007, Subsección 12.1.2]. Consideramos que $\overline{x}$ representa todas las variables de un estado del sistema. Note que las variables libres de $WP(F, Trans(\overline{x}, \overline{x}'))$ serán de $\overline{x}$.

Para poder computar WP sobre el interpolante y el contraejemplo, vamos a utilizar la relación de transición definida anteriormente en la Definición 4.7. La relación de transición de la Definición 4.7 considera todas las posibles formas de actualizar cada variable, dependiendo de la operación que se ejecuta. Sin embargo, en nuestro caso vamos a computar WP sobre el interpolante y una operación específica (digamos op_i) que se ejecuta en el contrajemplo. Por lo tanto, si instanciamos la relación de transición de la Definición 4.7 indicando que operación del modelo se ejecuta (poniendo op_i en verdadero y op_j en falso, para todo $j \neq i$), obtenemos la siguiente relación de transición R_{op_i}.

$$R_{op_i}(\overline{x}, \overline{x}') = (DomPost(op_i) \wedge ReqPost(op_i) \wedge Unchanged_{op_i}(\overline{x}, \overline{x}'))$$

El predicado $Unchanged_{op_i}(\overline{x}, \overline{x}')$ expresa que toda variable $\texttt{F}$ que no es controlada por op_i ($\texttt{F} \notin DomPost(op_i) \cup ReqPost(op_i)$) mantiene el mismo valor

que el estado anterior ($\mathtt{F}' = \mathtt{F}$). Vemos por ejemplo, cuál sería la relación de transición que definen las operaciones controladas del Modelo Operacional del `MinePumpController`:

$$
\begin{aligned}
&R_{\mathtt{switchPumpOn}}(\overline{x}, \overline{x}') = \\
&\qquad (\mathtt{PumpOn}' = True \wedge \mathit{Unchanged}(\mathtt{Methane}, \mathtt{LowWater}, \mathtt{HighWater}, \mathtt{Alarm})) \\
&R_{\mathtt{switchPumpOff}}(\overline{x}, \overline{x}') = \\
&\qquad (\mathtt{PumpOn}' = False \wedge \mathit{Unchanged}(\mathtt{Methane}, \mathtt{LowWater}, \mathtt{HighWater}, \mathtt{Alarm})) \\
&R_{\mathtt{raiseAlarm}}(\overline{x}, \overline{x}') = \\
&\qquad (\mathtt{Alarm}' = True \wedge \mathit{Unchanged}(\mathtt{Methane}, \mathtt{LowWater}, \mathtt{HighWater}, \mathtt{PumpOn})) \\
&R_{\mathtt{stopAlarm}}(\overline{x}, \overline{x}') = \\
&\qquad (\mathtt{Alarm}' = False \wedge \mathit{Unchanged}(\mathtt{Methane}, \mathtt{LowWater}, \mathtt{HighWater}, \mathtt{PumpOn}))
\end{aligned}
$$

Supongamos entonces que estamos en la situación de la Sección 4.2, tenemos el interpolante $I = \mathtt{Methane} = True \wedge \mathtt{PumpOn}' = True$, y computemos WP respecto de la operación `switchPumpOn`. Para este caso, vamos a obtener $WP(I, R_{\mathtt{switchPumpOn}}(\overline{x}, \overline{x}')) = \mathtt{Methane}$. Veamos en detalle los pasos del computo de WP.

$$
\begin{aligned}
& WP(I, R_{\texttt{switchPumpOn}}(\overline{x}, \overline{x}')) \\
& = \{\text{por definición de } WP\} \\
& \forall \overline{x}' \cdot R_{\texttt{switchPumpOn}}(\overline{x}, \overline{x}') \rightarrow I \\
& = \{\text{por definición de } R_{\texttt{switchPumpOn}}(\overline{x}, \overline{x}') \text{ e } I\} \\
& \forall \overline{x}' \cdot (\texttt{PumpOn}' = True \wedge \mathit{Unchanged}(\texttt{Methane}, \texttt{LowWater}, \texttt{HighWater}, \texttt{Alarm})) \\
& \qquad \rightarrow \texttt{Methane} = True \wedge \texttt{PumpOn}' = True) \\
& = \{\text{por definición de } \overline{x}\} \\
& \forall \texttt{Methane}', \texttt{LowWater}', \texttt{HighWater}', \texttt{PumpOn}', \texttt{Alarm}' \cdot \\
& \quad (\texttt{PumpOn}' = True \wedge \mathit{Unchanged}(\texttt{Methane}, \texttt{LowWater}, \texttt{HighWater}, \texttt{Alarm})) \\
& \qquad \rightarrow \texttt{Methane} = True \wedge \texttt{PumpOn}' = True) \\
& = \{\text{por definición de } \mathit{Unchanged}\} \\
& \forall \texttt{Methane}', \texttt{LowWater}', \texttt{HighWater}', \texttt{PumpOn}', \texttt{Alarm}' \cdot \\
& \quad (\texttt{PumpOn}' = True \wedge \texttt{Methane}' = \texttt{Methane} \wedge \texttt{LowWater}' = \texttt{LowWater} \\
& \qquad \wedge \texttt{HighWater}' = \texttt{HighWater} \wedge \texttt{Alarm}' = \texttt{Alarm}) \\
& \qquad \rightarrow \texttt{Methane} = True \wedge \texttt{PumpOn}' = True) \\
& = \{\text{rango unitario para la variable } \texttt{Methane}' = \texttt{Methane}. \\
& \quad \text{Eliminación de cuantificación y simplificación.}\} \\
& \forall \texttt{LowWater}', \texttt{HighWater}', \texttt{PumpOn}', \texttt{Alarm}' \cdot \\
& \quad (\texttt{PumpOn}' = True \wedge \texttt{LowWater}' = \texttt{LowWater} \\
& \qquad \wedge \texttt{HighWater}' = \texttt{HighWater} \wedge \texttt{Alarm}' = \texttt{Alarm}) \\
& \qquad \rightarrow \texttt{Methane} = True \wedge \texttt{PumpOn}' = True) \\
& = \{\text{Aplicamos el paso anterior para eliminar la cuantificación sobre} \\
& \quad \texttt{LowWater}', \texttt{HighWater}', \texttt{Alarm}'\} \\
& \forall \texttt{PumpOn}' \cdot (\texttt{PumpOn}' = True \rightarrow \texttt{Methane} = True \wedge \texttt{PumpOn}' = True) \\
& = \{\text{Eliminamos la cuantificación sobre } \texttt{PumpOn}' \text{ y simplificamos.}\} \\
& \texttt{Methane} = True
\end{aligned}
$$

Fortaleciendo Precondiciones Requeridas

Una vez computado el interpolante, vamos a analizar la traza desde atrás hacia adelante, comenzando desde el último tick, computando la *weakest precondition* (WP) del interpolante con respecto a la última operación, tratando de encontrar el primer lugar donde la violación podría ser evitada. Sea I el interpolante y $T = a_1; a_2; \ldots; a_k$ la traza del contraejemplo. Podemos tener

dos situaciones con respecto a $WP(a_k, I) = I'$, la weakest precondition entre la última operación y el interpolante. Si $\neg(I' \Rightarrow I)$, entonces el interpolante no vale antes de ejecutar a_k. Así, si prohibimos a_k de ocurrir cuando I' vale (agregando $\neg I'$ a la precondición requerida de a_k), removeríamos este contraejemplo. Esto puede hacerse, siempre y cuando no contradiga la condición de triggering de a_k (es decir, si la operación a_k no está obligada a ejecutarse cuando I' vale). Si $I' \Rightarrow I$, entonces por más de que prohibamos la ejecución de a_k, vamos a seguir violando el objetivo, ya que I sigue valiendo. En este caso, así como cuando al refinar la precondición requerida con $\neg I'$ contradice la condición de triggering requerida, debemos seguir buscando hacia atrás en la traza, para tratar de encontrar otra operación cuya ocurrencia cause la satisfacción del interpolante, y pueda ser "removida" de la traza. Note que la modificación realizada a la precondición requerida de a_k no garantiza que G_i no puede ser violado; sólo previene su violación al ejecutar a_k cuando I' vale.

Considere el siguiente objetivo `PumpOffWhenLowWater` que debe satisfacer el `MinePumpController`, y el contraejemplo retornado por LTSA, que serán utilizados para mostrar un ejemplo de fortalecimiento de una precondición requerida:

```
assert PumpOffWhenLowWater =
     □(tick→(LowWater → ◯(¬tick 𝒲 (tick ∧ ¬PumpOn))))
```

```
Trace to property violation in PumpOffWhenLowWater:
  tick         LowWater (s0)
  switchPumpOn LowWater ∧ PumpOn
  tick         LowWater ∧ PumpOn (s1)
```

En este caso `switchPumpOn` es la única operación diferente a tick que es ejecutada, la cuál inicialmente tiene como precondición requerida True y condición de triggering requerida False. El interpolante en este caso es $\texttt{LowWater} \wedge \texttt{PumpOn}'$. Yendo hacia atrás desde el último tick, podemos computar la weakest precondition de este interpolante con respecto a la última operación diferente a tick, `switchPumpOn`, obteniendo `LowWater`. Note que $\neg(\texttt{LowWater} \Rightarrow \texttt{LowWater} \wedge \texttt{PumpOn}')$. Luego, es posible falsificar el interpolante previniendo la ocurrencia de `switchPumpOn` cuando vale `LowWater`. Esto se logra agregando $\neg$`LowWater` a la precondición requerida de `switchPumpOn`:

```
Operation switchPumpOn
  ...
  ReqPre for PumpOffWhenLowWater
      ¬ LowWater
```

Este refinamiento puede realizarse, siempre y cuando no contradiga la condición de triggering requerida de la operación `switchPumpOn`. Para chequear si el nuevo conyunto identificado para una precondición requerida, contradice o no la condición de triggering requerida, consideramos la ecuación (KAOS meta-rule) de la Sección 4.3, la cuál siempre debe ser válida (es una meta regla del lenguaje KAOS [Dardenne et al., 1993]). Luego, si al modificar la precondición requerida con el nuevo conyunto (I') violamos esta propiedad, concluimos que el nuevo conyunto es contradictorio con la condición de triggering requerida o la precondición del dominio. Por lo tanto, debemos seguir computando WP hacia atrás en la traza utilizando I' en vez de I.

Para el ejemplo anterior, el chequeo que realizamos es el siguiente:

$$\begin{aligned} &\mathit{ReqTrig}(\texttt{switchPumpOn}) \wedge \mathit{DomPre}(\texttt{switchPumpOn}) \Rightarrow \mathit{ReqPre}(\texttt{switchPumpOn}) \\ &\qquad \equiv \mathit{False} \wedge \neg\texttt{PumpOn} \Rightarrow \texttt{LowWater} \end{aligned}$$

Como la fórmula anterior es válida, finalmente modificamos la precondición requerida de `switchPumpOn`, que se convierte en $\neg$`LowWater`.

En el proceso recién descripto, tenemos que realizar varios chequeos lógicos: chequear si una weakest precondition implica o no al interpolante y si la nueva precondición requerida no contradice a la condición de triggering requerida. La técnica realiza estos chequeos mediante el uso de SAT Solving. Note que chequear si la fórmula $A \Rightarrow B$ es válida, es equivalente a chequear que $A \wedge \neg B$ es insatisfactible.

Debilitando Condiciones de Triggering Requeridas

En la situación anterior, el contraejemplo es removido previniendo una operación que *aparece* en la traza de ser ejecutada. En otras situaciones, no es posible remover el contraejemplo restringiendo la ejecución de ciertas operaciones, debido a alguna de las siguientes razones: *(i)* la operación no es controlada por el agente de software; *(ii)* la operación está obligada a ejecutarse ya que vále su condición de triggering requerida; o *(iii)* antes de ejecutar la operación ya era válido el interpolante (es decir, es el caso de $I' \Rightarrow I$), por lo que prevenir su ejecución no evitará alcanzar el interpolante. Por lo tanto, la solución para esta situación consiste en *forzar* una operación que no fue ejecutada, a que ocurra cuando debe hacerlo. Para esto, debemos *debilitar* la condición de triggering requerida de dicha operación.

Sea $T = a_1; \ldots; a_i; \texttt{tick}; a_j; \ldots; a_n; \texttt{tick}$ la traza del contraejemplo e I el interpolante computado para este contraejemplo. Supongamos que no podemos remover el contraejemplo mediante el fortalecimiento de alguna operación en $a_j; \ldots; a_n$, por alguna de las razones explicadas anteriormente. Luego, como

no podemos prevenir ninguna operación de ocurrir de $\{a_j, \ldots, a_n\}$, vamos a tratar de remover el contraejemplo forzando una operación a ejecutarse cuando valga $I'' = WP(I, a_j; \ldots; a_n)$[1]. En caso de que no se ejecute ninguna operación entre los últimos ticks ($\{a_j, \ldots, a_n\} = \emptyset$), consideraremos que siempre se ejecuta `skip` ($R_{\texttt{skip}}(\overline{x}, \overline{x}') = \bigwedge(x' = x)$, con $x \in \overline{x}, x' \in \overline{x}'$) manteniendo el mismo valor de las variables del estado anterior. De esta manera, I'' siempre va a ser un predicado de estado sobre $\overline{x}$.

Para que la operación, llamemosla a_t, pueda ser disparada cuando vale I'', a_t debe satisfacer las siguientes dos condiciones:

- a_t debe estar habilitada a ejecutarse cuando vale I'':

$$I'' \Rightarrow DomPre(a_t) \wedge ReqPre(a_t). \tag{5}$$

- la ejecución de a_t cuando vale I'', debe falsificar el intepolante I:

$$(I'' \wedge \bigcirc DomPost(a_t)) \Rightarrow \neg I. \tag{6}$$

Note que I'' es un predicado sobre $\overline{x}$, $\bigcirc DomPost(a_t)$ nos dará un predicado sobre $\overline{x}'$, y el intepolante se refiere a $\overline{x}'$ (aunque, como en los ejemplos anteriores, también puede referirse a $\overline{x}$).

La técnica evalúa todas las operaciones *controladas* que no aparecen en $a_j; \ldots; a_n$, chequeando si alguna satisface las dos condiciones. Si encontramos tal operación a_t, refinamos su condición de triggering requerida como sigue:

$$ReqTrig(a_t) = ReqTrig^{pre}(a_t) \vee I'',$$

donde $ReqTrig^{pre}(a_t)$ es la condición de triggering requerida corriente de a_t. Note que agregar el nuevo disyunto I'' a la condición de triggering requerida de a_t no viola la meta regla de KAOS (KAOS meta-rule), ya que a_t satisface la condición (5).

Si ninguna operación satisface las condiciones (5) y (6), seguiremos mirando hacia atrás en la traza, desde a_i, y usando I'' en vez de I.

Considere el contraejemplo al objetivo `PumpOnWhenHighWaterANDNoMethane`, mostrado en la Subsección 4.4.1. En este caso, el interpolante computado es $\texttt{HighWater} \wedge \neg\texttt{Methane} \wedge \neg\texttt{PumpOn}'$. Note que, como ninguna operación es ejecutada entre los últimos eventos tick, $I'' = WP(I, \texttt{skip}) = \texttt{HighWater} \wedge \neg\texttt{Methane} \wedge \neg\texttt{PumpOn}$. Vamos a pasar chequear si existe alguna operación que cumpla con las dos condiciones (5) y (6). En particular, la operación `switchPumpOn`

[1] Se define inductivamente $WP(I, a_j; \ldots, a_n) \Leftrightarrow WP(WP(I, a_n), a_j; \ldots; a_{n-1})$.

puede ser ejecutada cuando $\texttt{HighWater} \wedge \neg\texttt{Methane} \wedge \neg\texttt{PumpOn}$, y su ejecución falsifica el interpolante I. Concretamente, considerando $\neg\texttt{LowWater}$ y False como la precondición requerida y condición de triggering requerida actuales, respectivamente, de la operación `switchPumpOn`, las siguientes fórmulas, que se corresponden a las condiciones (5) y (6), son válidas:

$$\texttt{HighWater} \wedge \neg\texttt{Methane} \wedge \neg\texttt{PumpOn} \Rightarrow \neg\texttt{PumpOn} \wedge \neg\texttt{LowWater} \tag{7}$$

$$\begin{aligned}&(\texttt{HighWater} \wedge \neg\texttt{Methane} \wedge \neg\texttt{PumpOn}) \wedge \\ &(\texttt{PumpOn'} = \mathit{True} \wedge \texttt{Methane'=Methane} \wedge \texttt{HighWater'=HighWater}) \\ &\quad \Rightarrow \neg(\texttt{HighWater} \wedge \neg\texttt{Methane} \wedge \neg\texttt{PumpOn'})\end{aligned} \tag{8}$$

Luego, la condición de triggering requerida de `switchPumpOn` es refinada como sigue:

$$\mathit{ReqTrig}(\texttt{switchPumpOn}) = \texttt{HighWater} \wedge \neg\texttt{Methane} \wedge \neg\texttt{PumpOn}$$

Al igual que para el caso de fortalecimiento de precondiciones requeridas, debilitar condiciones de triggering requeridas también involucra varios chequeos lógicos para garantizar la consistencia del refinamiento, los cuáles son realizados utilizando SAT Solving.

4.4.3 *Refinamiento Iterativo*

Las dos fases recién descritas se corresponden a una sola iteración de nuestra técnica de refinamiento. Estas fases son iterativamente aplicadas hasta que no se detectan más violaciones en la fase de Model Checking, lo que significa que la especificación satisface los objetivos, o hasta que alcanzamos un punto donde, solo mediante el refinamiento de precondiciones requeridas y condiciones de triggering requeridas, los objetivos no pueden ser alcanzados.

Sea O y G la especificación operacional inicial y el conjunto de objetivos. Discutamos acerca de la correctitud de la operacionalización obtenida, e incompletitud y terminación de nuestra técnica.

Operacionalización Correcta

Cuando el enfoque termina, su salida es $\mathit{Spec} = O \cup \mathit{Req}$, donde $\mathit{Req} = \{R_1, \ldots, R_n\}$ es el conjunto de pre/triggering condiciones requeridas computadas.

Veamos que *Req* garantiza ser una extensión consistente de O que operacionaliza correctamente los objetivos en G.

Teorema 4.2 (Consistencia de la Operacionalización)**.** *Sean* $\{R_1^O, \ldots, R_m^O\}$ *las condiciones requeridas del Modelo Operacional inicial* O*, y* G *un conjunto de objetivos de safety. Nuestra técnica computa* $\{R_1, \ldots, R_n\}$ *un conjunto finito de condiciones requeridas que serán* consistentes *entre si. Formalmente,*

$$\bigwedge_{i=0}^{m} [\![R_i^O]\!] \wedge \bigwedge_{i=0}^{n} [\![R_i]\!] \not\models false$$

Demostración. Para demostrar este teorema, veamos que en cada iteración de nuestra técnica, la condición requerida agregada es consistente con el Modelo Operacional corriente. Procedamos por inducción en n.

Caso Base: Como el Modelo Operacional inicial O es un input para nuestra técnica, vamos a asumir que todas las condiciones requeridas R_i^O son consistentes, es decir, $\bigwedge_{i=0}^{m} [\![R_i^O]\!] \not\models false$.

Caso Inductivo: Como hipótesis inductiva, vamos a considerar que hasta la k-ésima iteración el Modelo Operacional es consistente:

$$HI = \bigwedge_{i=0}^{m} [\![R_i^O]\!] \wedge \bigwedge_{i=0}^{k} [\![R_i]\!] \not\models false$$

Veamos entonces, que la nueva condición requerida R_{k+1} agregada en la $k+1$-ésima iteración sigue siendo consistente con el Modelo Operacional. Nuestra técnica sólo puede refinar precondiciones o condiciones de triggering requeridas, por lo tanto, consideremos esos dos casos para R_{k+1}:

- Supongamos que R_{k+1} es una nueva precondición requerida para *op*, es decir, $ReqPre(op) = ReqPre^{pre}(op) \wedge R_{k+1}$ (cuyas precondiciones requeridas antes del refinamiento son $ReqPre^{pre}(op) \subseteq \bigcup_{i=0}^{m} R_i^O \cup \bigcup_{i=0}^{k} R_i$). Como R_{k+1} es una precondición, lo que hace es prohibir la ejecución de la operación *op* en ciertos estados. Sin embargo, para evitar introducir alguna inconsistencia, debemos verificar que *op* no está obligada a ejecutarse en los estados que prohíbe R_{k+1}. Nuestra técnica, antes de producir efectivamente el refinamiento, verifica la condición (KAOS meta-rule). Su validez nos garantiza que la nueva precondición requerida R_{k+1} no contradice su condición de triggering requerida. Por lo tanto, la operacón *op* seguirá estando habilitada a ejecutarse en los estados en los que estába obligada a hacerlo antes del refinamiento. Luego, agregar la precondición requerida R_{k+1} sigue garantizando la consistencia de nuestra técnica.

- Supongamos ahora que R_{k+1} es una condición de triggering requerida para op, es decir, $ReqTrig(op) = ReqTrig^{pre}(op) \vee R_{k+1}$ (cuyas condiciones de triggering requeridas corrientes son $ReqTrig^{pre}(op) \subseteq \bigcup_{i=0}^{m} R_i^O \cup \bigcup_{i=0}^{k} R_i$). Note que nuestra técnica verifica si R_{k+1} cumple con la condición (5). Su validez nos asegura que la nueva condición de triggering R_{k+1} no contradice las precondiciones (de dominio y requeridas) de la operación op. Por lo tanto, por más que ahora estará obligada a ejecutarse en más estados (al agregar R_{k+1}), antes del refinamiento la operación ya estaba habilitada a ejecutarse en esos estados, y no había inconsistencia alguna, ya que vale la hipótesis inductiva HI. Más precisamente, supongamos que existe una condición $R \in \bigcup_{i=0}^{m} R_i^O \cup \bigcup_{i=0}^{k} R_i$ inconsistente con R_{k+1}. Como vale la condición (5), ocurre que $R_{k+1} \Rightarrow DomPre(op) \wedge ReqPre(op)$. Por lo tanto, R ya era inconsistente con $DomPre(op) \wedge ReqPre(op)$ antes de la iteración $k+1$. Pero esto contradice la hipótesis HI, por lo que no puede existir dicha condición R inconsistente con R_{k+1}.

□

Acabamos de demostrar que las condiciones requeridas computadas por nuestra técnica son consistentes. Para probar que nuestra técnica produce una correcta operacionalización de los objetivos de G (cuando termina), nos falta demostrar que las condiciones requeridas computadas garantizan G. Este punto es llamado *completitud* de la operacionalización (ver Sección 4.3), pero no tiene que ver con la completitud del algoritmo que sigue nuestra técnica (de hecho, mostraremos que es incompleto).

Teorema 4.3 (Completitud respecto a Objetivos)**.** *Sean* $\{R_1^O, \ldots, R_m^O\}$ *las condiciones requeridas del Modelo Operacional inicial* O *y* G *un conjunto de objetivos de safety. Cuando nuestra técnica finaliza, computa un conjunto de condiciones requeridas* $\{R_1, \ldots, R_n\}$ *garantizando que:*

$$\bigwedge_{i=0}^{m} [\![R_i^O]\!] \wedge \bigwedge_{i=0}^{n} [\![R_i]\!] \models G$$

Demostración. La validez de este teorema recae en que consideramos que la traducción del Modelo Operacional a un Modelo de Comportamiento presentada en [Letier et al., 2008] es correcta, y en la correctitud del algoritmo de model checking que LTSA implementa [Magee and Kramer, 2006]. Note además, que el enunciado del teorema asúme que nuestra técnica terminó, y computó un conjunto de condiciones requeridas $\{R_1, \ldots, R_n\}$. Es decir, nuestra técnica iteró n veces computando cada una de las condiciones requeridas R_i, con $0 \leq i \leq n$. En

la iteración $n+1$, la fase de Model Checking no encuentra ninguna violación a los objetivos de G, por lo que nuestra técnica finaliza retornando las n condiciones requeridas computadas. □

Los dos teoremas anteriores nos aseguran que, si nuestra técnica termina, entonces el Modelo Operacional computado será una *operacionalización correcta* de los objetivos. Más adelante vamos a discutir sobre la minimalidad de la operacionalización que puede obtener nuestra técnica.

Incompletitud de la Técnica

La técnica presentada no satisface completitud. Es decir, si existe un refinamiento (modificaciones a las condiciones requeridas) que puede satisfacer el conjunto de objetivos, nuestro enfoque puede fallar en el proceso de encontrar dicho refinamiento. La principal razón esta relacionada al hecho de que los objetivos pueden competir entre sí, es decir, tratando de satisfacer un objetivo puede impedirnos de satisfacer otro más adelante. Más precisamente, al remover los contraejemplos, la técnica puede remover transiciones del comportamiento del sistema que más tarde podrían ser necesario agregarlas para remover otro contraejemplo. Sin embargo, note que nuestra técnica no puede agregar transiciones, ya que las postcondiciones no son modificadas, ni nuevos estados son agregados, ya que las definiciones de las variables no son alteradas.

Terminación

Tal como ya hemos discutido en la fase de Refinamiento, cada refinamiento que realiza nuestra técnica remueve la traza del contraejemplo del sistema, en el sentido de que la misma violación no podrá ser obtenida. Sin embargo, puede haber otras trazas violando el mismo objetivo. Por lo que no podemos garantizar *terminación* fácilmente. Discutamos esto, para el caso de objetivos de *safety.*

Teorema 4.4 (Terminación para Objetivos de Safety)**.** *Nuestra técnica garantiza terminación para la de operacionalización de objetivos de safety.*

Demostración. Llamemos O al Modelo Operacional corriente y G al conjunto de objetivos de safety. La fase de Model Checking construye un LTS que corresponde, esencialmente, a la fórmula $O \wedge \neg G$. Si esta fórmula tiene trazas que la satisfacen, entonces estas son contraejemplos. Primero, note que el LTS para $O \wedge \neg G$ es *finito*: tiene un número finito de estados, y por supuesto, un número finito de transiciones. Tenemos que demostrar que cada refinamiento a las condiciones requeridas remueve *al menos* una transición, ya que existe la

posibilidad de agregar conyuntos/disyuntos redundantes, que no removerían ninguna transición.

Sea $T = a_1; \ldots; a_i; \texttt{tick}; a_j; \ldots; a_n; \texttt{tick}$ la traza del contraejemplo que retorna la fase de Model Checking. Veamos las dos situaciones que nuestra técnica considera:

- Cuando una precondición requerida es agregada, es porque una operación $a_k \in \{a_j; \ldots; a_n\}$ fue identificada, cuya ejecución en un estado que satisface I', nos conduce a un estado donde vale el interpolante I (donde $I' = WP(I, a_k; \ldots; a_n)$). Además, esta operación a_k y condición I' provienen del contraejemplo T, indicando que I' es una condición alcanzable, donde a_k es ejecutada. Por lo tanto, cuando fortalecemos la precondición requerida de a_k con $\neg I'$, efectivamente estamos removiendo esa transición particular.
- Al debilitar condiciones de triggering requeridas, también removemos transiciones. Cuando una condición de triggering es agregada, es porque una operación $a_t \notin \{a_j; \ldots; a_n\}$ ha sido identificada, la cual puede ser ejecutada cuando vale I'', y su ejecución falsifica el interpolante I (donde $I'' = WP(I, a_j; \ldots; a_n)$). Note que, al debilitar la condición de triggering requerida de a_t, las transiciones que removemos corresponden a la ejecución del evento `tick`. Antes del refinamiento, a partir de I'' se ejecutaban $a_j; \ldots; a_n$ y `tick` estaba habilitado a ejecutarse. Sin embargo, luego del refinamiento, cuando vale I'' la operación a_t obligadamente debe ocurrir, más allá de si ocurren las operaciones $a_j; \ldots; a_n$. Por lo tanto, la transición en la que `tick` ocurre, sin haber ejecutado a_t cuando I'' valía, es removida del sistema. Así, nuevamente nuestra técnica produce una eliminación de alguna transición del LTS.

Debido a que el número de transiciones del LTS que corresponde a $O \wedge \neg G$ es finito, y sólo alteramos este LTS removiendo aristas, podemos garantizar que nuestro proceso de refinamiento termina cuando analizamos objetivos de safety. □

4.5 ANALIZANDO PROPIEDADES DE LIVENESS

4.5.1 *Time Progress*

Supongamos que tenemos un Modelo Operacional O que operacionaliza correctamente, por ejemplo, el objetivo `PumpOffWhenLowWater` mencionado en la sección anterior. Esto significa que todas las trazas del LTS que representa O

(denotado como $\mathcal{L}_O$), satisfacen el objetivo `PumpOffWhenLowWater`. Sin embargo, el LTS $\mathcal{L}_O$ podría contener trazas en las cuales a partir de un estado, el tiempo (`tick`) no progresa, como:

$$\texttt{tick}; \texttt{aboveLow}; \texttt{tick}; \texttt{belowLow}; \texttt{tick}; \texttt{switchPumpOn}; \ldots$$

En esta traza ocurren los primeros tres ticks, y luego `tick` no vuelve a ocurrir. Por la forma en que se caracterizan los objetivos respecto de `tick`, este tipo de trazas satisfacen el objetivo `PumpOffWhenLowWater` de forma vacía. Es decir, al no progresar el evento `tick`, los objetivos no pueden ser evaluados en los estados observables (ver Sección 4.3), por lo tanto tampoco pueden ser violados. Note que si `tick` ocurriera después de la ejecución de `switchPumpOn`, entonces el objetivo `PumpOffWhenLowWater` sería violado.

Por otro lado, supongamos que O también operacionaliza el objetivo `PumpOnWhenHighWaterANDNoMethane`. Luego, el LTS $\mathcal{L}_O$ podría contener trazas como:

$$\texttt{tick}; \texttt{aboveLow}; \texttt{tick}; \texttt{switchPumpOn}; \ldots$$

Note en este caso que, a pesar de que la traza también satisface los objetivos de manera vacía, la ocurrencia de `tick` no produciría ninguna violación. Aunque, el progreso del evento `tick` sigue sin ser garantizado por O.

En sistemas de tiempo discreto, una propiedad común deseada es la del progreso del tiempo [Letier et al., 2008; Alrajeh et al., 2013], usualmente especificada como $\Box\Diamond$`tick`. Esta propiedad es llamada *Time Progress* (TP) y es un tipo particular de propiedades de progreso. A pesar de que el agente de software no tenga control sobre el evento `tick` (de hecho es el "reloj global" del sistema el encargado de que progrese), las violaciones de TP usualmente nos conducen a requisitos operacionales adicionales o asunciones faltantes sobre el ambiente [Alrajeh et al., 2013]. Por lo tanto, en esta sección mostraremos que nuestra técnica puede ser extendida para operacionalizar este tipo particular de propiedades.

Model Checking para TimeProgress

Para analizar la validez de TP sobre el Modelo Operacional, vamos a utilizar el model checker LTSA. Para esto, la propiedad TP es especificada como:

```
progress TP = {tick}
```

En LTSA, el comando `progress` indica que `tick` debe ejecutarse *infinitas veces* en cada traza del LTS [Magee and Kramer, 2006]. Para verificar esto, LTSA primero computa todos los *conjuntos terminales* del LTS. Sea el LTS

$\mathcal{L}_O = \langle Q, A, \delta, q_0 \rangle$. Un conjunto terminal $S \subseteq Q$ es un conjunto de estados, donde cada estado de S es alcanzable desde cualquier otro estado de S, en una o más transiciones, y no existen transiciones que salgan a un estado fuera de S. Si el evento `tick` pertenece a todos los conjuntos terminales, entonces TP es verificada como válida. Si `tick` no pertenece a algún conjunto terminal, entonces LTSA nos mostrará como contraejemplo, la traza que nos lleva a alcanzar dicho conjunto terminal. Sin embargo, LTSA por defecto asume *strong fairness* para decidir que evento es ejecutado en cada traza. Es decir, si un conjunto de transiciones están habilitadas infinitas veces a ejecutarse, entonces serán ejecutas infinitas veces. Luego, si analizamos TP bajo esta asunción, puede que haya trazas que violan la propiedad, pero no serán consideradas por LTSA.

Por lo tanto, para detectar violaciones de progreso, tendremos en cuenta las siguientes tres asunciones: *(i) preservación de la semántica de KAOS:* los eventos que inicializan o terminan un fluente pueden ocurrir sólo una vez entre dos ticks; *(ii) asunción de Máximo Progreso (MP):* una asunción común en los sistemas reactivos que le da prioridad a los eventos del software sobre todos los otros eventos incluyendo `tick` [de Roever and Hooman, 1989]; y *(iii) Garantía de Safety:* no queremos interferir en la satisfacción de los objetivos de safety para operacionalizar la propiedad de Time Progress. Por lo tanto, el chequeo es realizado sobre el sistema que representa la composición de todas las condiciones de dominio/requeridas y los objetivos de safety.

Para garantizar la asunción *(i)*, vamos a generar un proceso que asegura que un fluente no puede ser modificado más de una vez entre dos ticks. Sea `F` un fluente del Modelo Operacional O, y `F_Actions` $= \{e_0, \ldots, e_n\}$ el conjunto de eventos que modifican el valor de `F`, definimos el siguiente proceso en FSP:

```
set F_Actions = {e0, ..., en}
Interleaving_for_F_Actions = (tick ->S0),
  S0 = (tick -> S0 | {F_Actions} ->tick -> S0).
```

Este proceso fuerza a que los eventos de `F_Actions` ocurran a lo sumo una vez entre dos ticks consecutivos. Por ejemplo, para el fluente `PumpOn` definimos el siguiente proceso:

```
set PumpOn_Actions = {switchPumpOn, switchPumpOff}
Interleaving_for_PumpOn_Actions = (tick ->S0),
  S0 = (tick -> S0 | {PumpOn_Actions} ->tick -> S0).
```

Para garantizar la asunción de Máximo Progreso *(ii)*, utilizamos el operador de priorización de acciones (<<) que provee LTSA, para darle prioridad a los eventos del agente de software. Es decir, siempre que pueda, el software ejecutará sus acciones antes del siguiente `tick`. Además, como los objetivos de safety G

son expresados en FLTL, podemos utilizar el comando `constraint` de LTSA para construir un LTS $\mathcal{L}_G$, que puede componerse con el Modelo Operacional $\mathcal{L}_O$, y así garantizar la asunción *(iii)*.

Sea $\mathcal{L}_{Inter}$ el LTS formado por la composición de todos los procesos que garantizan la asunción *(i)*, y `ControlledActions` el conjunto de eventos controlados por el software. Finalmente, realizaremos la verificación de la propiedad TP sobre el siguiente LTS $\mathcal{L}_{TP}$:

$$\mathcal{L}_{TP} = (\mathcal{L}_O \parallel \mathcal{L}_{Inter} \parallel \mathcal{L}_G) << \{\texttt{ControlledActions}\}$$

Considere la especificación FSP para el `MinePumpController` de la Figura 23, en particular el proceso compuesto `MinePump` vendría a ser $\mathcal{L}_O$. La Figura 26 muestra una versión simplificada de lo que necesitamos agregar a la especificación de la Figura 23 para poder verificar el progreso de `tick` sobre el Modelo Operacional del `MinePumpController`.

En particular, para la especificación FSP de la Figura 26, LTSA encuentra la siguiente violación a la propiedad de progreso TP:

```
Progress violation: TP
Trace to terminal set of states:
  tick
  switchPumpOn
  raiseAlarm
  aboveLow
  signalMethane
Cycle in terminal set:
Actions in terminal set:
  {}
Progress Check in: 0ms
```

Veamos a continuación como podemos extender nuestra técnica para remover este tipo de violaciones, y lograr que el evento `tick` progrese.

Operacionalización para TimeProgress

En la violación retornada por LTSA, podemos ver que primero se enciende la bomba y se dispara la alarma (note que `switchPumpOn` y `raiseAlarm` tienen precedencia por ser controlados), y luego el nivel del agua supera Low y detecta niveles críticos de metano en el ambiente. Pero sin embargo, el evento `tick` no progresa, ya que `tick` no aparece en el conjunto terminal. De hecho, la violación

```
...
||MinePump = (....).

||Interleaving = (Interleaving_for_Methane_Actions
             ||  Interleaving_for_WaterLevel_Actions
             ||  Interleaving_for_PumpOn_Actions
             ||  Interleaving_for_Alarm_Actions
             ).
constraint PumpOnWhenHighWaterANDNoMethane =
  [] (tick -> ((HighWater && !Methane) ->X (! tick W (tick && PumpOn))))

constraint PumpOffWhenLowWater =
  [] (tick -> ((LowWater) -> X (! tick W (tick && !PumpOn))))

constraint PumpOffWhenMethane =
  [] (tick -> ((Methane) -> X (! tick W (tick && !PumpOn))))

constraint AlarmWhenMethane =
  [] (tick -> ((Methane) -> X (! tick W (tick && Alarm))))

||GOALS = (PumpOnWhenHighWaterANDNoMethane || PumpOffWhenLowWater
       || PumpOffWhenMethane ||AlarmWhenMethane).

set ControlledActions = {switchPumpOn,switchPumpOff,raiseAlarm,stopAlarm}

||System_TP = (MinePump || Interleaving || GOALS)<<{ControlledActions}.

progress TP = {tick}
```

Figura 26: Especificación FSP para verificar TP sobre el `MinePumpController`.

muestra una traza que alcanza un conjunto terminal vació (`{}`). Por lo tanto, esta violación se corresponde a un *"deadlock"* en el sistema: un punto a partir del cuál ningún evento puede ser ejecutado (no sólo `tick`).

Dado que tenemos un número finito de eventos y la asunción *(i)*, la cual prohíbe la repetición de eventos entre ticks, las violaciones de TP reportadas por LTSA, siempre serán *"deadlocks"*. Básicamente, la principal razón que nos lleva al deadlock es la forma en que especificamos las propiedades, es decir, los objetivos deben ser satisfechos en los estados en los que ocurre `tick`. Luego, el no progreso de tick se debe a que su ejecución violaría un objetivo, contradiciendo así a la asunción *(iii)*. Esta misma observación fué reportada en [Alrajeh et al., 2013].

Por lo tanto, para remover las violaciones de TP, nuestra técnica consiste en extender la traza del contraejemplo con un evento `tick` al final, y procedemos a computar un interpolante para el contraejemplo extendido y los objetivos de safety (que serán violados en el último estado, ya que ahora `tick` es ejecutado). Intuitivamente, este interpolante explica las razones de porque `tick` no puede progresar, y nos da una propiedad alcanzable del sistema que debemos remover para contribuir a la satisfacción de los objetivos de safety y la propiedad Time Progress. Para remover este contraejemplo, utilizamos la misma fase de Refinamiento de la técnica presentada en Sección 4.4.

Por ejemplo, al extender la violación de TP anterior con un `tick`, el objetivo `PumpOffWhenLowWater` será violado. Por lo que, para este caso, la fase de Refinamiento computará una precondición requerida para la operación `switchPumpOn` previniendo su ejecución cuando vale `LowWater`:

$$ReqPre(\texttt{switchPumpOn}) = \neg \texttt{LowWater}$$

Luego de este refinamiento, la misma violación para TP no volverá a ocurrir en el sistema. Este proceso de operacionalización de la propiedad de progreso TP es iterada, hasta que la fase de Model Checking no reporta más errores. Luego de satisfacer TP, procedemos a operacionalizar los objetivos de safety. Más adelante veremos por qué es conveniente operacionalizar primero TP y luego los objetivos de safety, para acercarnos a la minimalidad de la solución.

4.5.2 *Liveness: Propiedades de Reactividad*

Las propiedades de *liveness* han sido usadas extensivamente en sistemas reactivos, por ejemplo, en el trabajo de Manna y Pnueli [Manna and Pnueli, 1992]. En el contexto de los métodos orientados a objetivos, las propiedades de liveness son típicamente restringidas a *liveness acotado*. Por ejemplo, Letier argumenta

en [Letier, 2001] que las propiedades de respuesta como $\Box(trigger \rightarrow \Diamond response)$ (donde *response* es controlado por el software y *trigger* es monitoreado por el software) deben ser acotadas, debido a que de otra manera el agente responsable de este objetivo puede posponer su obligación indefinidamente, sin obtener una violación finita observable. Sin embargo, a veces para este tipos de propiedades es conveniente *abstraer* la cota (porque es *desconocida*, o porque si es demasiado grande, puede hacer que el espacio de estados *explote* ya que las unidades de tiempo deben ser explicitamente contadas en el modelo). Cuando las propiedades están muy lejos en la intersección de la interfaz "mundo/maquina" o abarca muchas interacciones entre en mundo y la máquina (piense en el patrón de *respuesta encadenada*), propiedades como $(\Box\Diamond As \rightarrow \Box\Diamond G)$ permiten abstraernos de los comportamientos acotados que el sistema deberá tener para satisfacer el objetivo. Por lo tanto, a nuestro criterio, es importante lidiar con propiedades de *liveness* en operacionalización de objetivos.

Nuestra técnica considera las propiedades de liveness que pueden ser expresadas con el patrón de *reactividad* $(\Box\Diamond As \rightarrow \Box\Diamond G)$, donde As y G son expresiones sin operadores temporales. Estas propiedades tienen dos partes: el antecedente o asunciones (As) y el consecuente u objetivos (G). Intuitivamente, podríamos pensar que esta propiedad nos dice: "Si el ambiente garantiza infinitas veces las asunciones As, entonces el software deberá garantizar infinitas veces los objetivos G". Este patrón es suficientemente general para abarcar varias propiedades de liveness [Manna and Pnueli, 1992]. El software que deba garantizar un requisito de este estilo, puede ignorar finitas veces a As, pero de ninguna manera lo puede hacer infinitamente. Manna y Pnueli expresan que esta descripción no puede ser tomada literalmente, ya que ninguna implementación podría esperar a ver si As ocurre un número finito o infinito de veces. Por lo tanto, cualquier implementación *razonable* de este requisito, debe intentar (sinceramente) de responder (G) a cada solicitud de As, a pesar de que la especificación de la propiedad *tolera* que la respuesta (G) falle un número finito de veces.

Una característica importante de las propiedades de reactividad, es que nos brindan información sobre la forma de sus contraejemplos (ver Figura 27), lo que nos dará la oportunidad de usar interpolación para remover este tipo de violaciones.

Model Checking para Reactividad

La fase de Model Checking para este caso, consiste en los siguientes tres puntos. Primero, codificamos el Modelo Operacional O en aserciones LTL como explicamos en la Subsección 4.3.1, y traducimos la propiedad de reactividad a una aserción FLTL asíncrona como explicamos en la Definición 4.3. Segundo, en

general vamos necesitar asumir ciertas *expectativas* sobre el ambiente, es decir, un conjunto de objetivos E que son garantizados por agentes ambientales. Estas expectativas E, también son codificados en aserciones FLTL asíncronas. Por último, le indicaremos a LTSA que no utilice la asunción de *strong fairness*, por las mismas razones que para verificar TP necesitamos.

Veamos un ejemplo. Supongamos que el `MinePumpController` tiene que satisfacer la siguiente propiedad: *"Si infinitas veces no hay metano en el ambiente (así la bomba puede ser prendida), luego infinitas veces el nivel del agua no será alto"*. Esta propiedad puede especificarse como:

```
assert Reactividad = (□◊¬Methane → □◊¬HighWater)
```

La expectativa que asumimos sobre el ambiente, indica que: "Si la bomba se enciende, eventualmente el nivel del agua no va a ser alto". Esta expectativa puede formalizarse como:

```
assert Expectativa = □(PumpOn → ◊¬HighWater)
```

Finalmente, el objetivo de liveness a verificar es el siguiente:

```
assert LivenessGoal = Expectativa ∧ Reactividad
```

Removiendo la asunción de *strong fairness* de LTSA, retorna el siguiente contraejemplo:

```
Violation of LTL property: @LivenessGoal
Trace to terminal set of states:
  tick
  aboveLow
  tick
  aboveHigh
  raiseAlarm
  tick
  signalMethane
Cycle in terminal set:
  tick    HighWater ∧ Methane  (s3)
  signalNoMethane
  tick    HighWater  (s4)
  signalMethane
```

Note que `(s4)` es el estado en el ciclo que satisface la asunción `¬Methane`, pero ni `(s3)` ni `(s4)` satisfacen el goal `¬HighWater`.

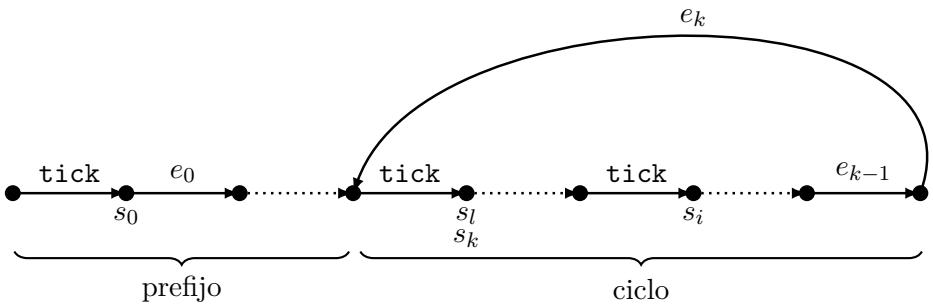

Figura 27: Contraejemplos para propiedades de Reactividad.

Refinamiento para Reactividad

En líneas generales, una violación para $G_{Live} = (\Box\Diamond As \rightarrow \Box\Diamond G)$, consiste de un *prefijo* (parte finita) alcanzando un *ciclo*, tal como muestra la Figura 27. De los estados *observables* del ciclo, al menos uno satisface las asunciones As, pero ninguno satisface los objetivos G. Recuerde que un estado se considera *observable*, si le llega una transición `tick` (ver Definición 4.2). Por ejemplo, del contraejemplo anterior retornado por LTSA, los estados `(s3)` y `(s4)` son estados observables del ciclo; mientras que s_0, s_l, s_i, y s_k son algunos estados observables de la Figura 27. Note que los estados s_l y s_k son los mismos, representando el "inicio" y "final" del ciclo del contraejemplo, respectivamente.

Para que podamos usar interpolación, y así refinar alguna condición requerida que remueva esta violación, vamos a tener que codificar el contraejemplo y la propiedad de reactividad, en fórmulas proposicionales. Veamos como podemos hacer esto.

Anteriormente, en la Subsección 4.4.2 presentamos cómo un contraejemplo para objetivos de safety puede traducirse a una fórmula proposicional (vea por ejemplo, la Figura 25). Básicamente, definimos una relación de transición $R(i, i+1)$ que captura los eventos que se ejecutan entre el $\texttt{tick}_i$ y $\texttt{tick}_{i+1}$, y cómo se actualiza el valor de cada fluente desde el estado s_i al s_{i+1}, dependiendo de los eventos ejecutados. Supongamos que T es la traza del contraejemplo de reactividad retornado por LTSA:

$$T = \texttt{tick}_0; e_0; \ldots; \{\texttt{tick}_l; \ldots; \texttt{tick}_i; \ldots; e_{k-1}; \texttt{tick}_{k-1}; \ldots; e_k\}$$

donde entre `{}` se encuentran los eventos que forman parte del ciclo.

La codificación de T en una fórmula proposicional, es muy similar que la utilizada para contraejemplos de safety, pero para este caso, la última transición

va a ir desde s_{k-1} a s_l (en vez de s_k). Luego, la fórmula proposicional F_T que caracteriza a T es la siguiente:

$$F_T = \bigwedge_{i=0}^{l-1} R(i, i+1) \wedge \bigwedge_{i=l}^{k-2} R(i, i+1) \wedge R(k-1, l)$$

Para el contraejemplo anterior, retornado por LTSA, construimos la fórmula $F_T = R(0,1) \wedge R(1,2) \wedge R(2,3) \wedge R(3,4) \wedge R(4,3)$. Puede encontrar una versión simplificada de esta especificación en la Figura 28.

Para computar un interpolante para este contraejemplo, vamos a codificar el objetivo de reactividad en la siguiente fórmula proposicional:

$$F_{G_{Live}} = (\bigvee_{i=l}^{k-1} As_i) \rightarrow (\bigvee_{i=l}^{k-1} G_i)$$

donde las expresiones As_i/G_i significan que las asunciones/objetivos valen en el estado s_i. Por ejemplo, el objetivo `LivenessGoal` para el `MinePumpController` es especificado como sigue:

```
(
# Extectation:
(PumpOn3 -> (!HighWater3 | !HighWater4))
& (PumpOn4 -> (!HighWater3 | !HighWater4))
#Assumption:
& (!Methane3 | !Methane4)
)
->
#Goal:
(!HighWater3 | !HighWater4)
```

Sea F_T la fórmula que caracteriza la traza del contraejemplo. Claramente, $F_T \wedge F_{G_{Live}}$ es insatisfacible. Luego, podemos computar un interpolante I para estas fórmulas. Este interpolante es una representación *más débil del ciclo* que explica cuál es el problema dentro del ciclo. Por ejemplo, continuando con el `MinePumpController`, el interpolante computado es:

```
   (Methane3 ∧ HighWater3 ∧ ¬PumpOn3)
∧ (¬Methane4 ∧ HighWater4 ∧ ¬PumpOn4)
```

Decimos que el interpolante es una representación más debil del ciclo, ya que sólo menciona las variables relevantes para la propiedad (por ejemplo, no menciona `LowWater` ni `Alarm`).

Para remover este contraejemplo, nuestra técnica busca una operación a_t que cumpla con las siguientes propiedades:

```
VAR
#variables
LowWater0,...,LowWater4: BOOLEAN; HighWater0,...,HighWater4:BOOLEAN;
Methane0,...,Methane4: BOOLEAN;
PumpOn0,...,PumpOn4: BOOLEAN;    Alarm0,...,Alarm4: BOOLEAN;
#operaciones
signalMethane0,signalNoMethane0: BOOLEAN
aboveHigh0, belowHigh0, belowLow0, aboveLow0: BOOLEAN
switchPumpOn0, switchPumpOff0, raiseAlarm0, stopAlarm0: BOOLEAN
....
signalMethane4,signalNoMethane4: BOOLEAN
aboveHigh4, belowHigh4, belowLow4, aboveLow4: BOOLEAN
switchPumpOn4, switchPumpOff4, raiseAlarm4, stopAlarm4: BOOLEAN

FORMULA
#I(0)
LowWater0 & !HighWater0 & !Methane0 & !PumpOn0 & !Alarm0
# R(0,1)
# R(1,2)
# R(2,3)
# R(3,4)
# R(4,3)
& ((aboveHigh4 & HighWater3) | (belowHigh4 & !HighWater3)
  | (!aboveHigh4 & !belowHigh4 & HighWater3=HighWater4))
& ((belowLow4 & LowWater3) | (aboveLow4 & !LowWater3)
  | (!belowLow4 & !aboveLow4 & LowWater3=LowWater4))
& ((signalMethane4 & Methane3) | (signalNoMethane4 & !Methane3)
  | (!signalMethane4 & !signalNoMethane4 & Methane3=Methane4))
& ((switchPumpOn4 & PumpOn3) | (switchPumpOff4 & !PumpOn3)
  | (!switchPumpOn4 &!switchPumpOff4 & PumpOn3=PumpOn4))
& ((raiseAlarm4 & Alarm3) | (stopAlarm4 & !Alarm3)
  | (!raiseAlarm4 & !stopAlarm4 & Alarm3=Alarm4))

#contraejemplo
.....
& !signalMethane3 & signalNoMethane3
& !belowLow3 & !aboveLow3 & !aboveHigh3 & !belowHigh3
& !switchPumpOn3 & !switchPumpOff3 & !raiseAlarm3 & !stopAlarm3

& signalMethane4 & !signalNoMethane4
& !belowLow4 & !aboveLow4 & !aboveHigh4 & !belowHigh4
& !switchPumpOn4 & !switchPumpOff4 & !raiseAlarm4 & !stopAlarm4
```

Figura 28: Especificación MSAT del `MinePumpController` para Reactividad.

- a_t debe poder ser ejecutada en algún estado observable del ciclo:

$$s_i \Rightarrow DomPre(a_t)_i \wedge ReqPre(a_t)_i \tag{9}$$

- y la ejecución de a_t debe alcanzar un estado en donde no valga el interpolante, es decir, su ejecución "rompe el ciclo":

$$(s_i \wedge DomPost(a_t)_{suc(i)}) \Rightarrow \neg I \tag{10}$$

  donde $suc(i) = i + 1$ si $i < k - 1$, sino $suc(i) = l$ cuando $i = k - 1$.

Si encontramos una operación que satisface ambas condiciones (9) y (10), entonces refinamos la condición de triggering requerida de a_t con la conjunción de su precondición requerida y la negación del objetivo G:

$$ReqTrig(a_t) = ReqTrig^{pre}(a_t) \vee (ReqPre(a_t) \wedge \neg G)$$

Note que no refinamos una nueva condición de triggering en base al interpolante. Peor aún, podemos producir una condición de triggering *más débil* de lo necesario. Igualmente, podemos garantizar que la técnica es correcta y consistente con respecto a las precondiciones, ya que son utilizadas en el refinamiento.

Suponiendo que $ReqPre(\texttt{switchPumpOn}) = \neg\texttt{LowWater} \wedge \neg\texttt{Methane}$ (basados en refinamientos previos, al operacionalizar objetivos de safety). Luego, la operación `switchPumpOn` puede ejecutarse en el estado `(s4)`, y cumple con estas dos condiciones:

$$(\neg\texttt{Methane4} \wedge \texttt{HighWater4} \wedge \neg\texttt{PumpOn4}) \rightarrow (\neg\texttt{PumpOn4} \wedge \neg\texttt{LowWater4} \wedge \neg\texttt{Methane4})$$

$$(\neg\texttt{Methane4} \wedge \texttt{HighWater4} \wedge \neg\texttt{PumpOn4}) \wedge$$
$$(\texttt{PumpOn3} = True \wedge \texttt{Methane3=Methane4} \wedge \texttt{HighWater3=HighWater4})$$
$$\rightarrow \neg((\texttt{Methane3} \wedge \texttt{HighWater3} \wedge \neg\texttt{PumpOn3}) \wedge (\neg\texttt{Methane4} \wedge \texttt{HighWater4} \wedge \neg\texttt{PumpOn4}))$$

Luego, debemos debilitar la condición de triggering requerida de `switchPumpOn` con la conjunción de su precondición requerida y la negación del objetivo de la siguiente manera:

$$ReqTrig(\texttt{switchPumpOn}) = \neg\texttt{Methane} \wedge \neg\texttt{LowWater} \wedge \texttt{HighWater}$$

4.6 HACIA UNA METODOLOGÍA

A lo largo de este capítulo hemos demostrado que nuestra técnica produce una operacionalización correcta y consistente, y que garantiza la satisfacción de

objetivos, tanto de safety, como de liveness. También mostramos que nuestra técnica es incompleta. Es decir, puede ocurrir que no encuentre una operacionalización correcta para los objetivos, siendo que existe una.

Como mencionamos en la Sección 4.3, una propiedad deseable de una operacionalización, es la *minimalidad*: las condiciones requeridas no deberían restringir más de lo necesario el comportamiento del software, para satisfacer los objetivos. Sin embargo, en la sección anterior vimos que para el caso de las propiedades de reactividad, las condiciones de triggering requeridas que refina nuestra técnica pueden ser más débiles de lo necesario, obligando a una operación a ejecutarse en más estados de los requeridos para garantizar la propiedad. Luego, nuestra técnica no puede garantizar que la operacionalización computada cumple con *minimalidad*.

Sin embargo, podemos brindar una metodología que indica cuál es el *orden* más conveniente para operacionalizar los objetivos, para acercarnos lo más posible a una solución minimal. El proceso de refinamiento que sigue nuestra metodología es el siguiente:

1. Comenzamos removiendo las violaciones de *Time Progress*. Como usamos la asunción de Máximo Progreso, el software antes del tick intentará ejecutar toda operación controlada posible. Para prevenir la ejecución de alguna de las operaciones que nos llevan a la violación de TP, en general, nuestra técnica produce precondiciones requeridas. Las precondiciones enriquecen la especificación, y ayudarán a que las condiciones de triggering refinadas (en las siguientes iteraciones) sean consistentes con la especificación.

2. Segundo, operacionalizar los objetivos de *safety*. En general, aquí surgen las condiciones de triggering requeridas para cada operación (y las precondiciones faltantes). Como ya habíamos computado las precondiciones requeridas anteriormente, las condiciones de triggering requeridas computadas en esta parte, tienen más garantías de ser consistentes con la especificación, y evitar futuras inconsistencias.

3. Por último, operacionalizar las propiedades de *liveness*. Como hemos visto, las condiciones de triggering requeridas computadas para garantizar propiedades de reactividad, utilizan las precondiciones de las operaciones. Por lo tanto, es importante realizar este paso al final, ya que tenemos más posibilidades de no generar condiciones de triggering demasiado débiles, forzando a una operación a ejecutarse en más estados de los necesarios.

Es importante mencionar que esta metodología, que extiende a la provista en [Alrajeh et al., 2013], no es más que una *heurística* que surgió desde la observación de los casos de estudio.

4.7 CASOS DE ESTUDIOS

En esta sección reportamos los resultados experimentales de aplicar nuestra técnica a dos modelos: el *Mine Pump Controller* [Kramer et al., 1983] y el Engineered Safety Feature Actuation System (ESFAS) [Letier, 2002]. Para el caso de operacionalización de objetivos de safety y la propiedad Time Progress, nuestro enfoque es comparado con la técnica basada en Programación Lógica Inductiva (la denotaremos como *ILP-framework*) introducido en [Alrajeh et al., 2009], con propósitos de evaluación y validación. Los escenarios positivos que necesita ILP-framework fueron producidos manualmente por nosotros (en particular, por Alrajeh una de los autores de [Alrajeh et al., 2009; Degiovanni et al., 2014]), siguiendo las guías provistas en [Alrajeh et al., 2013]. Sin embargo, estas intervenciones humanas, no fueron requeridas por nuestra técnica ya que es completamente automática. Por otro lado, los enfoque previos de operacionalización de objetivos, no lidian con propiedades de liveness, por lo que no tenemos resultados previos contra que compararnos para validar nuestra técnica. Discutiremos sobre este asunto más tarde.

La evaluación experimental se llevará a cabo siguiendo la metodología presentada anteriormente en la Sección 4.6. Primero operacionalizaremos la propiedad de progreso TP y luego los objetivos de safety. Cada modelo es acompañado por una descripción informal del sistema, una especificación del Modelo Operacional inicial, y un conjunto de objetivos especificados en FLTL. Los experimentos aquí presentados pueden descargarse desde `http://dc.exa.unrc.edu.ar/staff/rdegiovanni/icse2014.zip` y reproducirse siguiendo las instrucciones que allí puede encontrar.

4.7.1 *Mine Pump Controller*

Este modelo fue utilizado como ejemplo motivador a lo largo de este capítulo. Puede encontrar los detalles en la Sección 4.2. Recuerde que este modelo tiene tres variable ambientales (`Methane`, `LowWater` y `HighWater`), y dos variables controladas (`PumpOn` y `Alarm`). Las variables monitoreadas son modificadas por eventos ambientales: `belowLow`, `aboveLow`, `aboveHigh` y `belowHigh` modifican el nivel del agua en la mina, mientras que `signalMethane` y `signalNoMethane` ocurren acorde a la presencia de niveles críticos de metano en el ambiente. El sistema puede activamente controlar el estado de la bomba y la alarma con los eventos `switchPumpOn`, `switchPumpOff`, `raiseAlarm` y `stopAlarm`. La Figura 22 describe el Modelo Operacional inicial para el `MinePumpController`. Inicialmente, las operaciones están habilitadas a ejecutarse mientras lo permita

la precondición de dominio (es decir, la precondición requerida es $True$), y no estarán obligadas a ejecutarse en ningún estado (es decir, la condición de triggering requerida es $False$).

Los objetivos que debe operacionalizar el `MinePumpController`, de safety y progreso, son los siguientes:

```
progress TP = {tick}

assert PumpOnWhenHighWaterANDNoMethane =
  [] (tick -> ((HighWater && !Methane) ->X (! tick W (tick && PumpOn))))

assert PumpOffWhenLowWater =
  [] (tick -> ((LowWater) -> X (! tick W (tick && !PumpOn))))

assert PumpOffWhenMethane =
  [] (tick -> ((Methane) -> X (! tick W (tick && !PumpOn))))

assert AlarmWhenMethane =
  [] (tick -> ((Methane) -> X (! tick W (tick && Alarm))))

assert Goals = (PumpOnWhenHighWaterANDNoMethane && PumpOffWhenLowWater
       && PumpOffWhenMethane && AlarmWhenMethane).
```

Siguiendo nuestra metodología propuesta de la Sección 4.6, vamos a comenzar removiendo las violaciones de Time Progress, para luego operacionalizar los objetivos de safety.

Elaboración de Requisitos para Time Progress

ITERACIÓN 1. Llamemos O_0 al Modelo Operacional inicial. Considerando las asunciones necesarias, podemos chequear el progreso del evento `tick` sobre O_0, para la cual LTSA retorna el siguiente contraejemplo:

```
Progress violation: TP
Trace to terminal set of states:
  tick
  switchPumpOn
  raiseAlarm
  aboveLow
  signalMethane
Cycle in terminal set:
Actions in terminal set:
  {}
```

```
Progress Check in: 0ms
```

Extendiendo este contraejemplo con el evento `tick` al final, el interpolante computado es: `LowWater0` $\wedge$ `PumpOn1`. Nuestra técnica computa weakest precondition respecto de las operaciones ejecutadas en el contraejemplo. En particular, como $WP(\texttt{LowWater0} \wedge \texttt{PumpOn1}, \texttt{switchPumpOn}) = \texttt{LowWater0}$, la técnica detecta que la ejecución de `switchPumpOn` nos llevaría a violar los objetivos, si `tick` se ejecutara al final del contraejemplo. Por lo tanto, para remover esta violación a TP, nuestra técnica realiza el siguiente refinamiento, y continua a la siguiente iteración:

$$ReqPre_1(\texttt{switchPumpOn}) = \neg\texttt{LowWater} \qquad \text{(MP1)}$$

ITERACIÓN 2. Le agregamos esta precondición requerida al Modelo Operacional inicial, obteniendo $O_1 = O_0 \cup \{ReqPre_1(\texttt{switchPumpOn})\}$. Luego, al chequear TP sobre O_1, nuevamente LTSA retorna un contraejemplo:

```
Progress violation: TP
Trace to terminal set of states:
  tick
  raiseAlarm
  signalMethane
  tick
  stopAlarm
  aboveLow
  signalNoMethane
Cycle in terminal set:
Actions in terminal set:
  {}
Progress Check in: 0ms
```

Al extender el contraejemplo con el evento `tick`, el interpolante computado es: `Methane1` $\wedge \neg$`Alarm2`. Este interpolante nos indica que la alarma no debería apagarse, ya que hay metano en el ambiente (y así no violar el objetivo `AlarmWhenMethane`). Para remover esta violación a TP, nuestra técnica computa la siguiente precondición requerida:

$$ReqPre(\texttt{stopAlarm}) = \neg\texttt{Methane} \qquad \text{(MP2)}$$

ITERACIÓN 3. Luego del refinamiento anterior, obtenemos $O_2 = O_1 \cup \{ReqPre(\texttt{stopAlarm})\}$. Nuevamente, utilizamos LTSA para verificar la validez de TP sobre O_2, y obtenemos el siguiente contraejemplo:

```
Progress violation: TP
Trace to terminal set of states:
  tick
  raiseAlarm
  aboveLow
  signalMethane
  tick
  switchPumpOn
  aboveHigh
  signalNoMethane
Cycle in terminal set:
Actions in terminal set:
  {}
Progress Check in: 0ms
```

Note que si `tick` ocurriera, violaríamos el objetivo `PumpOffWhenMethane`. Para este caso, el interpolante computado es: $\texttt{Methane1} \wedge \texttt{PumpOn2}$. Luego, el refinamiento que nuestra técnica realiza para remover esta violación a TP, es la siguiente:

$$ReqPre_2(\texttt{switchPumpOn}) = \neg\texttt{Methane} \tag{MP3}$$

ITERACIÓN 4. Considere ahora $O_3 = O_2 \cup \{ReqPre_2(\texttt{switchPumpOn})\}$. Luego LTSA reporta el siguiente contraejemplo cuando analiza TP sobre O_3:

```
Progress violation: TP
Trace to terminal set of states:
  tick
  raiseAlarm
  aboveLow
  tick
  switchPumpOn
  stopAlarm
  aboveHigh
  tick
  switchPumpOff
```

```
  raiseAlarm
  belowHigh
  signalMethane
Cycle in terminal set:
Actions in terminal set:
  {}
Progress Check in: 0ms
```

El inteprolante en este caso es $(\neg\texttt{Methane2} \wedge \texttt{HighWater2} \wedge \neg\texttt{PumpOn3})$, mostrando que si `tick` fuera ejecutado, el objetivo `PumpOnWhenHighWaterANDNoMethane` sería violado. En base a este contraejemplo e interpolante, la técnica refina una precondición para `switchPumpOff`:

$$\mathit{ReqPre}(\texttt{switchPumpOff}) = \neg(\neg\texttt{Methane} \wedge \texttt{HighWater}) \qquad \text{(MP4)}$$

ITERACIÓN 5. Luego de agregar la condición requerida anterior, obtenemos $O_4 = O_3 \cup \{\mathit{ReqPre}(\texttt{switchPumpOff})\}$. Al analizar si O_4 satisface la propiedad de progresso TP, LTSA retorna la siguiente respuesta:

```
No progress violations detected.
Progress Check in: 7ms
```

Esto indica que el Modelo Operacional O_4 garantiza el progreso del evento `tick`, por lo tanto, satisface TP.

Elaboración de Requisitos para Objetivos de Safety

ITERACIÓN 6. Hasta aquí, el Modelo Operacional O_4 operacionaliza correctamente la propiedad TP. Ahora, nuestra técnica comenzará el proceso de refinamiento para que satisfaga los objetivos de safety. Al verificar la validez de los objetivos sobre O_4, LTSA retorna el siguiente contraejemplo:

```
Trace to property violation in Goals:
  tick    LowWater
  signalMethane   Methane && LowWater
  tick    Methane && LowWater
  tick    Methane && LowWater
Analysed in: 0ms
```

Note, que este contraejemplo se corresponde a una violación al objetivo `AlarmWhenMethane`. Al computar interpolante para el contraejemplo y los objetivos,

obtenemos: `Methane1` $\wedge \neg$ `Alarm2`. Como ninguna operación se ejecuta entre los últimos dos ticks, nuestra técnica busca disparar una operación cuya ejecución evite alcanzar el interpolante. Dicha operación es `raiseAlarm`, por lo que la técnica refina la siguiente condición de triggering requerida:

$$\mathit{ReqTrig}(\texttt{raiseAlarm}) = \neg\texttt{Alarm} \wedge \texttt{Methane} \qquad \text{(MP5)}$$

ITERACIÓN 7. $O_5 = O_4 \cup \{\mathit{ReqTrig}(\texttt{raiseAlarm})\}$ se obtiene agregando la condición de triggering recién computada. Usamos LSTA para verificar si O_5 satisface los objetivos, y obtenemos el siguiente contraejemplo:

```
Trace to property violation in Goals:
  tick    LowWater
  aboveLow
  tick
  aboveHigh   HighWater
  tick    HighWater
  tick    HighWater
Analysed in: 1ms
```

El objetivo particular violado en este caso, es `PumpOnWhenHighWaterAND-NoMethane`. El proceso de interpolación arroja el interpolante: ($\neg$`Methane2` $\wedge$ `HighWater2` $\wedge \neg$`PumpOn3`). Nuevamente, como ninguna operación es ejecutada entre los últimos ticks, nuestra técnica busca forzar una operación a ocurrir, para evitar alcanzar el interpolante. Para esto, refina una condición de triggering requerida para `switchPumpOn`, que forzará su ejecución cuando:

$$\mathit{ReqTrig}(\texttt{switchPumpOn}) = \neg\texttt{PumpOn} \wedge \neg\texttt{Methane} \wedge \texttt{HighWater} \qquad \text{(MP6)}$$

ITERACIÓN 8. A partir de la condición de triggering anterior, obtenemos $O_6 = O_5 \cup \{\mathit{ReqTrig}(\texttt{switchPumpOn})\}$. Para el Modelo Operacional O_6, LTSA reporta la siguiente violación a los objetivos:

```
Trace to property violation in Goals:
  tick    LowWater
  aboveLow
  tick
  belowLow    LowWater
  switchPumpOn    PumpOn && LowWater
  tick    PumpOn && LowWater
  tick    PumpOn && LowWater
Analysed in: 1ms
```

Note que el objetivo violado es `PumpOffWhenLowWater`. Cuando computamos un interpolante, obtenemos: `LowWater2` $\wedge$ `PumpOn3`. Nuevamente, nuestra técnica refina una condición de triggering para una operación para remover este contraejemplo. En este caso:

$$ReqTrig_1(\texttt{switchPumpOff}) = \texttt{PumpOn} \wedge \texttt{LowWater} \qquad \text{(MP7)}$$

ITERACIÓN 9. La anterior condición requerida es agregada al Modelo Operacional $O_7 = O_6 \cup \{ReqTrig_1(\texttt{switchPumpOff})\}$. Para este modelo, LTSA reporta el contraejemplo:

```
Trace to property violation in Goals:
  tick    LowWater
  aboveLow
  tick
  signalMethane   Methane
  switchPumpOn    Methane && PumpOn
  tick    Methane && PumpOn
  raiseAlarm    Methane && PumpOn && Alarm
  tick    Methane && PumpOn && Alarm
Analysed in: 6ms
```

Note que el objetivo que se viola en este caso es `PumpOffWhenMethane`. Por lo tanto, el interpolante computado expresa que: `Methane2` $\wedge$ `PumpOn3`. Si computamos $WP(\texttt{Methane2} \wedge \texttt{PumpOn3}, \texttt{raiseAlarm}) = \texttt{Methane2} \wedge \texttt{PumpOn2}$. Note que la operación `raiseAlarm` está obligada a ejecutarse cuando `Methane`, por lo tanto no podemos prohibir su ejecución. Luego, nuestra técnica busca una operación que pueda disparar cuando valga `Methane2` $\wedge$ `PumpOn2`, de forma tal, que su ejecución no alcance el interpolante. Dicha operación es `switchPumpOff`, y la técnica refina la siguiente condición de triggering:

$$ReqTrig_2(\texttt{switchPumpOff}) = \texttt{PumpOn} \wedge \texttt{Methane} \qquad \text{(MP8)}$$

ITERACIÓN 10. Finalmente, a partir del refinamiento anterior, obtenemos $O_8 = O_7 \cup \{ReqTrig_2(\texttt{switchPumpOff})\}$. Al verificar si O_8 satisface los objetivos, LTSA retorna:

```
No deadlocks/errors
Analysed in: 2ms
```

Como nuestra técnica satisface los objetivos, el proceso de operacionalización finaliza correctamente luego de 10 iteraciones, produciendo 8 condiciones requeridas para las operaciones (en las iteraciones 5 y 10 no refina ninguna condición).

$$ReqPre(\texttt{switchPumpOn}) = \neg\texttt{LowWater} \quad \text{(MP1)}$$
$$ReqPre(\texttt{stopAlarm}) = \neg\texttt{Methane} \quad \text{(MP2)}$$
$$ReqPre(\texttt{switchPumpOn}) = \neg\texttt{Methane} \quad \text{(MP3)}$$
$$ReqPre(\texttt{switchPumpOff}) = \neg(\neg\texttt{Methane} \wedge \texttt{HighWater}) \quad \text{(MP4)}$$
$$ReqTrig(\texttt{raiseAlarm}) = \neg\texttt{Alarm} \wedge \texttt{Methane} \quad \text{(MP5)}$$
$$ReqTrig(\texttt{switchPumpOn}) = \neg\texttt{PumpOn} \wedge \neg\texttt{Methane} \wedge \texttt{HighWater} \quad \text{(MP6)}$$
$$ReqTrig(\texttt{switchPumpOff}) = \texttt{PumpOn} \wedge \texttt{LowWater} \quad \text{(MP7)}$$
$$ReqTrig(\texttt{switchPumpOff}) = \texttt{PumpOn} \wedge \texttt{Methane} \quad \text{(MP8)}$$

Figura 29: Operacionalización Correcta para el `MinePumpController`.

La Figura 29 resume las iteraciones realizadas por el proceso de refinamiento. Las precondiciones requeridas (MP1)-(MP4) remueven las violaciones a Time Progress, y las condiciones de triggering requeridas (MP5)-(MP8) garantizan la satisfacción de los objetivos de safety.

Comparación con ILP-framework

Cuando lo comparamos con los requerimientos aprendidos por ILP-framework en [Alrajeh et al., 2009], para este modelo, observamos que ambas técnicas iteran exactamente el mismo número de veces (10 iteraciones). Ambos enfoques producen las mismas condiciones requeridas, excepto por la condición de triggering requerida (MP6). ILP-framework aprende una condición de triggering más débil para `switchPumpOn`: `HighWater`. Debido a este problema de sobre-generalización de ILP, la condición aprendida introduce un deadlock en el sistema:

```
Trace to DEADLOCK:
  tick (s0)
  aboveLow
  tick (s1)
  aboveHigh
  signalMethane
  tick (s2)
  belowHigh
  signalNoMethane
```

El deadlock es producido porque en el estado (s2), ambos HighWater y Methane son verdaderos. Luego, la precondición requerida (MP3) indica que switchPumpOn no puede ejecutarse cuando Methane, pero la condición de triggering aprendida obliga a switchPumpOn a ocurrir cuando HighWater. En nuestro caso, la condición de triggering requerida refinada en (MP6) es más fuerte, requiriendo que la presencia de metano sea falsa cuando el nivel del agua sea alto. Por otro lado, para remover el deadlock, ILP-framework es forzado a volver hacia iteraciones previas y requiere al ingeniero que introduzca más escenarios, o produzca manualmente el refinamiento.

Con respecto a los tiempos de ejecución, es importante mencionar que, para este modelo, nuestra técnica es significativamente más eficiente. Mientras ILP-framework requiere aproximadamente 29 segundos por iteración, nuestra técnica toma menos de 1 segundo por iteración.

4.7.2 *Engineered Safety Feature Actuation System (ESFAS)*

El Engineered Safety Feature Actuation System (ESFAS) fue originalmente introducido por Courtis y Parnas en [Courtois and Parnas, 1993] (en el Capítulo 5 usaremos este mismo ejemplo para presentar nuestra técnica de análisis de especificaciones SCR). Más tarde, Letier reportó una especificación KAOS para este mismo modelo en [Letier, 2002], en el cuál muestra por completo el refinamiento desde objetivos de alto nivel, a los de bajo nivel, y como podrían ser operacionalizados.

El sistema ESFAS es encargado de mantener la presión de agua de un reactor nuclear en niveles aceptables. Para esto, ESFAS monitorea tres variables ambientales: la presión del agua mediante PressureBelowLow (por debajo de una constante 'Low') y PressureAbovePermit (por encima de una constante 'Permit'); y un par de interruptores para bloquear o resetear el sistema (Occurs_block y Occurs_reset, respectivamente). Además, ESFAS puede controlar dos variables: Overridden indica si se ha "bloqueado" el sistema de inyección segura, y SafetyInjection indica cuando el sistema de inyección segura esta encendido. Básicamente, el sistema debe comenzar la inyección segura cuando la presión se vuelve demasiado baja.

Las variables monitoreadas son controladas por los siguientes eventos ambientales: lowerPressureBelowLow y raisePressureAboveLow modifican PressureBelowLow; los eventos raisePressureAbovePermit y lowerPressureBelowPermit modifican PressureAbovePermit; block y tock modifican Occurs_block; y reset y tock modifican Occurs_reset. Las variables controladas son modificadas por las siguientes operaciones del software: Overridden es encendido por

```
Operation overrideSignal
  DomPre  ¬Overridden
  DomPost Overridden
  ReqPre  True
  ReqTrig False

Operation enableSignal
  DomPre  Overridden
  DomPost ¬Overridden
  ReqPre  True
  ReqTrig False

Operation sendSignal
  DomPre  ¬SafetyInjection
  DomPost SafetyInjection
  ReqPre  True
  ReqTrig False

Operation stopSignal
  DomPre  SafetyInjection
  DomPost ¬SafetyInjection
  ReqPre  True
  ReqTrig False
```

Figura 30: Modelo Operacional inicial para el ESFAS.

overrideSignal y apagado por enableSignal; mientras que SafetyInjection se enciende con sendSignal y se apaga con stopSignal. Para modelar esto, consideramos las siguientes definiciones de fluentes:

```
fluent Overridden = <overrideSignal, enableSignal, True>
fluent SafetyInjection = <sendSignal,stopSignal >
fluent PressureBelowLow = <lowerPressureBelowLow,
                           raisePressureAboveLow, True>
fluent PressureAbovePermit = <raisePressureAbovePermit,
                              lowerPressureBelowPermit>
fluent Occurs_block = <block,tock >
fluent Occurs_reset = <reset,tock >
```

El evento tock es ejecutado exactamente después de tick. Este evento auxiliar es introducido como la acción de terminación de las acciones realizadas por el operador de presionar los botones de reseteo o bloqueo.

La Figura 30 muestra el Modelo Operacional para el ESFAS. Inicialmente asumimos a *True* y *False* como la precondición requerida y la condición de triggering requerida, respectivamente, de cada operación controlada. Además, consideramos dos asunciones ambientales para indicar que el operador no puede presionar el botón de reseteo y bloqueo al mismo tiempo:

$$Assumption\_1 = \Box(\texttt{tick} \wedge \texttt{Occurs\_block} \rightarrow \neg\texttt{Occurs\_reset})$$
$$Assumption\_2 = \Box(\texttt{tick} \wedge \texttt{Occurs\_reset} \rightarrow \neg\texttt{Occurs\_block})$$

A continuación presentamos los objetivos de safety que deben ser garantizados por ESFAS, formalizados en FLTL como:

```
assert SafetyInjectionWhenLowWaterPressureAndNotOverridden =
    □ (tick → ( (PressureBelowLow ∧ ¬Overridden)
                    → ◯(¬tick 𝒲 (tick ∧ SafetyInjection))))
assert SIEnabledWhenPressureAbovePermitOrManualReset =
    □ (tick → ( (Occurs_reset ∨ PressureAbovePermit)
                    → ◯(¬tick 𝒲 (tick ∧ ¬Overridden))))
assert SIOverriddenWhenBlockSwOnAndPressureLessThanPermit =
    □ (tick → ( (Occurs_block ∧ ¬PressureAbovePermit)
                    → ◯(¬tick 𝒲 (tick ∧ Overridden))))
```

Intuitivamente, el primer objetivo indica que la señal de inyección segura debe encenderse cuando la presión de agua es menor a 'Low' y el sistema de inyección segura no fue bloqueado (este es el principal objetivo de ESFAS). El segundo objetivo expresa que la inyección segura se debe habilitar cuando el nivel de presión de agua se incrementa sobre 'Permit' o cuando el botón de reseteo es presionado. El tercer objetivo establece que la inyección segura debe bloquearse cuando el interruptor de bloqueo es presionado y el nivel de presión de agua es menor a 'Permit'. La Figura 31 muestra una versión simplificada de la especificación en FSP del Modelo Operacional inicial del `ESFAS`.

Elaboración de Requisitos para Time Progress

Siguiendo nuestra metodología propuesta de la Sección 4.6, vamos a comenzar removiendo las violaciones de Time Progress, para luego operacionalizar los objetivos de safety.

ITERACIÓN 1. Llamemos O_0 al Modelo Operacional inicial de la Figura 30. Considerando las asunciones necesarias para analizar TP sobre O_0, LTSA retorna el siguiente contraejemplo:

```
Progress violation: TP
Trace to terminal set of states:
  tick
  tock
  enableSignal
  sendSignal
  tick
  tock
  overrideSignal
  stopSignal
  block
```

```
// Variables del Modelo Operacional
fluent Overridden = <overrideSignal, enableSignal> initially 1
fluent SafetyInjection = <sendSignal, stopSignal>
fluent PressureBelowLow = <lowerPressureBelowLow,
                           raisePressureAboveLow> initially 1
fluent PressureAbovePermit = <raisePressureAbovePermit,
                              lowerPressureBelowPermit>
fluent Occurs_block = <block, tock>
fluent Occurs_reset = <reset, tock>
//Modelado del Ambiente
PressureLevel = (tick ->tock-> P1 ),
  P1 = (tick -> tock-> P1|
        raisePressureAboveLow ->tick -> tock ->  P2),
  P2  = (tick -> tock-> P2|
        lowerPressureBelowLow ->tick -> tock -> P1
        | raisePressureAbovePermit-> tick ->tock ->P3),
  P3 = (tick->tock-> P3
        | lowerPressureBelowPermit ->tick ->tock -> P2).
Operator = (tick ->S0),
  S0 = (tick -> S0 | {block, reset} ->tick -> S0).
||Environment = ( PressureLevel || Operator).

// Condiciones de DOMINIO
constraint DomPre_overrideSignal =
  [](tick && ! ! Overridden) ->X(! overrideSignal W tick))
constraint DomPre_enableSignal =
  [](tick && ! Overridden) ->X(! enableSignal W tick))
constraint DomPre_sendSignal =
  [](tick && ! ! SafetyInjection) ->X(! sendSignal W tick))
constraint DomPre_stopSignal =
  [](tick && ! SafetyInjection) ->X(! stopSignal W tick))
||Controller = ( DomPre_overrideSignal || DomPre_enableSignal
             || DomPre_sendSignal || DomPre_stopSignal).
StartWithTick = (tick ->tock -> S0 ),
  S0 = (tick -> tock -> S0 | {AllEvents} ->S0).

||ESFAS = (StartWithTick || Operator || Controller).
// Objetivos a satisfacer por ESFAS
progress TP = {tick}
assert SIOverriddenWhenBlockSwOnAndPressureLessThanPermit =
  [] (tick -> ((Occurs_block && !PressureAbovePermit)
    -> X (! tick W (tick && Overridden))))
assert SIEnabledWhenPressureAbovePermitOrManualReset =
  [] (tick -> ((Occurs_reset || PressureAbovePermit)
    -> X (! tick W (tick && !Overridden))))
assert SafetyInjectionWhenLowWaterPressureAndNotOverridden =
  [] (tick -> (PressureBelowLow && !Overridden
    -> X (! tick W (tick && SafetyInjection))))
```

Figura 31: Especificación FSP para el `MinePumpController`.

```
  raisePressureAboveLow
Cycle in terminal set:
Actions in terminal set:
  {}
Progress Check in: 1ms
```

Si extendemos este contraejemplo con el evento `tick`, podemos computar un interpolante, obteniendo: ($\neg$`SafetyInjection2` $\wedge$ `PressureBelowLow1` $\wedge$ $\neg$`Overridden1`). Note que este interpolante muestra que si `tick` ocurriera, entonces el objetivo `SafetyInjectionWhenLowWaterPressureAndNotOverridden` sería violado. Computando WP, nuestra técnica detecta que debe prohibir la ejecución de `stopSignal` cuando ($\neg$`Overridden` $\wedge$ `PressureBelowLow`) para evitar alcanzar el interpolante. Antes de realizar el refinamiento, le técnica verifica que la nueva precondición requerida no contradiga su condición de triggering requerida (inicialmente $False$):

$$False \wedge \texttt{SafetyInjection} \Rightarrow (\neg\texttt{Overridden} \wedge \texttt{PressureBelowLow})$$

Como la condición anterior se verifica correctamente, la técnica refina la siguiente precondición requerida:

$$\mathit{ReqPre}(\texttt{stopSignal}) = \neg(\neg\texttt{Overridden} \wedge \texttt{PressureBelowLow}) \quad \text{(ESFAS1)}$$

ITERACIÓN 2. $O_1 = O_0 \cup \{\mathit{ReqPre}(\texttt{stopSignal})\}$ se obtiene agregando la anterior precondición. Luego, si chequeamos con LTSA la validez de TP sobre O_1, obtenemos:

```
Progress violation: TP
Trace to terminal set of states:
  tick
  tock
  enableSignal
  sendSignal
  reset
  tick
  tock
  overrideSignal
  block
  raisePressureAboveLow
Cycle in terminal set:
Actions in terminal set:
  {}
Progress Check in: 1ms
```

Podemos computar un interpolante al extender el contraejemplo con `tick`, obteniendo: `Overridden2` $\wedge$ `Occurs_Reset1`. Note que este interpolante se corresponde a una violación del objetivo `SIEnabledWhenPressureAbovePermit-OrManualReset`. Nuestra técnica detecta que debe prevenir la ejecución de `overrideSignal` cuando vale `Occurs_Reset`. Para esto, refina la siguiente precondición requerida:

$$ReqPre_1(\texttt{overrideSignal}) = \neg\texttt{Occurs\_reset} \qquad \text{(ESFAS2)}$$

ITERACIÓN 3. Luego de computar la precondición requerida anterior, obtenemos $O_2 = O_1 \cup \{ReqPre_1(\texttt{overrideSignal})\}$. LTSA reporta la siguiente violación para TP sobre O_2:

```
Progress violation: TP
Trace to terminal set of states:
  tick
  tock
  enableSignal
  sendSignal
  tick
  tock
  overrideSignal
  block
  tick
  tock
  enableSignal
  stopSignal
  block
  raisePressureAboveLow
Cycle in terminal set:
Actions in terminal set:
  {}
Progress Check in: 1ms
```

Luego de extender el contraejemplo con `tick`, computamos el siguiente interpolante: (`Occurs_block2` $\wedge$ $\neg$`PressureAbovePermit2` $\wedge$ $\neg$`Overridden3`). Note que `block` y `raisePressureAboveLow` no son operaciones controladas, y `stopSignal` no es capaz de controlar el valor del interpolante. Sin embargo, la ejecución de `enableSignal` es la responsable de que alcancemos el interpolante. Por lo tanto, nuestra técnica previene su ejecución cuando:

$$ReqPre(\texttt{enableSignal}) = \neg(\neg\texttt{PressureAbovePermit} \wedge \texttt{Occurs\_block}) \qquad \text{(ESFAS3)}$$

ITERACIÓN 4. Al agregar la condición requerida anterior, obtenemos el Modelo Operacional $O_3 = O_2 \cup \{ReqPre(\texttt{enableSignal})\}$. Luego, al analizar TP sobre O_3, LTSA reporta el siguiente contraejemplo:

```
Progress violation: TP
Trace to terminal set of states:
  tick
  tock
  enableSignal
  sendSignal
  raisePressureAboveLow
  reset
  tick
  tock
  stopSignal
  raisePressureAbovePermit
  tick
  tock
  overrideSignal
  sendSignal
  block
  lowerPressureBelowPermit
Cycle in terminal set:
Actions in terminal set:
  {}
Progress Check in: 2ms
```

Agregando `tick` al final de la traza, podemos computar el siguiente interpolante: $\texttt{Overridden3} \wedge \texttt{PressureAbovePermit2}$. La técnica detecta que la operación que puede controlar el valor del interpolante es `overrideSignal`, por lo que previene su ejecución mediante la precondición requerida siguiente:

$$ReqPre_2(\texttt{overrideSignal}) = \neg\texttt{PressureAbovePermit} \qquad \text{(ESFAS4)}$$

ITERACIÓN 5. Luego de obtener $O_4 = O_3 \cup \{ReqPre_2(\texttt{overrideSignal})\}$, analizamos TP con LTSA, obteniendo como resultado:

```
Progress Check...
-- States: 800 Transitions: 2311 Memory used: 29328K
No progress violations detected.
Progress Check in: 8ms
```

Hasta aquí el Modelo Operacional O_4 garantiza el progreso del evento `tick` (TP). Veamos ahora cuáles son los refinamientos necesarios para garantizar los objetivos de safety.

Elaboración de Requisitos para Objetivos de Safety

ITERACIÓN 6. Al verificar la validez de los objetivos de safety sobre O_4, LTSA retorna el siguiente contraejemplo:

```
Trace to property violation in Goals:
  tick    Overridden && PressureBelowLow  (s0)
  tock    Overridden && PressureBelowLow
  enableSignal    PressureBelowLow
  tick    PressureBelowLow  (s1)
  tock    PressureBelowLow
  tick    PressureBelowLow  (s2)
Analysed in: 1ms
```

Este contraejemplo corresponde a un caso en el cual la presión del agua es demasiado bajo y el sistema no está bloqueado en el estado `(s1)`, pero en la siguiente unidad de tiempo `(s2)` el sistema de inyección segura está apagado. Esta traza muestra una violación al objetivo `SafetyInjectionWhenLowWaterPressureAndNotOverridden`. El proceso continúa por computar un interpolante para el contraejemplo y el objetivo violado, obteniendo $\neg$`SafetyInjection2` $\wedge$ `PressureBelowLow1` $\wedge \neg$`Overridden1`. Computamos WP respecto a `tock` llevandonos a un predicado sobre el estado `(s1)`, pero note que no podemos prevenir `tock` de ser ejecutado porque no es controlable. Por lo que nuestra técnica tratará de remover el contraejemplo forzando la ocurrencia de una operación tal que su ejecución evite el interpolante. La operación `sendSignal` satisface las dos condiciones para refinar una condición de triggering requerida:

$$\neg\texttt{SafetyInjection} \wedge \neg\texttt{Overridden} \wedge \texttt{PressureBelowLow} \Rightarrow \neg\texttt{SafetyInjection} \wedge True \quad \text{(A)}$$

$$(\neg\texttt{SafetyInjection} \wedge \texttt{PressureBelowLow} \wedge \neg\texttt{Overridden}) \wedge (\texttt{SafetyInjection}' = True \wedge Unchanged(\texttt{Overridden}, \texttt{PressureBelowLow}, ...)) \Rightarrow \neg(\neg\texttt{SafetyInjection}' \wedge \texttt{PressureBelowLow} \wedge \neg\texttt{Overridden}) \quad \text{(B)}$$

La condición (A) asegura que `sendSignal` puede ser ejecutado cuando vale $\neg$`SafetyInjection` $\wedge \neg$`Overridden` $\wedge$ `PressureBelowLow`, mientras que la condición (B) asegura que la ejecución de `sendSignal` evita el interpolante. Luego,

forzando sendSignal a ocurrir cuando ¬SafetyInjection ∧ ¬Overridden ∧ PressureBelowLow, el contraejemplo es removido. Esta iteración termina refinando para sendSignal la condición de triggering requerida (ESFAS5):

$$ReqTrig(\texttt{sendSignal}) = \neg\texttt{SafetyInjection} \wedge \neg\texttt{Overridden} \wedge \texttt{PressureBelowLow} \quad \text{(ESFAS5)}$$

ITERACIÓN 7. Obtenemos $O_5 = O_4 \cup \{ReqTrig(\texttt{sendSignal})\}$ luego de refinar sendSignal. Para este Modelo Operacional, LTSA reporta la siguiente violación a los objetivos:

```
Trace to property violation in Goals:
  tick    Overridden && PressureBelowLow
  tock    Overridden && PressureBelowLow
  reset   Overridden && Occurs_reset && PressureBelowLow
  tick    Overridden && Occurs_reset && PressureBelowLow
  tock    Overridden && PressureBelowLow
  tick    Overridden && PressureBelowLow
Analysed in: 1ms
```

Note que esta traza se corresponde con una violación al objetivo SIEnabledWhen-PressureAbovePermitOrManualReset. El interpolante computado para este caso es: Overridden2 ∧ Occurs_reset1. Nuevamente, como ninguna operación se ejecuta entre los últimos ticks, nuestra técnica fuerza una operación a ejecutarse, para evitar alcanzar el interpolante.

$$ReqTrig_1(\texttt{enableSignal}) = \texttt{Overridden} \wedge \texttt{Occurs\_reset} \quad \text{(ESFAS6)}$$

ITERACIÓN 8. Obtenemos $O_6 = O_5 \cup \{ReqTrig_1(\texttt{enableSignal})\}$ luego de agregar la condición de triggering requerida para enableSignal. Verificamos si O_6 satisface los objetivos, y LTSA reporta lo siguiente:

```
Trace to property violation in Goals:
  tick    Overridden && PressureBelowLow   (s0)
  tock    Overridden && PressureBelowLow
  enableSignal    PressureBelowLow
  sendSignal    PressureBelowLow && SafetyInjection
  block   Occurs_block && PressureBelowLow && SafetyInjection
  tick    Occurs_block && PressureBelowLow && SafetyInjection (s1)
  tock    PressureBelowLow && SafetyInjection
  tick    PressureBelowLow && SafetyInjection   (s2)
```

Note que en `(s1)` vale `PressureBelowLow` y `Occurs_block`, pero en el siguiente `(s2)`, `Overridden` sigue siendo falso. Esta traza es una violación al objetivo `SIOverriddenWhenBlockSwOnAndPressureLessThanPermit`. Computamos un interpolante para este contraejemplo y objetivo violado, obteniendo: $(\neg\texttt{Overridden2} \wedge \neg\texttt{PressureAbovePermit1} \wedge \texttt{Occurs\_block1})$. Para remover esta violación, nuestra técnica fuerza la ocurrencia de `overrideSignal` con la siguiente condición de triggering requerida:

$$\begin{aligned} &\mathit{ReqTrig}(\texttt{overrideSignal}) = \\ &\quad \neg\texttt{Overridden} \wedge \texttt{Occurs\_block} \wedge \neg\texttt{PressureAbovePermit} \end{aligned} \tag{ESFAS7}$$

ITERACIÓN 9. Luego de este refinamiento, $O_7 = O_6 \cup \{\mathit{ReqTrig}_2(\texttt{enableSignal})\}$. LTSA encuentra un contraejemplo para los objetivos sobre O_7:

```
Trace to property violation in Goals:
  tick    Overridden && PressureBelowLow
  tock    Overridden && PressureBelowLow
  raisePressureAboveLow   Overridden
  tick    Overridden
  tock    Overridden
  raisePressureAbovePermit    PressureAbovePermit && Overridden
  tick    PressureAbovePermit && Overridden (s2)
  tock    PressureAbovePermit && Overridden
  tick    PressureAbovePermit && Overridden (s3)
Analysed in: 7ms
```

La traza anterior muestra una violación a `SIEnabledWhenPressureAbovePermitOrManualReset`. Luego, el interpolante computado es: $\texttt{Overridden3} \wedge \texttt{PressureAbovePermit2}$. Para remover esta violación, nuestra técnica forzará la operación `enableSignal` cuando:

$$\mathit{ReqTrig}_2(\texttt{enableSignal}) = \texttt{Overridden} \wedge \texttt{PressureAbovePermit} \tag{ESFAS8}$$

ITERACIÓN 10. Finalmente, agregamos la última condición de triggering refinada y obtenemos $O_8 = O_7 \cup \{\mathit{ReqTrig}_2(\texttt{enableSignal})\}$. Al analizar la validez de los objetivos sobre O_8, LTSA retorna:

```
No deadlocks/errors
Analysed in: 10ms
```

Por lo tanto, nuestra técnica finaliza luego de 10 iteraciones, retornando el Modelo Operacional O_8, que es una operacionalización correcta de los objetivos

$$ReqPre(\texttt{stopSignal}) = \neg(\neg\texttt{Overridden} \wedge \texttt{PressureBelowLow}) \quad \text{(ESFAS1)}$$
$$ReqPre_1(\texttt{overrideSignal}) = \neg\texttt{Occurs\_reset} \quad \text{(ESFAS2)}$$
$$ReqPre_2(\texttt{overrideSignal}) = \neg\texttt{PressureAbovePermit} \quad \text{(ESFAS3)}$$
$$ReqPre(\texttt{enableSignal}) = \neg(\neg\texttt{PressureAbovePermit} \wedge \texttt{Occurs\_block}) \quad \text{(ESFAS4)}$$
$$ReqTrig(\texttt{sendSignal}) = \quad \text{(ESFAS5)}$$
$$\neg\texttt{SafetyInjection} \wedge \neg\texttt{Overridden} \wedge \texttt{PressureBelowLow}$$
$$ReqTrig_1(\texttt{enableSignal}) = \texttt{Overridden} \wedge \texttt{Occurs\_reset} \quad \text{(ESFAS6)}$$
$$ReqTrig(\texttt{overrideSignal}) = \quad \text{(ESFAS7)}$$
$$\neg\texttt{Overridden} \wedge \texttt{Occurs\_block} \wedge \neg\texttt{PressureAbovePermit}$$
$$ReqTrig_2(\texttt{enableSignal}) = \texttt{Overridden} \wedge \texttt{PressureAbovePermit} \quad \text{(ESFAS8)}$$

Figura 32: Operacionalización Correcta para `ESFAS`.

de safety y de la propiedad de progreso TP. Durante el proceso de operacionalización, nuestra técnica computó 8 condiciones requeridas, ilustradas en la Figura 32. Las condiciones requeridas refinadas desde (ESFAS1)-(ESFAS4) remueven las violaciones a la propiedad TP, y las refinadas desde (ESFAS5)-(ESFAS8) garantizan la satisfacción de los objetivos de safety.

Note que, por ejemplo, en (ESFAS5), ¬`SafetyInjection` es redundante en la condición de triggering requerida de `sendSignal`, debido a que ¬`SafetyInjection` es la precondición de dominio de `sendSignal`. El enfoque garantiza que la condición de triggering requerida (ESFAS7) de `overrideSignal` no contradice sus precondiciones requeridas (ESFAS2) y (ESFAS3), gracias a la primer asunción ambiental, la cuál expresa que `Occurs_block` ⇒ ¬`Occurs_reset`. La segunda asunción es usada para justificar que las condiciones de triggering requeridas (ESFAS6) y (ESFAS8) de `enableSignal` no contradicen su precondición requerida (ESFAS4).

Comparación con ILP-framework

Cuando comparamos nuestra técnica con ILP-framework, notamos que ambas técnicas iteran el mismo número de veces. Sin embargo, la primer diferencia aparece en la sexta iteración. ILP-framework remueve el contraejemplo, pero debido al problema de sobre generalización de ILP, produce una condición de triggering demasiado débil para `sendSignal`: `PressureBelowLow` (es decir, en vez de ¬`Overridden` ∧ `PressureBelowLow` como en (ESFAS5)). Esta condición

fuerza a `sendSignal` a ocurrir cuando la presión de agua esta por debajo de 'Low' sin importar si el sistema está bloqueado o no.

En la séptima y novena iteración, ambos enfoques producen las mismas condiciones requeridas. En la octava iteración sin embargo, una segunda diferencia es detectada. El ILP-framework aprende una condición de triggering más débil para `overrideSignal`: `Occurs_block` (note la diferencia con (ESFAS7)). Esta condición aprendida produce un deadlock en el sistema, por razones similares al caso del Mine Pump Controller. Para remover este deadlock, ILP-framework requiere al ingeniero volver a previas iteraciones y refinar manualmente la condición requerida, o deberá proveer más escenarios positivos y negativos, y ejecutar nuevamente el proceso de operacionalización.

Respecto a los tiempos de ejecución, como ambos enfoques utilizan LTSA, la parte comparativa en tiempo es Interpolación + SAT vs. Inductive Learning. En el caso de ILP-framework, el tiempo crece de 6 segundos en la primer iteración a 18 segundos en la octava, porque el conjunto de ejemplos se incrementa (el ingeniero acumula los escenarios positivos de cada iteración). Nuestro enfoque requiere menos de 1 segundo por iteración.

4.7.3 *Analizando propiedades de Reactividad*

Enfoques previos para operacionalización de objetivos no lidian con objetivos de liveness, por lo que no tenemos resultados previos o casos de estudio con cuales comparar, para validar nuestra técnica. En particular, el enfoque basado en patrones de Letier clasifica estos objetivos como irrealizables [Letier, 2001]. Por lo tanto, las especificaciones que utilizamos (MinePumpController, ESFAS) no poseen objetivos de liveness asignados a agentes, para ser operacionalizados.

Por esta razón, hemos desarrollado algunos objetivos de liveness, basados en el conocimiento que tenémos de los modelos, que deben ser satisfechos, a pesar de que no aparecen explícitamente en la especificación. Como explicamos en la Sección 4.6, nuestra metodología indica que las propiedades de reactividad deben ser operacionalizadas luego de los objetivos de progreso y safety. Por lo tanto, como nuestros objetivos de liveness son artificiales, deberemos remover algunos objetivos de safety que implican a nuestras propiedades desarrolladas. De esta manera, podremos mostrar como nuestra técnica es capaz de remover violaciones de liveness. Además, como explicamos en Sección 4.5, no usaremos asunciones de fairness, y deberemos considerar ciertas asunciones de liveness sobre el ambiente (por ejemplo, si al bomba esta encendida, eventualmente el nivel del agua no será alto).

MINE PUMP CONTROLLER. En la Sección 4.5 mostramos como nuestra técnica operacionaliza propiedades de liveness, utilizando como ejemplo el `MinePumpController`. En particular, la propiedad de reactividad que desarrollamos para este caso fue: "Si infinitas veces no hay metano en el ambiente (así la bomba puede ser prendida), luego infinitas veces el nivel del agua no será alto". Para poder obtener un contraejemplo para este objetivo, tuvimos que remover de la Figura 29, las condiciones (MP4) y (MP6) que operacionalizaban el objetivo `PumpOnWhenHighWaterANDNoMethane`. De esa manera, LTSA produjo el contraejemplo mencionado en la Sección 4.5, y fue removido computando una condición de triggering requerida para `switchPumpOn`:

$$ReqTrig(\texttt{switchPumpOn}) = \neg\texttt{Methane} \wedge \neg\texttt{LowWater} \wedge \texttt{HighWater} \quad \text{(MP-L1)}$$

Puede encontrar los detalles de este refinamiento en la Sección 4.5.

Reactividad para ESFAS

Supongamos que, para el sistema ESFAS, debemos garantizar la siguiente propiedad: *"Si infinitas veces el usuario no bloquea el sistema presionando el botón de bloqueo, entonces infinitas veces la presión del agua será baja"*. Podemos formalizar en FLTL esta propiedad como:

```
assert Reactividad = (□◊¬Occurs_block → □◊¬PressureBelowLow)
```

Para este caso, la expectativa que asumimos sobre el ambiente, indica que: "Si el sistema de inyección segura se enciende, eventualmente el nivel de la presión de agua no va a ser bajo". Esta expectativa puede formalizarse como:

```
assert Expectativa = □(SafetyInjection → ◊¬PressureBelowLow)
```

Finalmente, el objetivo de liveness a ser operacionalizado es el siguiente:

```
assert LivenessGoal= Expectativa ∧ Reactividad
```

Asumiendo que no tenemos la condición (ESFAS5) en el Modelo Operacional, y removiendo la asunción de *strong fairness* de LTSA, obtenemos el siguiente contraejemplo:

```
Violation of LTL property: @LivenessGoal
Trace to terminal set of states:
  tick    PressureBelowLow  (s0)
  tock    PressureBelowLow
  tick    PressureBelowLow  (s1)
```

```
Cycle in terminal set:
  tock    PressureBelowLow
  tick    PressureBelowLow  (s1)
LTL Property Check in: 48ms
```

Note que la parte del ciclo del contraejemplo de reactividad, está formado por un sólo estado observable: `(s1)`. En este estado, la asunción ¬`Occurs_block` es válida, pero no vale el objetivo ¬`PressureBelowLow`.

Para remover esta violación, nuestra técnica procede a computar un interpolante para el contraejemplo y la propiedad de reactividad. Primero debemos codificar el contraejemplo en una fórmula proposicional, tal fue explicado en la Sección 4.5. La Figura 33 muestra la especificación del contraejemplo en MSAT. Por otro lado, podemos codificar la propiedad de liveness, en la siguiente fórmula proposicional:

```
#Expectativa
(SafetyInjection1 -> !PressureBelowLow1)
#Reactividad
& !Occurs_block1 -> !PressureBelowLow1
```

Considerando el contraejemplo de la Figura 33 y la codificación de la propiedad de reactividad, podemos computar un interpolante, obteniendo: ¬`SafetyInjection1` ∧ ¬`Occurs_block1` ∧ `PressureBelowLow1`.

Note que este interpolante es una representación más débil del ciclo, centrandose sólo en las variables relevantes para la propiedad de reactividad. Para remover este contraejemplo, nuestra técnica busca una operación que pueda ser ejecutada en el estado `(s1)`, tal que su ejecución nos lleva a un estado donde no vale el interpolante. La operación `sendSignal` cumple con estas condiciones:

$$\neg\texttt{SafetyInjection} \wedge \neg\texttt{Occurs\_block} \wedge \texttt{PressureBelowLow} \rightarrow \neg\texttt{SafetyInjection}$$

$$\begin{aligned}&(\neg\texttt{SafetyInjection} \wedge \neg\texttt{Occurs\_block} \wedge \texttt{PressureBelowLow})\\ &\wedge (\texttt{SafetyInjection}' = \mathit{True} \wedge \mathit{Unchanged}(\texttt{Occurs\_block}, \texttt{PressureBelowLow}, ...))\\ &\quad \rightarrow \neg(\neg\texttt{SafetyInjection}' \wedge \neg\texttt{Occurs\_block}' \wedge \texttt{PressureBelowLow}')\end{aligned}$$

Luego, nuestra técnica debilita la condición de triggering requerida para la operación `sendSignal`, agregando un nuevo disyunto, formado por la conjunción entre la precondición requerida (que es True) y la negación del objetivo ¬¬`PressureBelowLow`. Por lo tanto, la condición requerida computada es:

$$\mathit{ReqTrig}(\texttt{sendSignal}) = \neg\neg\texttt{PressureBelowLow} \qquad \text{(ESFAS-L1)}$$

```
VAR
#Variables
PressureBelowLow0, PressureAbovePermit0: BOOLEAN
PressureBelowLow1, PressureAbovePermit1: BOOLEAN
Occurs_reset0,Occurs_block0, Occurs_reset1, Occurs_block1: BOOLEAN
Overridden0, SafetyInjection0, Overridden1, SafetyInjection1: BOOLEAN
#Operaciones
raisePressureAboveLow0, lowerPressureBelowLow0, block0, reset0: BOOLEAN
raisePressureAbovePermit0, lowerPressureBelowPermit0, tock0: BOOLEAN
overrideSafetyInjection0, enableSafetyInjection0: BOOLEAN
sendSafetyInjectionSignal0, stopSafetyInjectionSignal0: BOOLEAN
raisePressureAboveLow1, lowerPressureBelowLow1, block1, reset1: BOOLEAN
raisePressureAbovePermit1, lowerPressureBelowPermit1, tock1: BOOLEAN
overrideSafetyInjection1, enableSafetyInjection1: BOOLEAN
sendSafetyInjectionSignal1, stopSafetyInjectionSignal1: BOOLEAN

FORMULA
# I(s0)
!Occurs_reset0 & !Occurs_block0 & PressureBelowLow0 & !PressureAbovePermit0
& Overridden0 & !SafetyInjection0
# R(s0, s1)
...
# R(s1, s1)
& ((raisePressureAboveLow1 & !PressureBelowLow1)
| (lowerPressureBelowLow1 & PressureBelowLow1)
| (!raisePressureAboveLow1 & !lowerPressureBelowLow1))
& ((raisePressureAbovePermit1 & PressureAbovePermit1)
| (lowerPressureBelowPermit1 & !PressureAbovePermit1)
| (!raisePressureAbovePermit1 !lowerPressureBelowPermit1))
& ((block1 & Occurs_block1) | (tock1 & !Occurs_block1) | (!block1 & !tock1))
& ((reset1 & Occurs_reset1) | (tock1 & !Occurs_reset1) | (!reset1 & !tock1))
& ((overrideSafetyInjection1 & Overridden1)
| (enableSafetyInjection1 & !Overridden1)
| (!overrideSafetyInjection1 & !enableSafetyInjection1))
& ((sendSafetyInjectionSignal1 & SafetyInjection1)
| (stopSafetyInjectionSignal1 & !SafetyInjection1)
| (!sendSafetyInjectionSignal1 & !stopSafetyInjectionSignal1))

# counterexample
& !raisePressureAboveLow0 & !lowerPressureBelowLow0
& !raisePressureAbovePermit0 & !lowerPressureBelowPermit0
& !block0 & !reset0 & tock0
& !overrideSafetyInjection0 & !enableSafetyInjection0
& !sendSafetyInjectionSignal0 & !stopSafetyInjectionSignal0

& !raisePressureAboveLow1 & !lowerPressureBelowLow1
& !raisePressureAbovePermit1 & !lowerPressureBelowPermit1
& !block1 & !reset1 & tock1
& !overrideSafetyInjection1 & !enableSafetyInjection1
& !sendSafetyInjectionSignal1 & !stopSafetyInjectionSignal1
```

Figura 33: Especificación MSAT para el ESFAS.

Luego de agregar esta condición de triggering requerida, LTSA no detecta más violaciones de liveness. Note que en este caso, la anterior condición de triggering requerida es más débil que (ESFAS5), poniendo en evidencia que nuestra técnica no puede garantizar minimalidad.

4.8 TRABAJOS RELACIONADOS

Los métodos orientados a objetivos (como KAOS [Dardenne et al., 1993] e I* [Yu, 1997]) han sido el foco de numerosos trabajos de la Ingeniería de Requisitos. Un importante aspecto de este tipo de enfoques, es la noción de relacionar objetivos de alto nivel, que sólo pueden ser alcanzados mediante la cooperación de varios agentes, con objetivos de bajo nivel que pueden ser asignados a agentes específicos, algunos de los cuales son componentes de software a construir. El refinamiento de objetivos ha sido extensamente estudiado, por ejemplo, en [Anton, 1997].

La operacionalización de objetivos apunta a producir requisitos en base a cada operación que será provista por un agente específico para garantizar la satisfacción del objetivo del cual es responsable. Entre las técnicas para operacionalización de objetivos podemos mencionar al framework NFR [Mylopoulos et al., 1992] y CREWS [Rolland et al., 1998]. Sin embargo, estos enfoques se basan en *requisitos no funcionales*, o son informales, por lo que no pueden ser completamente verificados. Enfoques más formales, como [Fuxman et al., 2001; Fuxman et al., 2004], permiten verificar la correctitud de una operacionalización, pero no soportan la elaboración del Modelo Operacional.

El uso de generalización en el contexto de modelos de objetivos no es novedoso. Por ejemplo, [van Lamsweerde and Willemet, 1998] presenta un método para inferir aserciones declarativas a partir de escenarios. Este método induce objetivos desde escenarios provistos por los stakeholders, usando un proceso de inferencia inductivo basado en Explanation-Based Learning (EBL) [Mitchell, 1997]. Aparte de que no puede usarse específicamente para operacionalización, la técnica de aprendizaje de [van Lamsweerde and Willemet, 1998] no considera conocimiento previo existente (por ejemplo, los objetivos y requisitos operacionales) durante el proceso de inferencia. Esto hace que el método no sea correcto y pueda producir especificaciones inconsistentes.

Interpolación ha sido utilizado para analizar software, notablemente por McMillan [Mcmillan, 2005], en combinación con model checking basado en SAT, para la verificación de circuitos [McMillan, 2003]. También ha sido utilizado para algoritmos automáticos de abstracción CEGAR (counterexample guided abstraction refinement) [Clarke et al., 2003]. Esencialmente, interpolación es

usado en el contexto de la verificación de modelos abstractos (imprecisos). Cuando un contraejemplo es obtenido, se debe chequear si el contraejemplo es real o es producto de la imprecisión del modelo. Si el contraejemplo es espurio, al construir la conjunción del modelo concreto con el contraejemplo, obtenemos una fórmula insatisfactible. Interpolación *explica* cuál es la diferencia entre el modelo abstracto y concreto del sistema, que nos conduce al contraejemplo espurio. El interpolante obtenido puede agregarse al modelo abstracto para hacerlo más preciso, y así remover el anterior contraejemplo espurio. Este proceso es iterado hasta que no se encuentran más violaciones, o hasta que un contraejemeplo real (no espurio) es producido. En este trabajo, proponemos usar interpolación para un propósito diferente, pero relacionado. Básicamente, desde los contraejemplos concretos mostrando violaciones a objetivos, producimos fórmulas insatisfactibles para las cuales podemos computar interpolantes. Estos interpolantes son usados para refinar el Modelo Operacional concreto. A diferencia del refinamiento de abstracciones, en nuestro caso no tenemos un modelo de referencia para realizar los refinamientos (el modelo concreto en el caso del refinamiento de abstracciones). En cambio, tenemos *objetivos* que debemos operacionalizar de forma correcta. La desviación desde estos objetivos es lo que guía nuestro proceso en el uso de interpolación, para refinar el Modelo Operacional.

Otros enfoques relacionados son los trabajos sobre síntesis de Modelos de Comportamiento (en forma de LTS) a partir de objetivos declarativos [D'Ippolito et al., 2010; D'Ippolito et al., 2011; Letier and Heaven, 2013]. A diferencia de nuestro enfoque, el cual produce condiciones pre/triggering requeridas *declarativas* para operaciones controladas, estos trabajos basados en síntesis computan el comportamiento operacional para las operaciones controladas.

Como mencionamos anteriormente, la técnica presentada en este capítulo está muy relacionada al framework presentado en [Alrajeh et al., 2009], el cual provee un método semi-automático que utiliza model checking para verificar la satisfacción de los objetivos e Inductive Logic Programming (ILP) para aprender los requisitos operacionales faltantes. Hay varias diferencias importantes con nuestro enfoque. [Alrajeh et al., 2009] requiere la intervención del usuario para que provea escenarios positivos para la etapa de learning, por lo que su resultado depende de la consistencia y "riqueza" de los escenarios provistos [Alrajeh et al., 2013], mientras que en nuestro enfoque esto no es requerido. [Alrajeh et al., 2009] utiliza ILP, el cual trata de buscar las condiciones mas comprimidas (es decir, con el menor número de fluentes apareciendo en las condiciones requeridas), por lo que es propenso a generar condiciones sobre-generalizadas (tal como mostramos en los casos de estudio de la Sección 4.7). Nuestra técnica, por otro lado, usa interpolación, lo cuál produce condiciones más precisas, ya que el interpolante es necesariamente implicado por el contraejemplo y necesariamente nos conduce

a la violación del objetivo. Exactamente por estas razones, el refinamiento basado en interpolación podría requerir más iteraciones para alcanzar una operacionalización correcta, comparado con el enfoque basado en ILP.

Finalmente, nuestra técnica es capaz de lidiar con un amplio rango de propiedades de liveness, las cuales no pueden ser manejadas por ningún enfoque previo para la operacionalización de objetivos.

4.9 RESUMEN

En este capítulo presentamos un enfoque para la operacionalización de objetivos, que *automáticamente* computa condiciones pre/triggering requeridas para operaciones, para satisfacer un conjunto de objetivos. Nuestro enfoque no depende de información provista por el usuario ni de sus caracteríasticas, como ocurre en [Alrajeh et al., 2009], que depende de la consistencia y riqueza de los escenarios provistos por el usuario. Además, nuestro enfoque se basa en Interpolación y SAT Solving, y aplica a objetivos de *safety* y un tipo particular bastante amplio de propiedades de *liveness*, expresables con el patrón de *reactividad*. Hemos evaluado nuestra técnica sobre dos modelos tomados de la literatura, y comparado con el enfoque semi-automático presentado en [Alrajeh et al., 2009], mostrando que en estos casos nuestro enfoque puede operacionalizar los objetivos de forma efectiva y más eficiente.

5

Validación y Verificación de Requisitos de Software

5.1 Introducción

En el capítulo anterior presentamos una técnica para la elaboración formal de requisitos de software, que particularmente puede ser utilizada en las *etapas tempranas* del proceso de ingeniería de requisitos. La etapa de elaboración de requisitos evoluciona hasta que finalmente obtenemos una especificación formal precisa, a la cual podemos analizar automáticamente para detectar ambigüedades, inconsistencias e incompletitudes. En general, estos tipos de análisis se llevan a cabo en lo que se conoce como *etapa tardía* del proceso de ingeniería de requisitos.

Como mencionamos en la Sección 3.5, SCR es un lenguaje exitosamente utilizado para la elaboración, especificación y análisis de sistemas con componentes lógicos complejos, como los sistemas de seguridad crítica o tolerantes a fallas. Entre los ejemplos más exitosos en los que se utilizó SCR, podemos mencionar: el programa operacional de vuelo del *A-7 aircraft* [K. Heninger and Shore, 1978], un sistema de comunicaciones submarinas [Heitmeyer and McLean, 1983], y componentes de seguridad críticos de la planta nuclear de Darlington en Canada [van Schouwen et al., 1993]. Actualmente, existe una amplia gama de herramientas de análisis para especificaciones SCR, que se concentran en el denominado *SCR Toolset* [Heitmeyer et al., 2005], que no sólo permiten analizar la consistencia y completitud de la especificación, sino que además podemos generar escenarios de ejecución y verificar propiedades temporales, de manera automática o asistida, usando una variedad de técnicas, como el model checking y la demostración automática.

Sin embargo, el análisis automático de las especificaciones SCR puede ser muy complicado mayormente por dos factores. Primero, las especificaciones de requisitos SCR usualmente contienen muchas variables que se refieren a medidas del mundo real, debido a que SCR busca alcanzar una buena precisión de sus requisitos ya que, en general, se aplica a sistemas críticos. Por ejemplo, una variable entera `mAlt` puede denotar la altitud de un avión, que se extiende desde

0 a 10000 pies. Luego, el espacio de estados completo de las especificaciones SCR usualmente resulta ser muy grande. Segundo, es muy común que muchos eventos estén habilitados a disparar un cambio de estados entre el estado actual y el siguiente, por lo que explorar los estados alcanzables del sistema resulta exponencialmente complejo. Debido a estos problemas, las técnicas automáticas como model checking no pueden lidiar con especificaciones SCR de tamaño moderado y grandes. Luego, estas técnicas son típicamente aplicadas a abstracciones manuales ad hoc, o versiones reducidas de las especificaciones [Heitmeyer et al., 2005]. Por ejemplo, es frecuente que los rangos de las variables numéricas sean manualmente reducidos antes de usar model checking, como tal podría ser el caso de la variable `mAlt` siendo restringida a tomar valores entre 0 y 100 pies (en vez de 10000).

Los dos enfoques mencionados anteriormente tienen varias desventajas. Las abstracciones manuales ad hoc requieren una cuidadosa observación de la especificación original y la propiedad a verificar por una persona con experiencia, una tarea que resulta muy costosa y consume demasiado tiempo. Por otro lado, trabajar con especificaciones reducidas también puede ser muy problemático. Por ejemplo, verificar la propiedad sobre la especificación reducida no garantiza la ausencia de errores en la especificación original. Además, los casos de tests generados sobre la especificación reducida no pueden ser utilizados para evaluar el comportamiento del sistema que implementa la especificación original (los tests deberían ser adaptados manualmente).

Como las especificaciones de requisitos de software son utilizadas para tareas de validación – contrastar las expectativas del usuario contra los requerimientos – y verificación – contrastar el comportamiento del sistema contra los requerimientos – es deseable mantener el *nivel de detalle original* de estas especificaciones. Esto, y los problemas ya mencionados, motivan la necesidad de desarrollar técnicas automáticas de abstracción que sean capaces de lidiar con especificaciones de requisitos tales como fueron originalmente diseñadas, una tarea muy difícil para los enfoques actuales.

En este capítulo presentamos una novedosa combinación entre abstracción y model checking para analizar especificaciones SCR, en particular, aplicada a la verificación de propiedades y la generación de casos de test. Como mostraremos en la Sección 5.6, nuestra técnica es más eficiente y escala mejor a especificaciones grandes que las técnicas de análisis existentes.

Dada una especificación SCR y la propiedad a verificar como entrada, nuestra técnica comienza "relajando" (*debilitando*) las restricciones sobre las variables monitoreadas numéricas de la especificación SCR, permitiendo que estas puedan cambiar arbitrariamente dentro de su dominio en respuesta a

eventos de entrada (en vez de pequeños pasos, como ocurre usualmente). La *especificación relajada* luego es analizada utilizando un algoritmo de *abstracción Lazy* especialmente diseñado para SCR, presentado en [Degiovanni et al., 2011]. Note que la especificación relajada es una *abstracción conservativa* de la original, es decir, que considera más ejecuciones. Por ejemplo, note que los eventos pueden modificar las variables numéricas en pequeños intervalos en la especificación original, mientras que en la relajada pueden cambiar arbitrariamente dentro de su rango. Luego, las propiedades de safety que pueden ser probadas sobre la especificación relajada, también son válidas en la especificación original.

La ventaja de ejecutar la abstracción Lazy sobre la especificación relajada es que el número de predicados de abstracción necesarios para que el análisis converja es significativamente menor al necesario para analizar la especificación original. Intuitivamente, esto ocurre porque la mayoría de las especificaciones SCR contienen variables numéricas con rangos muy grandes, y que usualmente están restringidas a cambiar de a pequeños pasos (ver Sección 3.5). Por ejemplo, en la especificación del piloto automático de un avión [Bharadwaj and Heitmeyer, 1997], la variable `mAlt` que representa la altitud va de 0 a 10000 pies, como ya mencionamos. Pero además, la especificación establece que "el avión no puede cambiar su altitud en más de 50 pies sin que el sistema lo detecte". De esta manera, si el proceso de abstracción debe rastrear el valor de `mAlt` desde 0 a 5000 pies (por ejemplo), deberá introducir 500 predicados de abstracción para representar los estados intermedios. Esto puede hacer que el procedimiento de abstracción no finalice, debido al enorme espacio de estados a explorar (el tamaño del modelo abstracto es exponencial al número de predicados). Sin embargo, rastrear tantos estados intermedios no es necesario en la práctica para probar un número importante de propiedades de safety (ver Sección 5.6). Así, la intuición de nuestra técnica es que, cuando utilizamos la especificación relajada como entrada, el algoritmo de abstracción Lazy evita introducir predicados para los estados intermedios de las variables numéricas, en casos donde no es necesario conocer el valor específico de estas variables para probar una propiedad de safety.

Como mencionamos previamente, la especificación relajada considera más comportamientos que la original. Luego, la ejecución de la abstracción Lazy puede producir *falsos positivos*: ejecuciones sobre la especificación relajada que violan la propiedad que no pueden ser reproducidas en la original. Para esto, proponemos un método de concretización para las ejecuciones (abstractas) de la especificación relajada, y una técnica automática basada en model checking para generar ejecuciones concretas desde las abstractas, si existe alguna. De otra manera, proveemos una técnica correcta (pero incompleta) para refinar la especificación relajada para remover los falsos positivos. Es importante remarcar

que el análisis de las ejecuciones abstractas (provenientes de la especificación relajada) es muchas veces más simple para el model checker que el análisis de la especificación completa.

Para concluir, evaluamos experimentalmente nuestro enfoque sobre la verificación de propiedades (safety) y la generación de casos de test para especificaciones SCR, comparándolo con técnicas relacionadas. Los casos de estudios considerados en los experimentos abarcan especificaciones previamente utilizadas para evaluar las técnicas de análisis automáticas para SCR [Gargantini and Heitmeyer, 1999]. Es importante remarcar que nosotros utilizamos las especificaciones tales como aparecen en la literatura, en su nivel de detalle original, sin aplicar ninguna abstracción manual o reducción de la especificación. Los resultados de los experimentos muestran que nuestra técnica es más eficiente y escala mejor para especificaciones grandes, permitiendonos verificar propiedades y generar casos de tests para especificaciones que no pueden ser manejadas de buena manera por enfoques relacionados.

El resto del capítulo se organiza como sigue: la Sección 5.2 motiva la técnica, describiendo el problema de aplicar abstracción cuando la especificación SCR contiene demasiadas variables numéricas; la Sección 5.3 presenta nuestro algoritmo de abstracción especialmente diseñado para analizar especificaciones de requisitos SCR; y la Sección 5.4 presenta en detalle toda nuestra técnica para analizar especificaciones de requisitos, utilizando el algoritmo de abstracción de la sección anterior. La Sección 5.5 describe en que consisten la generación de casos de test y la verificación de invariantes para especificaciones de requisitos SCR. Luego, presentamos una extensa evaluación experimental de nuestra técnica y discutimos los resultados obtenidos en la Sección 5.6. Finalmente, presentamos los trabajos relacionados en la Sección 5.7 y concluimos el capítulo en la Sección 5.8.

5.2 SCR, ABSTRACCIÓN, Y EL PROBLEMA CON LAS VARIABLES NUMÉRICAS

Previamente mencionamos que la eficiencia de nuestro algoritmo de abstracción Lazy puede verse afectada cuando la propiedad a analizar depende fuertemente de variables monitoreadas numéricas, con dominio grande y variación pequeña. Para ilustrar mejor este problema, considere el Safety Injection System (SIS) presentado en Sección 3.5, cuyo objetivo es mantener el nivel de presión del agua (representada por la variable `mWaterPres`) en un nivel aceptable (representado por el modo `Permitted`). El rango de `mWaterPres` es de 0 a 5000, se asume que inicialmente vale 14 y puede cambiar en a lo sumo

10 unidades de un estado a otro, una asunción que forma parte de la relación NAT, relacionada a la frecuencia con la que el sistema monitorea la variable. La Tabla 6 muestra la Tabla de Transición de Modos para el SIS, donde puede observarse que cuando el nivel del agua es menor a 900, entonces el modo del sistema es `TooLow`, entre 900 y 3999 es `Permitted`, y si es mayor o igual a 4000, el modo es `High`.

Supongamos que queremos verificar si es posible alcanzar el modo `Permitted`. Observe que, como inicialmente el nivel del agua es 14 y su variación es a lo sumo 10, la variable `mWaterPres` debe ser incrementada al menos 89 veces (para superar a la constante `Low`). Para verificar si `Permitted` es alcanzable, podríamos utilizar nuestro algoritmo de abstracción Lazy adaptado a especificaciones SCR, que introduciremos en detalle en la Sección 5.3. Sin embargo, este algoritmo debería refinar demasiados predicados de abstracción, y probablemente falle en el proceso de verificación. La razón por la que este enfoque debe refinar demasiados predicados es que debe caracterizar los 89 estados intermedios - uno por cada evento modificando `mWaterPres`- necesarios para que el sistema se mueva de modo `TooLow` a `Permitted`. Los predicados descubiertos tienen la forma `mWaterPres<=i`, con `i=14,24,34,44,...,894` (`mWaterPres` puede ser incrementada hasta en 10 unidades por vez). Claramente, `mWaterPres<=i` describe el conjunto de estados concretos donde el valor de `mWaterPres` es menor o igual a la constante `i`.

Luego de descubrir los predicados ya mencionados, nuestro algoritmo Lazy podría producir una traza abstracta mostrando una posible forma de alcanzar el modo `Permitted`, tal como muestra la Figura 34. El estado `s0` de la traza es una abstracción del estado inicial en el cual `mWaterPres=14`. En `s0` todos los predicados `mWaterPres<=i` (con `i=14,24,...,894`) son verdaderos, ya que inicialmente es cierto que `mWaterPres<=14`, `mWaterPres<=24`, etc. Comenzando en `s0`, el primer evento que se ejecuta en la Figura 34 incrementa el valor de `mWaterPres` en 10 unidades, llevando al sistema al estado abstracto `s1`. Ahora, como `mWaterPres` se incremento en 10 y en `s0` valía que `mWaterPres<=14`, en `s1` no podemos asegurar si `mWaterPres<=14` vale o no. Por lo que utilizados una `*` para representarlo en la traza, indicando que ese predicado puede ser verdadero o falso. Lo que si podemos asegurar es que `mWaterPres<=24` vale en `s1`, por lo que `s1` representa todos los estados concretos en los que `mWaterPres` es menor o igual a 24. De manera similar, al incrementar en 10 nuevamente `mWaterPres` desde `s1`, ya no podemos asegurar en `s2` que `mWaterPres<=24`, pero si vale que `mWaterPres<=34`. Por lo que el estado abstracto `s2` representa todos los estados concretos donde `mWaterPres` es menor o igual a 34. En lineas generales, s_i denota los estados donde `mWaterPres<=i`, para `i` desde 14 a 894. Finalmente,

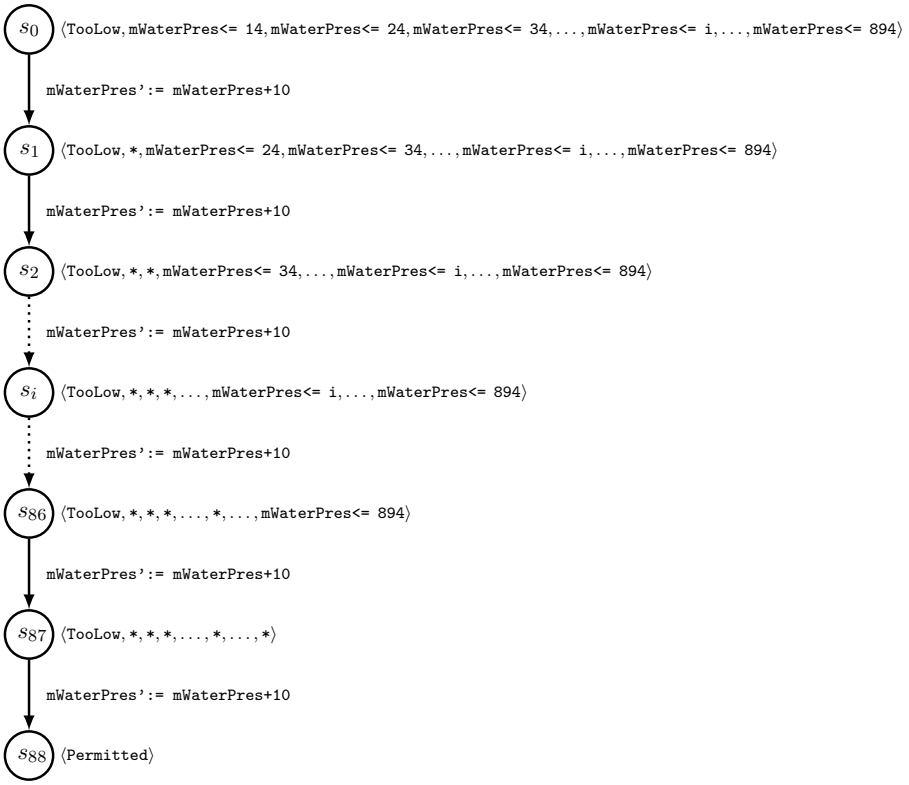

Figura 34: Contrajemplo abstracto generado por la abstracción Lazy.

en el estado abstracto `s87` vale que `mWaterPres<=894`, y al incrementar en 10 la variable `mWaterPres` el sistema puede moverse al modo `Permitted`.

En abstracción por predicados el tiempo requerido para construir el modelo abstracto es exponencial al número de predicados. Esto explica la imposibilidad de construir eficientemente un modelo abstracto para el ejemplo anterior. Incluso para hacer peor las cosas, note que el número de predicados puede crecer muy rápidamente. Por ejemplo, si la variación aceptada para `mWaterPres` fuera de a lo sumo 5 en vez de 10, es fácil de ver que la técnica de abstracción recién descripta necesitará refinar el doble de predicados. Es importante remarcar que este tipo de problema no es particular al modelo del SIS, sino que muchas especificaciones SCR de la literatura lo sufren [Bultan and Heitmeyer, 2008; Heitmeyer et al., 2005; Fraser and Gargantini, 2009].

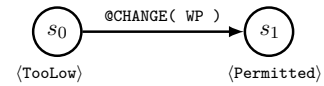

Figura 35: Contraejemplo abstracto generado luego de debilitar la relación NAT.

5.2.1 *Un nuevo método de abstracción para SCR*

Para hacer frente a la dificultad mencionada anteriormente, vamos a realizar un paso extra de abstracción antes de ejecutar nuestro algoritmo de abstracción Lazy sobre la especificación SCR. Este paso de abstracción consiste en *relajar* (debilitar) las restricciones impuestas por NAT sobre las variables monitoreadas numéricas de la especificación. Por ejemplo, para el caso del SIS, vamos a remover la restricción que NAT impone sobre la variable `mWaterPres` indicando que puede variar a lo sumo en 10 unidades. Luego de este paso de abstracción obtenemos una *especificación relajada*, cuyos eventos de entrada (ver Subsección 3.5.1) permiten modificar la variable `mWaterPres` de manera arbitraria dentro de su dominio, es decir, de un valor a cualquier otro.

Observe que la especificación relajada es una abstracción de la original, en el sentido de que permite más comportamientos. Todos los comportamientos de la especificación original aún son válidos en la especificación relajada, pero algunos comportamientos de esta última no son permitidos en la especificación original (por ejemplo, incrementar de un sólo paso en 20 unidades a la variable `mWaterPres` en el SIS).

En la sección previa argumentamos que es prácticamente imposible que nuestro algoritmo de abstracción Lazy puede generar un traza que alcance el modo `Permitted` a partir del estado inicial, siendo posible generar dicha ejecución. Veamos que ocurre si tratamos de solucionar este mismo problema sobre la especificación relajada.

Recordando la tabla de Transición de Modos del SIS (Tabla 6)), para que el sistema alcance el modo `Permitted` desde `TooLow`, la variable `mWaterPres` debe ser incrementada hasta `Low`(900) o más. Note que en la especificación relajada del SIS sólo un evento es necesario para que `mWaterPres` pase de 14 a un valor mayor o igual a 900. Luego, una ejecución posible retornada por nuestro algoritmo de abstracción Lazy es mostrada en la Figura 35. Para este caso, el algoritmo de abstracción Lazy no necesita refinar ninguna información sobre los cambios realizados a `mWaterPres` para generar esta ejecución, por lo que finaliza en unos pocos segundos.

El ejemplo anterior muestra que la abstracción Lazy puede beneficiarse enormemente luego de relajar la relación NAT de la especificación SCR. Mostraremos

además en la sección experimental (ver Sección 5.6) que esto también ocurre para varias de las especificaciones tomadas de la literatura.

Sin embargo, aún no hemos encontrado una ejecución que alcance `Permitted` en la especificación original del SIS: sólo sabemos que `Permitted` es alcanzable en la especificación relajada. Por lo tanto, proponemos utilizar una técnica de análisis para SCR basada en model checking, para explorar todo el espacio de estados abstractos de la Figura 35.

Aquí es importante aclarar que explorar el espacio de estados de un traza abstracta (como la de la Figura 35) es, en general, mucho más simple que explorar el espacio de estados de la especificación original. Intuitivamente, podemos pensar que estas trazas abstractas nos dan un "fragmento" del espacio de estados de la especificación original. Por ejemplo, la ejecución de la Figura 35 es un fragmento del comportamiento completo del SIS, que comienza en el estado inicial, ejecuta sólo eventos que cambian la variable `mWaterPres` (puede cambiar en a lo sumo 10 unidades), y finaliza cuando alcanza un estado con modo `Permitted`. Note que la especificación original del SIS también considera los eventos que cambian las variables monitoreadas `mBlock` y `mReset`, así como también los estados con modo `High`.

En resumen, nuestro nuevo enfoque primero relaja la especificación SCR, luego ejecuta nuestro algoritmo de abstracción Lazy sobre la especificación relajada, y finalmente genera instancias concretas de ejecuciones sobre la especificación original utilizando model checking. Este enfoque es capaz de responder positivamente sobre la alcanzabilidad del modo `Permitted` en pocos segundos, mientras que si sólo utilizáramos nuestro algoritmo de abstracción Lazy podría quedarse estancado refinando predicados tratando de generar el enorme modelo abstracto.

5.3 ABSTRACCIÓN LAZY PARA ESPECIFICACIONES SCR

En esta sección presentamos nuestro algoritmo de abstracción Lazy para SCR, introducido en [Degiovanni et al., 2011]. Este algoritmo utiliza las clases de modo para localizar los predicados de abstracción, y explota procesos de modularización (ver Subsección 5.3.3) para alcanzar mejor precisión y eficiencia en el análisis.

Sin perdida de generalidad, a lo largo de esta sección asumimos a *Spec* como la especificación SCR a analizar, con clase de modo $M_1, \ldots, M_k$. Para clarificar la presentación, solemos denotar como m a una tupla de modos $(m_1, \ldots, m_n) \in M_1 \times \ldots \times M_k$.

5.3.1 *Abstracción*

Nuestro algoritmo esta basado en el framework de abstracción Lazy presentado en [Henzinger et al., 2002]. Antes de definir nuestro dominio abstracto sobre especificaciones SCR, primero presentamos algunas definiciones generales tomadas de [Henzinger et al., 2002] para comprender mejor este trabajo.

Definición 5.1 (Estructura de Regiones)**.** *Sea* $\mathcal{S} = \langle S, A, \delta, q_0 \rangle$ *un LTS. Una* estructura de regiones $\mathcal{R} = (R, \bot, \sqcup, \sqcap, post, [.])$ *para el LTS* $\mathcal{S}$ *es una estructura que consiste en un conjunto* R *de regiones, un elemento* $\bot \in R$*, funciones totales* $\sqcup, \sqcap : R \times R \to R$*,* $post : R \times A \to R$*, y una función total de* extensión $[.] : R \to 2^S$ *Las siguientes propiedades deben valer para todas las regiones* $r, r' \in R$ *y evento* $e \in A$*:*

$$[\bot] = \emptyset \quad (1) \qquad [r \sqcap r'] = [r] \cap [r'] \quad (3)$$

$$[r \sqcup r'] = [r] \cup [r'] \quad (2) \qquad [post(r, e)] = \{s' \in S \mid \exists s \in [r] \cdot s \overset{e}{\rightsquigarrow} s'\} \quad (4)$$

Intuitivamente, una región r representa el conjunto de estados $[r] \subseteq S$. Además, *post* es conocido como el operador *strongest postcondition*; $post(r, e)$ describe el conjunto de estados que pueden ser obtenidos mediante la ejecución del evento e comenzando en algún estado de $[r]$.

Definición 5.2 (Estructura de Abstracción)**.** *Sea* $\mathcal{S} = \langle S, A, \delta, q_0 \rangle$ *un LTS. Una* estructura de abstracción $\mathcal{A} = (\mathcal{R}, post^{\mathcal{A}}, \trianglelefteq)$ *para el LTS* $\mathcal{S}$ *consiste de una estructura de regiones* $\mathcal{R}$ *para* $\mathcal{S}$*,* $post^{\mathcal{A}} : R \times A \to R$ *es una operador de* strongest postcondition *abstracto, y* $\trianglelefteq \subseteq R \times R$ *es un preorden de precisión.* $post^{\mathcal{A}}$ *debe ser una sobre-aproximación del operador post, es decir,* $post(r, e) \subseteq post^{\mathcal{A}}(r, e)$ *para cualquier* $r \in R, e \in A$*.*

Dentro de una estructura de abstracción, una región r representa un *estado abstracto* cuya concretización es $[r]$. El operador abstracto $post^{\mathcal{A}}$ permite propagar los estados abstractos. El preorden de precisión $\trianglelefteq$ indica que tan cerca está $post^{\mathcal{A}}$ del operador exacto *post*. En otras palabras, si $r \trianglelefteq r'$, diremos que r es más precisa que r', en el sentido de que $post^{\mathcal{A}}$ nos conduce a regiones más precisas. Esto es, para dos regiones equivalentes $r \equiv r'$, si $r \trianglelefteq r'$ luego $post(r, e) \equiv post(r', e) \subseteq post^{A}(r, e) \subseteq post^{A}(r', e)$, para todo evento e.

5.3.2 *Semántica SCR de "modo explítico"*

Para definir nuestro algoritmo de abstracción para SCR, primero debemos extender levemente la semántica original de las especificaciones SCR para

hacerlas de "modo explícito". La idea es hacer explícito el modo del estado actual y siguiente en los eventos que son ejecutados. De esta manera, vamos a partir cada evento de entrada $e = (\texttt{mV}, v, v')$ en varios nuevos eventos de entrada en los cuales el modo origen m y destino m' van a aparecer explícitamente, para todos los modos del sistema.

Así, un *evento de entrada de modo explícito* es un tripla $e_m = (m, e, m')$, con $e = (\texttt{mV}, v, v')$ un evento de entrada. Decimos que e_m cambia el estado del sistema desde s a s', denotado como $s \overset{e_m}{\rightsquigarrow} s'$, si y sólo si, $MD(s) = m$, $MD(s') = m'$ y $s \overset{e}{\rightsquigarrow} s'$ (recuerde que MD retorna el modo único de un estado). Intuitivamente, e_m establece que la variable $\texttt{mV}$ cambia su valor de v a v', y el sistema se mueve del modo m al modo m' (note que m y m' pueden ser el mismo modo). Note además que, para que el evento e_m este habilitado en el estado s debe ocurrir que $s(\texttt{mV}) = v$ y $MD(s) = m$, y debe existir un estado s' con $MD(s') = m'$ tal que $s \overset{e}{\rightsquigarrow} s'$.

Sea $\Sigma = (S, S_0, E^m, T)$ el LTS asociado a la especificación SCR *Spec*, con clase de modo M $(M_1 \times \ldots \times M_n)$, definimos el conjunto de eventos de entrada de modo explícito $E_{M\times M}$ como:

$$E_{M\times M} = \{(m, e, m') | e \in E^m \wedge m, m' \in M\}$$

La semántica de modo explícito para SCR consiste en utilizar el conjunto $E_{M\times M}$ como el conjunto de eventos de entrada en el LTS asociado con la especificación. Si la semántica original para una especificación SCR esta dada por el LTS $\Sigma = (S, S_0, E^m, T)$, la semántica de modo explícito es definida por el LTS $\Sigma_{M\times M} = (S, S_0, E_{M\times M}, T)$. El Teorema 5.1 de abajo asegura que el conjunto de estados alcanzables de $\Sigma_{M\times M}$ coincide con los estados alcanzables de Σ. Además, si podemos verificar una propiedad de safety P usando la semántica de modo explícito, entonces también podemos asegurar que P vale utilizando la semántica original.

Teorema 5.1. *Reach($\Sigma_{M\times M}$)=Reach(Σ).*

Demostración. La prueba de este teorema de deduce inmediatamente desde el Lema 5.2 y Lema 5.3. □

El Lema 5.2 muestra que para cada ejecución de Σ existe una ejecución en $\Sigma_{M\times M}$ que comparte la misma secuencia de estados, mientras que el Lema 5.3 prueba la recíproca.

Lema 5.2. *Para cada ejecución σ de $\Sigma = (S, S_0, E^m, T)$ existe una ejecución σ_m de $\Sigma_{M\times M} = (S, S_0, E_{M\times M}, T)$ tal que $\forall i \cdot 0 \leq i \cdot \sigma(i) = \sigma_m(i)$.*

Demostración. Hagamos inducción sobre i:

- Caso base: Cuando $i = 0$, ocurre que $\sigma(0) = s_0$, con $s_0 \in S_0$. Como s_0 también es estado inicial en $\Sigma_{M\times M}$, entonces $\sigma_m(0) = s_0$.
- Para el caso inductivo, asumamos que la propiedad vale para un natural k, es decir, $\forall i \cdot 0 \leq i \leq k \cdot \sigma(i) = \sigma_m(i)$. Sea $e = (\texttt{mV}, v_1, v_2) \in E^m$ el evento de entrada tal que $\sigma(k) \overset{e}{\rightsquigarrow} \sigma(k+1)$. Por hipótesis, como $\sigma(k) = \sigma_m(k)$, ocurre que tienen el mismo modo $MD(\sigma(k)) = MD(\sigma_m(k))$. Luego el evento de entrada de modo explícito $e_m = (MD(\sigma(k)), e, MD(\sigma(k+1)))$ está habilitado a ejecutarse en $\sigma_m(k)$, por lo que $\sigma_m(k) \overset{e_m}{\rightsquigarrow} \sigma_m(k+1)$. Como la función de transformación T definida por las tablas es determinista, debe ocurrir que $\sigma(k+1) = \sigma_m(k+1)$, finalizando la prueba.

□

Lema 5.3. *Para cada ejecución σ_m de $\Sigma_{M\times M} = (S, S_0, E_{M\times M}, T)$ existe una ejecución σ de $\Sigma = (S, S_0, E^m, T)$ tal que $\forall i \cdot 0 \leq i \cdot \sigma_m(i) = \sigma(i)$.*

Demostración. Hagamos inducción sobre i:

- Caso base: Cuando $i = 0$, ocurre que $\sigma_m(0) = s_0$, con $s_0 \in S_0$. Como s_0 también es un estado inicial en Σ, entonces $\sigma(0) = s_0$.
- Para el caso inductivo, asumamos que la propiedad vale para un natural k, es decir, $\forall i \cdot 0 \leq i \leq k \cdot \sigma_m(i) = \sigma(i)$. Sea $e_m = (m_k, (\texttt{mV}, v_1, v_2), m_{k+1}) \in E_{M\times M}$, con $m_k = MD(\sigma_m(k))$ y $m_{k+1} = MD(\sigma_m(k+1))$, el evento de entrada de modo preciso tal que $\sigma_(k) \overset{e_m}{\rightsquigarrow} \sigma_(k+1)$. Por hitótesis, como $\sigma_m(k) = \sigma(k)$, ocurre que tienen el mismo modo $m_k = MD(\sigma(k))$. Luego el evento de entrada $(\texttt{mV}, v_1, v_2)$ está habilitado a ejecutarse en $\sigma(k)$, por lo que $\sigma(k) \overset{(\texttt{mV},v_1,v_2)}{\rightsquigarrow} \sigma(k+1)$. Nuevamente, como la función de transformación T definida por las tablas es determinista, debe ocurrir que $\sigma_m(k+1) = \sigma(k+1)$, finalizando la prueba.

□

5.3.3 *Modularización de la Función de Transformación*

Modularización por Eventos

Note que, si sabemos cual variable monitoreada `mV` cambia en una transición, por la One-Input Assumption, también sabemos que las restantes variables

monitoreadas no pueden cambiar en esa transición. Además, sabemos que los eventos que no dependen del cambio de `mV` no pueden ocurrir (utilizando la relación de dependencia entre las variables del evento y `mV`, ver Subsección 3.5.1). Por ejemplo, en la Tabla de Transición de Modos del SIS (Tabla 6) todos los eventos dependen del cambio de `mWaterPres`, y cualquier cambio de otra variable monitoreada no puede disparar ninguno de los eventos de dicha tabla.

Así, para el conjunto de eventos de entrada que modifican la variable monitoreada `mV`, podemos computar una versión más simple $T_{\texttt{mV}}$ de la función de transformación T, removiendo las fórmulas en las Tablas de Modo y Eventos que no dependan de la variable `mV`. Esta nueva función de transformación $T_{\texttt{mV}}$ es equivalente a T con respecto a cualquier evento $e = (mV, \cdot, \cdot)$ (cualquier evento que modifica `mV`), es decir, $T_e(s, e) = T(s, e)$ para cualquier estado s y evento e.

Proponemos *modularizar* la función de transformación T de la especificación SCR explotando la observación anterior. La idea es automáticamente derivar desde T múltiples funciones de transformación $T_{e_1}, \ldots, T_{e_k}$: una por cada conjunto de eventos que modifican la misma variable monitoreada $\texttt{mV}_i$, con $i = 1 \ldots k$. Luego, cuando el evento $e_i = (\texttt{mV}_i, \cdot, \cdot)$ ocurre, podemos aplicar su correspondiente $T_{\texttt{mV}_i}$ para computar el nuevo estado.

La función de transformación modularizada es mucho más pequeña que la global, y puede ser codificada en fórmulas lógicas mucho más simples (ya que consideran menos eventos de entrada). Estos nos brindará numerosas ventajas a la hora de construir el modelo abstracto para una especificación SCR, tal como veremos en la siguiente sección.

Por ejemplo, para el caso del SIS, si el evento a considerar es $e = (\texttt{mReset}, \cdot, \cdot)$, las Tablas de Eventos y Transición de Modos resultantes son las siguientes:

$$
\begin{aligned}
&tOverridden' = \\
&F_e(mBlock, mReset, mcPressure, tOverridden, mBlock', mReset', mcPressure') = \\
&\quad \begin{cases} false & \text{if } (mcPressure = TooLow \wedge mReset' = On \wedge mReset = Off) \vee \\ & \quad\ (mcPressure = Permitted \wedge Reset' = On \wedge mReset = Off) \\ tOverridden & \text{otherwise} \end{cases}
\end{aligned}
$$

$$
\begin{aligned}
&mcPressure' = \\
&F_e(mWaterPres, mcPressure, mWaterPres') = \\
&\quad \{\ mcPressure \qquad true
\end{aligned}
$$

Note que como la Tabla de Transición de Modos del SIS (Tabla 6) no depende de la variable `mReset`, todas las filas pueden ser removidas en $T_{\texttt{mReset}}$,

y la variable `mcPressure` debe permanecer con el mismo valor de su estado corriente.

Modularización por Modos

Si utilizamos la semántica SCR de modo explítico, podemos alcanzar aún un grado más fino de modularización de la función de transformación. Para este caso, también vamos a utilizar el modo de origen y destino de cada evento de entrada de modo explícito para modularizar aún más la función de transformación T. Para cada par de modos m y m', vamos a obtener la función de transformación $T_{m,m'}$ removiendo desde T todos los eventos que no pueden ocurrir cuando el sistema parte de un estado en modo m a otro en modo m'. Similar a la modularización anterior, $T_{m,m'}$ puede ser aplicada para obtener un nuevo estado cuando ocurre un evento de entrada de modo explícito $e_m = (m, \cdot, m')$ (es decir, e_m modifica cualquier variable monitoreada, pero los modos de origen y destino deben ser m y m', respectivamente).

Sean m y m' el modo origen y destino, para obtener $T_{m,m'}$ debemos realizar las siguientes acciones sobre T:

1. Para las Tablas de Modo, remover todas las fórmulas de T representando filas que no pasan del modo m al m'.
2. Para las Tablas de Condición y Eventos, remover todas las fórmulas de T que codifican filas que no tienen como modo origen a m.
3. De cualquier tabla, remover todos los eventos que no comienzan en m, y no pasan al modo m'.

Por ejemplo, considere la Tabla de Condición del SIS (Tabla 2), asumiendo que el sistema está en modo `High` y permanece en ese modo. Luego de aplicar este paso de modularización, obtenemos $T_{High,High'}$ de la siguiente manera:

$$
\begin{aligned}
&cSafetyInjection = \\
&F_{cHigh,High'}(mcPressure, tOverridden) = \\
&\qquad \begin{cases} Off & \text{if } mcPressure = High \end{cases}
\end{aligned}
$$

En la Tabla de Eventos del SIS (Tabla 4) hay una sola fórmula con modo origen `High`, que consiste del evento `@F(mcPressure=High)`. En este caso, debido

a la regla 3, podemos remover esta fila ya que el modo destino no debe ser `High`. Finalmente, nos queda que `tOverridden` no puede cambiar de valor:

$$tOverridden' =$$
$$F_{eHigh,High'}(mBlock, mReset, mcPressure, tOverridden, mBlock', mReset', mcPressure') =$$
$$\begin{cases} tOverridden & \text{true} \end{cases}$$

Para la Tabla de Transición de Modos (Tabla 6), obviamente no hay filas describiendo que `mcPressure` permanece en el mismo modo, lo que nos queda:

$$mcPressure' =$$
$$F_{mHigh,High'}(mWaterPres, mcPressure, mWaterPres') =$$
$$\begin{cases} mcPressure & \text{true} \end{cases}$$

Es importante remarcar que los dos pasos de modularización pueden ser aplicados en conjunción sobre los eventos de entrada. Para cada evento de entrada de modo explícito $e = (m, (mV, \cdot, \cdot), m')$, primero aplicaremos modularización por eventos de entrada obteniendo $T_{\texttt{mV}}$, para luego aplicarle (a cada $T_{\texttt{mV}}$) la modularización sobre modos obteniendo $T_{m,\texttt{mV},m'}$, que serán utilizadas para presentar nuestro algoritmo de abstracción para SCR.

5.3.4 *Abstracción Lazy para SCR*

Una forma particular de abstraer el espacio de estados es mediante *Abstracción por Predicados* [Graf and Saïdi, 1997]. En este contexto, las regiones son caracterizadas por conjuntos de propiedades de estados llamadas *predicados de soporte*. Intuitivamente, un predicado P describe a un subconjunto de estados de S que satisfacen el predicado P, los cuales serán denotados como $[P]$. De esta manera, los estados abstractos pueden ser descriptos mediante conjunciones de predicados (o sus respectivas negaciones). Por ejemplo, $s_1^{\mathcal{A}} : P_1 \wedge \neg P_2$ y $s_2^{\mathcal{A}} : \neg P_2$ son estados abstractos válidos con predicados de soporte: P_1 y P_2; $s_1^{\mathcal{A}}$ representa estados concretos que satisfacen P_1 y $\neg P_2$, mientras que $s_2^{\mathcal{A}}$ representa estados que satisfacen $\neg P_2$ (pero pueden o no satisfacer P_1).

Nuestra técnica de abstracción considera como predicados de soporte, fórmulas en Aritmética Entera Lineal de Primer Orden (LIA). Esto significa que un predicado de soporte P es una fórmula tal que $P \in LIA$. Utilizamos esta lógica porque es *decidible* y existen varias herramientas que implementan procedimientos de desición para dicha lógica (nosotros utilizaremos MathSAT [Bruttomesso et al., 2008]), y es lo suficientemente expresiva para describir todas las especificaciones SCR que utilizaremos en este trabajo.

A diferencia de los primeros enfoques de abstracción por predicados donde los predicados eran considerados como globales [Graf and Saïdi, 1997], *Abstracción por Predicados Lazy* [Henzinger et al., 2002] permite que los predicados de soporte sean locales a regiones. En general, cuando se verifica una propiedad sobre programas, los predicados de soporte son útiles sólo en ciertas partes del modelo abstracto. Luego, la localidad de los predicados sobre regiones permite reducir el numero de invocaciones al procedimiento de decisión a la hora de generar el modelo abstracto [Henzinger et al., 2002].

En nuestro caso, hemos observamos que al verificar un propiedad sobre especificaciones SCR, los diferentes modos usualmente requieren diferentes conjuntos de predicados. Luego, el framework de abstracción Lazy fue una elección natural para utilizar como base de nuestro enfoque. De esta manera, nuestras regiones abstractas cuentan con una función $\Pi : M \mapsto 2^{LIA}$ que mapea cada modo de la especificación a un conjunto de predicados de soporte. Además, haremos que los modos sean concretos en nuestro algoritmo de abstracción, es decir, cada región tendrá un modo explícito asociado. Definamos a continuación nuestras regiones abstractas.

Definición 5.3 (Región atómica de "modo expliícito"). *Sea* $\Sigma_{M\times M} = (S, S_0, E_{M\times M}, T)$ *el LTS de modo explícito asociado a la especificación SCR. Una* región atómica de modo explícito *es un 3-upla* (m, φ, Π) *compuesta de un modo* $m \in M$*, una función que mapea modos a predicados de soporte* $\Pi : M \mapsto 2^{LIA}$*, y una fórmula* φ *sobre los predicados de* $\Pi(m)$*. La concretización de* (m, φ, Π) *es definida como* $[(m, \varphi, \Pi)] = [m \wedge \varphi]$*. Es decir,* $[(m, \varphi, \Pi)]$ *es el conjunto de estados con modo* m *que satisfacen* φ*. Para dos regiones atómicas de modo preciso* (m, φ, Π) *y* (m', φ', Π')*, definimos:*

- $(m, \varphi, \Pi) \sqcup (m', \varphi', \Pi') = (m, \varphi \vee \varphi', \lambda m.\Pi(m) \cup \Pi'(m))$ *si* $m = m'$*;* $\perp$ *en otro caso.*
- $(m, \varphi, \Pi) \sqcap (m', \varphi', \Pi') = (m, \varphi \wedge \varphi', \lambda m.\Pi(m) \cup \Pi'(m))$ *si* $m = m'$*;* $\perp$ *en otro caso.*
- *Sea* $e_m = (m, (mV, v, v'), m')$*,* $post((m, \varphi, \Pi), e_m) = (m', \varphi^{post}_{e_m}, \Pi_{ls})$ *donde* $\varphi^{post}_{e_m}$ *es la strongest postcondition de* φ *con respecto al evento* e_m*, y* $\Pi_{ls}(m')$ *es el menor conjunto de* $\Pi(m')$ *que contiene todos los predicados en* $\varphi^{post}_{e_m}$*. En otro caso, si* $e_m = (m'', \cdot, \cdot)$ *y* $m \neq m''$*, entonces* $post((m, \varphi, \Pi), e_m) = \perp$*.*

Note que podemos definir la strongest postcondition $\varphi_{e_m}^{post}$ usando nuestra semántica SCR de modo explícito con la siguiente fórmula LIA:

$$\begin{aligned}\varphi_{e_m}^{post} &= \varphi \wedge e_m \wedge T_{m,mV,m'} \\ &= \varphi \wedge (m \wedge \texttt{mV} = v \wedge m' \wedge \texttt{mV} = v') \wedge T_{m,mV,m'}\end{aligned}$$

Ahora podemos definir el dominio abstracto que usará nuestro algoritmo de abstracción: "conjuntos de regiones atómicas de modo explícito".

Definición 5.4 (Estructura de Regiones de Modo Explícito)**.** *Sea A el conjunto de regiones atómicas de modo explícito, una* estructura de regiones de modo explícito $\mathcal{R} = (R, \bot, \sqcup, \sqcap, post, [.])$ *para un LTS $\Sigma_{M\times M}$ consiste de :*

- *$R \subseteq 2^A$ un conjunto de regiones de modo explícito. Cada región $r \in R$ será un conjunto de regiones atómicas de modo explícito.*
- *Una función de concretización $[.]$ para $r \in R$, definida como $[r] = \bigcup_{a\in r}[a]$.*
- *Para $r, r' \in R$, los elementos restantes de $\mathcal{R}$ son definidos como:*
  - $r \sqcup r' = r \cup r' \cup \{a \sqcup a' | a \in r \text{ y } a' \in r'\}$,
  - $r \sqcap r' = \{a \sqcap a' | a \in r \text{ y } a' \in r'\}$,
  - *$r \sqsubseteq r'$ si para todo $a \in r$ existe $a'_1, \ldots, a'_k \in r'$ tales que $a \sqsubseteq a'_1 \sqcup \ldots \sqcup a'_k$,*
  - $post(r, e_m) = \bigcup_{a\in r} post(a, e_m)$.

Note que nuestros estados abstractos se ajustan exactamente con la noción de regiones de [Jhala, 1999], cambiando el concepto de program counters (de un lenguaje imperativo) por los modos de una especificación SCR, y eliminando la pila de invocaciones, ya que en SCR no hay invocaciones a métodos. Por lo tanto, aquí vamos a proveer las definiciones necesarias para dicho framework de abstracción, y referimos al lector a [Jhala, 1999] para sus pruebas de consistencia.

Nuestro algoritmo utilizará una función estándar de abstracción para abstracción por predicados, similar a la introducida en [Graf and Saïdi, 1997].

Definición 5.5 (Función de Abstracción)**.** *La función de abstracción $\alpha : LIA \times 2^{LIA} \mapsto LIA$ toma una fórmula φ y un conjunto de predicados $P \subseteq LIA$, y retorna una fórmula de LIA representando la abstracción por predicados de φ con respecto a P. α es definida como sigue:*

$$\alpha(\varphi, P) = \bigwedge_{p\in P} \{p \mid \varphi \to p\} \wedge \bigwedge_{p\in P} \{\neg p \mid \varphi \to \neg p\}$$

Así, $\alpha(\varphi, P)$ es una conjunción de predicados y negaciones de predicados tomados de P. Nosotros utilizamos un SMT Solver para LIA para decidir cuando un predicado $p \in P$ o su negación, son implicados por φ (note que puede que ninguno de los dos casos valga, y en ese caso el predicado p no formará parte de $\alpha(\varphi, P)$). Se conoce que para todo P y φ, una función de abstracción por predicados α como la introducida produce una sobre-aproximación del conjunto de estados descripto por φ, es decir, $[\varphi] \subseteq [\alpha(\varphi, P)]$.

Ahora estamos listos para introducir la estructura de abstracción que será utilizada como base de nuestro algoritmo.

Definición 5.6 (Estructura de Abstracción de Modo Explícito)**.** *Una* estructura de abstracción de modo explícito *es una 3-uple* $A = (\mathcal{R}, post^{\mathcal{A}}, \trianglelefteq)$*, donde:*

- $\mathcal{R}$ *es una estructura de regiones de modo explícito,*
- *El operador abstracto de strongest postcondition para* $r \in R, e \in E_{M\times M}$*, es* $post^{\mathcal{A}}(r, e) = \bigcup_{a\in r} post^{\mathcal{A}}(a, e_m)$*, donde* $post^{\mathcal{A}}$ *para una región atómica* a *y evento de entrada* e_m *de modo explícito es:*
  - $post^{\mathcal{A}}((m, \varphi, \Pi), e_m) = (m', \alpha(\varphi^{post}_{e_m}, \Pi(m')), \Pi)$ *si* $e_m = (m, \cdot, m')$.
  - $post^{\mathcal{A}}((m, \varphi, \Pi), e_m) = \bot$ *si* $e_m = (m'', \cdot, \cdot)$ *con* $m \neq m''$.
- *Para una región* r *definimos* $r.\Pi = \bigcup_{(\cdot,\cdot,\Pi)\in r} \Pi$*. Luego, sean* $r_1, r_2 \in R$ *y* $\Pi_1 = r_1.\Pi$ *y* $\Pi_2 = r_2.\Pi$*, el preorden de presición es definido como* $r_1 \trianglelefteq r_2$*, si y sólo si,* $\forall m \in M \cdot \Pi_2(m) \subseteq \Pi_1(m)$.

A continuación, probamos que, usando nuestra semántica SCR de modo explícito, $post^{\mathcal{A}}$ es una sobreaproximación de $post$.

Teorema 5.4. $post^{\mathcal{A}}$ *es una sobreaproximación de post, es decir, para todo* $r \in R, e_m \in E_{M\times M}, [post(r, e_m)] \subseteq [post^{\mathcal{A}}(r, e_m)]$.

Demostración. Si probamos que $[post(a, e_m)] \subseteq [post^{\mathcal{A}}(a, e_m)]$ para toda región atómica de modo explícito $a \in r$, entonces por la definición de $[.]$ para regiones de modo explícito ($[r] = \bigcup_{a\in r}[a]$), tenemos que $[post(r, e_m)] \subseteq [post^{\mathcal{A}}(r, e_m)]$. Probemos entonces que $[post(a, e_m)] \subseteq [post^{\mathcal{A}}(a, e_m)]$.

Sea $a = (m, \varphi, \Pi)$. Cuando $e_m = (m'', \cdot, \cdot)$ y $m \neq m''$, por definición tenemos $post(a, e_m) = \bot$, $post^{\mathcal{A}}(a, e_m) = \bot$, y así $[post(a, e_m)] \subseteq [post^{\mathcal{A}}(a, e_m)]$.

En otro caso, sea $e_m = (m, \cdot, m')$. Luego,

$$[post(a, e_m)] = [(m', \varphi^{post}_{e_m}, \Pi_{ls})] = [m' \wedge \varphi^{post}_{e_m}]$$

y

$$[post^{\mathcal{A}}(a, e_m)] = [(m', \alpha(\varphi_{e_m}^{post}, \Pi(m')), \Pi)] = [m' \wedge \alpha(\varphi_{e_m}^{post}, \Pi(m'))]$$

Como α es una función de abstracción ocurre que $[\varphi_{e_m}^{post}] \subseteq [\alpha(\varphi_{e_m}^{post}, \Pi(m'))]$. Luego, se deduce que $[m' \wedge \varphi_{e_m}^{post}] \subseteq [m' \wedge \alpha(\varphi_{e_m}^{post}, \Pi(m'))]$, quedando completa la prueba. □

5.3.5 *El Algoritmo*

Nuestro algoritmo de abstracción Lazy es dado en Algoritmo 5.1. Es un algoritmo simbólico de búsqueda hacia adelante (symbolic forward search algorithm), basado en la estructura de abstracción de modo explícito de la Definición 5.6, con la capacidad de refinar (de manera Lazy) las regiones abstractas, para remover contraejemplos espurios. Este algoritmo funciona de manera similar al algoritmo de Lazy Abstraction para programas imperativos presentado en [Henzinger et al., 2002], el cuál construye un árbol de alcanzabilidad abstracto y trata de verificar que todo estado abstracto en el árbol satisface la propiedad φ_{prop}. Cuando termina, dos casos son posibles. Por un lado, retorna la región que describe el conjunto de estados abstractos alcanzables (y una sobre-aproximación del conjunto de estados concretos alcanzables) cuando la propiedad dada φ_{prop} es válida (línea 36). Por otro lado, retorna una traza concreta t evidenciando una violación a la propiedad φ_{prop} (línea 15).

En el algoritmo, la función Π que mapea modos a predicados de soporte es considerada como una variable global. Además, asumimos que el operador abstracto $post^{\mathcal{A}}$ crea regiones utilizando la variable global Π. Al comienzo del algoritmo (línea 2) Π comienza con sólo un predicado en cada modo: la propiedad a ser verificada φ_{prop}. Luego, Π crece monótonamente durante la ejecución: los predicados de soporte son agregados a Π cada vez que un nuevo contraejemplo espurio aparece.

Llamemos *init* al predicado que describe el estado inicial de la especificación SCR, y m_{init} al modo del estado inicial. Luego, en las líneas 3 y 4, creamos el nodo raíz r del árbol abstracto de alcanzabilidad con la región r_0, donde r_0 contiene sólo una región atómica con modo m_{init}, el predicado que se obtiene de abstraer *init* con Π ($\alpha(init, \Pi(m_{init}))$ y el mapping (global) de predicados Π.

Después de la inicialización descripta, el algoritmo comienza un proceso iterativo en la línea 5. En este proceso, cada nodo puede estar en alguno de los dos estados: *no marcado* (cuando sus sucesores abstractos aún no fueron computados), y *marcados* en otro caso. Además, los nodos marcados son distinguidos entre los *cubiertos* y los *no-cubiertos*. Los nodos cubiertos son hojas en el árbol abstracto

Algorithm 5.1 SCR-LazyAbs

1: **function** SCR-LazyAbs
2: $\Pi = \lambda m.\{\varphi_{prop}\}$, para todo $m \in M$
3: $r_0 = \{(m_{init}, \alpha(init, \Pi(m_{init}), \Pi)\}$
4: crear nodo raíz $r : r_0$
5: **while** haya nodos sin marcar **do**
6: tomar un nodo sin marcar $n : r_n$
7: **if** para todo $a \in r_n$, con $a = (\cdot, \varphi_n, \cdot)$, ocurre que $\varphi_{prop} \notin \varphi_n$ **then**
8: $--$ la propiedad puede no valer en el nodo n
9: $--$ $\sigma_{\mathcal{A}}$ es la traza abstracta de error (desde r hasta n)
10: $\sigma_{\mathcal{A}} = r : r_0 \xrightarrow{\sigma} n : r_n$
11: $--$ chequear si existe una traza concreta t en la concretización de $\sigma_{\mathcal{A}}$
12: $(feasible, t, \Pi') = solve(\sigma_{\mathcal{A}})$
13: **if** $feasible$ **then**
14: $--$ $\sigma_{\mathcal{A}}$ contiene una traza t concreta de error
15: **return** traza de error t
16: **else**
17: $--$ $\sigma_{\mathcal{A}}$ es una traza abstracta espuria
18: $--$ $\Pi' : M \to 2^{LIA}$ son los interpolantes que hacen espuria a $\sigma_{\mathcal{A}}$
19: Sea $n' : r'_n$ el último ancestro de n en $\sigma_{\mathcal{A}}$, tal que, $\Pi'(m'_n) \neq \emptyset$
20: para todo $a = (m'_n, \cdot, \cdot), a \in r'_n$
21: Sea $n'' : r''_n$ el predecesor inmediato de $n' : r'_n$ en $\sigma_{\mathcal{A}}$
22: $--$ Refinamos el árbol abstracto de alcanzabilidad a partir de n''
23: tirar el sub-árbol a partir de n''
24: desmarcar n''
25: $--$ actualizar Π para incluir los interpolantes descubiertos de Π'
26: $\Pi = \lambda m.(\Pi(m) \cup \Pi'(m))$
27: **else if** $r_n \sqsubseteq r'_n$ para algún nodo marcado no-cubierto $n' : r'_n$ **then**
28: $--$ n es cubierto por n'
29: marcar n como cubierto
30: **else**
31: $--$ construir los sucesores abstractos de n
32: **for** cada $e_m \in E_{M \times M}$ **do**
33: $r'_n = post^{\mathcal{A}}(r_n, e_m)$
34: **if** $r'_n \neq \bot$ **then**
35: construir un nodo $n' : r'_n$ y crear un arco $n : r_n \xrightarrow{e_m} n' : r'_n$
36: marcar n como no-cubierto
37: **return** la región $\bigsqcup\{r_u | u : r_u$ es un nodo marcado nocubierto$\}$

de alcanzabilidad, y corresponde a nodos cuyas regiones abstractas ya fueron visitadas previamente en una parte del árbol previamente computada. Por otro lado, los nodos no-cubiertos son aquellos para los cuales sus sucesores abstractos fueron expandidos usando el operador abstracto de strongest postcondition $post^{\mathcal{A}}(r_n, e_m)$.

En cada paso del ciclo, el algoritmo toma un nodo no marcado $n : r_n$ (línea 6), y decide sobre uno de los siguientes tres casos posibles disjuntos, dependiendo de las características de n.

Primero, si al menos una región atómica en r_n no tiene al predicado φ_{prop} en forma positiva (no es necesario que aparezca negado), puede ser el caso de que la propiedad φ_{prop} no valga. Así, el contraejemplo abstracto $\sigma_{\mathcal{A}}$, que comienza en el nodo raíz y finaliza en el nodo n (línea 10), debe ser examinado para decidir si es posible construir un contraejemplo concreto a partir de $\sigma_{\mathcal{A}}$. La función *solve*, en línea 12, es usada para chequear la factibilidad de $\sigma_{\mathcal{A}}$. Dicha función retorna una tupla $(feasible, t, \Pi')$, donde $feasible$ es un valor booleano indicando si existe una traza concreta contenida en $\sigma_{\mathcal{A}}$. Si ese es el caso, entonces t almacena dicha traza concreta que viola la propiedad φ_{prop}, y es retornada en la línea 15 para mostrar dicha violación al usuario. En caso de que $feasible$ sea falso, el valor de t es indefinido ya que $\sigma_{\mathcal{A}}$ es un contraejemplo espurio, y un conjunto de predicados de soporte para cada modo es computado en Π', usando un método basado en interpolación descripto a continuación en la Subsección 5.3.6. Los predicados en Π' son útiles para remover las traza espurias como $\sigma_{\mathcal{A}}$. En estos casos, el algoritmo busca el último ancestro de n en $\sigma_{\mathcal{A}}$ que no necesite ser refinado, digamos n', ya que ningún predicado fué descubierto para los modos de los nodos en el camino desde la raíz hasta dicho nodo n' en la traza $\sigma_{\mathcal{A}}$ (línea 19-20). Todo el sub-árbol del árbol abstracto de alcanzabilidad que comienza en n'', el predecesor inmediato de n' en $\sigma_{\mathcal{A}}$, debe ser eliminado (línea 23) ya que debe ser recomputado utilizando los nuevos predicados de Π' recién descubiertos. Luego, el nodo n'' es desmarcado (línea 24), y Π es actualizado para incluir los nuevos predicados de Π' (línea 26). Note que los predicados son agregados localmente a los modos en los cuales fueron descubiertos, similar a Lazy Abstraction, donde agregan los predicados localmente a los program counters correspondientes [Henzinger et al., 2002].

Segundo, si la región r_n del nodo n está contenida en otra región r'_n, para algún nodo marcado no-cubierto n', podemos dejar de expandir n ya que sus sucesores ya han sido previamente computados cuando n' fué expandido (línea 27). En este caso, el algoritmo marca al nodo n como cubierto (línea 29) y procede a la siguiente iteración.

Tercero, si ninguno de los dos casos anteriores aplica (n no viola φ_{prop} y n no esta cubierto por otro nodo), entonces se computan los sucesores abstractos de la región r_n, con respecto a $post^A$ y todos los eventos de entrada de modo explícito. Se crean nuevos nodos no marcados que almacenan las nuevas regiones, y finalmente n es marcado como no-cubierto (línea 32-36).

Para concluir esta sección, vamos a discutir a continuación, el procedimiento *solve*, utilizado para descubrir nuevos predicados de abstracción de manera automática.

5.3.6 *Refinamiento basado en Interpolación*

[Henzinger et al., 2004] propone un enfoque para refinar abstracciones basado en *interpolación* [McMillan, 2003]. El procedimiento *solve* del Algoritmo 5.1 sigue este enfoque para refinar las regiones abstractas durante el computo del árbol abstracto de alcanzabilidad. Aquí, presentamos la idea básica de refinamiento de abstracciones mediante interpolación, para más detalles el lector puede referirse a [Henzinger et al., 2004].

Definición 5.7 (Interpolación sobre Trazas)**.** *Sea $\sigma = \varphi_1, \varphi_2, \ldots, \varphi_k$ una secuencia de fórmulas tales que su conjunción es inconsistente ($\varphi_1 \wedge \varphi_2 \wedge \cdots \wedge \varphi_k = false$). Un* interpolante *para σ es una secuencia de fórmulas $\psi_0, \psi_1, \ldots, \psi_k$ tales que:*

(a) $\psi_0 = True$ *and* $\psi_k = False$*,*

(b) $\forall i \cdot 1 \leq i \leq k \cdot \psi_{i-1} \wedge \varphi_i \Rightarrow \psi_i$*, and*

(c) $\forall i \cdot 1 \leq i \leq k \cdot vars(\psi_i) \subseteq vars(\varphi_1, \ldots, \varphi_i) \cap vars(\varphi_{i+1}, \ldots, \varphi_k)$*.*

Para las fórmulas en LIA $\varphi_1, \ldots, \varphi_k$, los interpolantes $\psi_0, \ldots, \psi_k$ siempre existen, y pueden ser computados eficientemente desde la prueba de insatisfactibilidad de la conjunción de σ. Si σ representa una traza abstracta espuria, los interpolantes proveen predicados adecuados para agregar a cada región de la traza, así de esta manera σ es removida del comportamiento abstracto del sistema. Veamos esto en más detalle.

Sea $\sigma_{\mathcal{A}}$ una traza abstracta espuria tomada como entrada por *solve* en línea 12 del Algoritmo 5.1. $\sigma_{\mathcal{A}}$ es una secuencia de nodos $n_1 : r_1, \ldots, n_k : r_k$ y eventos de entrada de modo explícito $e_1, \ldots, e_{k-1}$, tales que $post^{\mathcal{A}}(r_i, e_i) = r_{i+1}$ para $i = 1 \ldots k-1$. Debido a la estructura de abstracción que utiliza nuestro algoritmo, ocurre que cada $r_i = \{(m_i, \varphi_i, \Pi)\}$. Así, para computar un interpolante y remover $\sigma_{\mathcal{A}}$, *solve* genera la siguiente secuencia de fórmulas en LIA: $\sigma_c =$

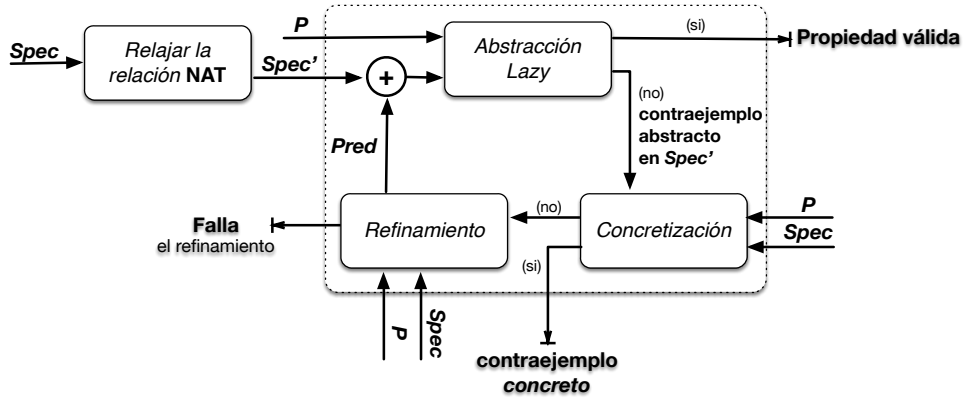

Figura 36: Resumen de nuestra técnica iterativa.

$m_1 \wedge \varphi_1, m_2 \wedge \varphi_2, \ldots, m_k \wedge \varphi_k$. Luego, *solve* invoca a un procedimiento de desición capaz de generar interpolantes para σ_c, lo que nos lleva a las fórmulas $\psi_0, \ldots, \psi_k$. Note que, por la Definición 5.7, cada ψ_{i-1} $(1 \leq i \leq k)$ es relevante sólo en el modo m_i. Así, *solve* construye el mapping Π' agregando cada ψ_{i-1} al conjunto de $\Pi'(m_i)$.

En nuestra implementación, utilizamos el SMT Solver MathSAT [Bruttomesso et al., 2008] para computar automáticamente estos interpolantes.

5.4 LA TÉCNICA: ANÁLISIS BASADO EN ABSTRACCIÓN PARA ESPECIFICACIONES TABULARES SCR

En esta sección describiremos en detalle nuestra técnica para analizar eficientemente requisitos de software capturados mediante especificaciones tabulares SCR. El enfoque consiste de un procedimiento de *refinamiento de abstracción guiado por contraejemplo* (CEGAR) usado sobre el algoritmo de abstracción Lazy recién presentado en la sección anterior. La Figura 36 muestra los pasos principales de nuestra técnica. Esta toma como entrada una especificación SCR *Spec* y una propiedad (de safety) P, y retorna si P es válida sobre *Spec* o no, mostrando una ejecución concreta de *Spec* que viola P en el último caso. Básicamente, nuestra técnica funciona de la siguiente manera.

Primero, construye una especificación abstracta *Spec'* relajando las restricciones de NAT sobre las variables monitoreadas numéricas de *Spec*. Las variables numéricas de *Spec'* tienen permitido cambiar arbitrariamente dentro de su dominio. Como ilustramos en la Sección 5.2, esto hace que *Spec'* sea más fácil

de analizar que *Spec* cuando utilizamos nuestro algoritmo de abstracción Lazy presentado en la sección anterior.

El siguiente paso de nuestra técnica es invocar a nuestro algoritmo de *Abstracción Lazy* con la especificación relajada *Spec'* y la propiedad P. Si el algoritmo prueba que P es válida sobre *Spec'*, luego P también es válida en *Spec* (debido a que *Spec'* es una abstracción conservativa de *Spec*), y el análisis finaliza. En otro caso, nuestro procedimiento de abstracción Lazy retorna un contraejemplo abstracto σ de *Spec'*, que puede o no representar una ejecución válida violando P en la especificación original *Spec* (ya que puede ocurrir que σ modifica variables numéricas que no son posibles de realizar en la especificación *Spec* con la relación NAT original).

Dado un contraejemplo abstracto σ, el enfoque tiene que averiguar si σ tiene una contraparte concreta (es decir, una violación conreta en *Spec*), o si es sólo una ejecución espuria introducida por relajar la relación NAT. El paso de *Concretización* del enfoque codifica la concretización de σ en el lenguaje de un model checker, y utiliza una herramienta de model checking para explorar el espacio de estados de σ en búsqueda de un contraejemplo concreto de *Spec*. Si dicho contraejemplo concreto existe, luego una violación de P en *Spec* ha sido encontrada, y el enfoque finaliza. En otro caso, el paso de *Refinamiento* del enfoque extrae un nuevo predicado *Pred* desde σ, el cuál es agregado a *Spec'* para remover el contraejemplo abstracto espurio σ.

A partir de este punto, el proceso es repetido hasta que prueba que P vale en *Spec* o hasta que encuentra un traza concreta de *Spec* que viola la propiedad P. Note que, sin embargo, el paso de refinamiento puede fallar debido a su incompletitud (ver Subsección 5.4.2)

En las siguientes secciones, describimos cada paso del enfoque mostrado en la Figura 36 en mayor detalle (con la excepción del algoritmo de abstracción Lazy, que ya fue presentado en la Sección 5.3). La Subsección 5.4.1 explica como se realiza el paso de *Relajar la relación NAT* para construir la especificación abstracta *Spec'*. La Subsección 5.4.2 discute como los contraejemplos abstractos de *Spec'* son concretizados y analizados via model checking. Finalmente, la Subsección 5.4.3 explica como nuestro enfoque extrae predicados desde el contraejemplo espurio de *Spec'*, y como este predicado es usado para refinar la especificación *Spec'*.

5.4.1 *Relajar la Relación* ***NAT***

Usualmente, las variables numéricas de las especificaciones de requerimientos están habilitadas a cambiar en pequeños pasos dentro de su domino [Bultan and Heitmeyer, 2008; Heitmeyer et al., 2005; Fraser and Gargantini, 2009]. Esta es normalmente una asunción que tiene que ver con la frecuencia con la que el sistema monitorea un valor. Este tipo de restricciones sobre las variables monitoreadas, forman parte normalmente de la relación NAT de una especificación SCR (ver Sección 3.5). Por ejemplo, la relación NAT del SIS impone que la variable monitoreada `mWaterPres` puede cambiar en a lo sumo 10 unidades en cada paso. Formalmente, la relación de transición para la variable `mWaterPres` puede ser descripta por el conjunto $\gamma_{\texttt{mWaterPres}}$ de abajo:

$$\gamma_{\texttt{mWaterPres}} = \{(x, x') : 0 \leq |x - x'| \leq 10 \wedge 0 \leq x \leq 5000 \wedge 0 \leq x' \leq 5000\} \quad (5)$$

donde la restricción $0 \leq |x - x'| \leq 10$ se debe a NAT, y $0 \leq x \leq 5000 \wedge 0 \leq x' \leq 5000$ corresponde al dominio de la variable `mWaterPres`.

En general, la relación de transición para una variable monitoreada numérica `mV` es:

$$\gamma_{mV} = \{(x, x') : \mathbf{NAT}(x, x') \wedge TY(x) \wedge TY(x')\} \quad (6)$$

donde $\mathbf{NAT}(x, x')$ son las restricciones impuestas por NAT y $TY(x), TY(x')$ aseguran que x y x' pueden tomar sólo valores dentro de su dominio.

Como explicamos en la sección anterior, el primer paso de nuestro enfoque consiste en relajar la relación NAT de la especificación SCR *Spec*, removiendo las variaciones permitidas sobre las variables numéricas. Así, la especificación resultante *Spec'* permite a las variable numéricas cambiar arbitrariamente dentro de su dominio. Lo que hace nuestro enfoque es remover $\mathbf{NAT}(x, x')$ desde γ_{mV}, llevandonos a la siguiente relación de transición para una variable numérica `mV`:

$$\gamma'_{mV} = \{(x, x') : TY(x) \wedge TY(x')\} \quad (7)$$

γ'_{mV} es la relación de transición de `mV` en *Spec'*. Nuestra técnica aplica esta idea a cada variable monitoreada numérica `mV` de *Spec* para producir *Spec'*.

Claramente, *Spec'* permite todos los comportamientos de *Spec* y posiblemente más. Luego, los estados alcanzables de *Spec* son un subconjunto de los estados alcanzables de *Spec'*. Este hecho es establecido en el siguiente teorema, cuya validez es obviamente válida por las razones ya mencionadas.

Teorema 5.5. *Sea* Σ, Σ' *los LTSs asociados a las especificaciones SCR Spec y Spec', respectivamente. Luego,* $Reach(\Sigma) \subseteq Reach(\Sigma')$.

Luego, *Spec'* es una sobreaproximación correcta de *Spec* en el sentido que cualquier propiedad de safety que valga en *Spec'* también será válida en *Spec*. Así, podemos utilizar *Spec'* para probar propiedades de safety usando nuestro algoritmo de abstracción Lazy para SCR de manera eficiente, y extrapolar los resultados a la especificación original *Spec*.

Volviendo al ejemplo del SIS, aplicando el procedimiento anterior a la equación 5, produce:

$$\gamma'_{\texttt{mWaterPres}} = \{(x, x') : 0 \leq x \leq 5000 \wedge 0 \leq x' \leq 5000\} \tag{8}$$

permitiendo a `mWaterPres` en *SIS'* cambiar arbitrariamente.

5.4.2 *Fase de* Concretización

Luego de construir la especificación *Spec'* en el paso anterior, invocamos a nuestro algoritmo de abstracción Lazy (presentado en la Sección 5.3) para verificar la validez de P. En caso de que valga, la técnica finaliza verificando la propiedad, en caso de que no valga, el algoritmo va a retornar una traza como contraejemplo abstracto σ' para la propiedad P en *Spec'*. Esta traza σ' tiene la siguiente forma:

$$\sigma' = s_0\ e_0\ s_1\ e_1 \ldots e_{m-1}\ s_m \tag{9}$$

donde cada $s_i = (m_i, \varphi_i, \Pi_i)$ es una región atómica, cada e_i es un evento de entrada de modo explícito de *Spec'*, y P no aparece positivamente en φ_m (de otra manera, σ' no sería un contraejemplo en *Spec'*).

El paso de *Concretización* de nuestro enfoque (ver Figura 36) explora el espacio de estados del contraejemplo σ', tratando de averiguar si σ' codifica una ejecución concreta de *Spec* que viole P. Si este es el caso, el enfoque termina y retorna dicha traza concreta de error al usuario. En otro caso, σ' es un contraejemplo espurio debido a la abstracción realizada sobre *Spec* que nos condujo a *Spec'*. Si ese es el caso, entonces para seguir verificando P sobre *Spec*, el paso de *Refinamiento* de nuestro enfoque tratará de aprender un predicado tal que remueva σ' desde *Spec'*. Este proceso de refinamiento es explicado en la Subsección 5.4.3. Ahora vamos a discutir como nuestra técnica decide si σ' codifica un contraejemplo concreto de P o es espurio.

Note que en σ' hay dos fuentes de abstracción respecto a *Spec*: los estados s_i, y los eventos de entrada de modo explícito e_i que corresponden a cambios arbitrarios de las variables monitoreadas numéricas debido a la relajación de NAT en *Spec'*. Para cada evento e_i modificando una variable numérica `mV` en

Spec', hemos observado que podemos "simular" el efecto de este evento repitiendo varias ejecuciones de e_i en *Spec* (tantas repeticiones como sean necesarias para mover `mV` de cualquier valor a cualquier otro, dentro de su dominio). Denotemos con e_i* a la ejecución repetida del evento e_i como describimos anteriormente. Luego, asumamos que, si e_i no modifica una variable numérica, entonces $e_i* = e_i$. La concretización de σ', denotada como $[\sigma']$, mostrada a continuación, describe el conjunto de ejecuciones de *Spec* representadas por σ':

$$[\sigma'] = [s_0]\ e_0 *\ [s_1]\ e_1 * \ldots e_{m-1} *\ [s_m] \tag{10}$$

donde $[s_i]$ es la concretización de regiones atómicas de modo explícito.

Como mencionamos en la Sección 5.2, podemos pensar a $[\sigma']$ como un "fragmento" de la especificación completa *Spec*, tal que: *(i)* involucra estados cuyos modos coinciden con $s_i = (m_i, \varphi_i, \Pi_i)$, (ii) sólo se va a ejecutar el subconjunto de eventos $e_0, \ldots, e_{m-1}$, y (iii) el orden en el cuál estos eventos se ejecutan es fijo (e_0 es ejecutado primero, luego e_1, y así sucesivamente). De esta manera, explorar el espacio de estados de $[\sigma']$ debería ser más simple que explorar el espacio completo de *Spec* (los resultados experimentales en la Sección 5.6 proveen evidencia para esta afirmación).

Note que $[\sigma']$ puede ser pensado como una especificación SCR por si misma, con un conjunto reducido de eventos (con respecto a *Spec*), y un orden fijo para ejecutarlos. Luego, podemos realizar una exploración exhaustiva del espacio de estados de $[\sigma']$ (buscando una violación a P) usando herramientas de model checking, explotando tranducciones ya existentes de SCR al lenguaje del model checker. Para esta parte de nuestro enfoque, utilizamos el model checker Spin [Holzmann, 2004], el cuál en nuestros experimentos ha mostrado un gran desempeño para explorar el espacio de estados de $[\sigma']$ (a pesar de que a veces falla al analizar la especificación completa).

Codificación en Promela del Contraejemplo Abstracto

Para traducir $[\sigma']$ a PROMELA (ver Subsección 2.3.2) utilizamos el método basado en [Bharadwaj and Heitmeyer, 1999]. En líneas generales, dada la especificación SCR *Spec*, la técnica de [Bharadwaj and Heitmeyer, 1999] produce una especificación PROMELA que tiene la siguiente estructura:

```
1 /* Variables del Modelo */
2 do
3 :: /* un cambio no-determinista en una variable monitoreada */
4    /* definiciones de tablas */
5    /* actualizacion de variables */
6 od
```

En la primer parte (línea 1) se definen todas las variables del modelo. Como la traducción describe la transición de un estado al siguiente, para cada variable SCR se definen dos variables en Promela (por ejemplo, `mWaterPres` y `mWaterPresP`) para modelar su estado actual y el siguiente. La ejecución de la especificación SCR es modelada como un ciclo infinito (línea 2-6) donde primero ocurre no-deterministicamente un evento de entrada (línea 3), luego se propagan los cambios a las variables dependientes (modos, términos, controladas) acorde la definición de las tablas (línea 4), y finalmente se actualizan las variables Promela (línea 5). Los eventos de entrada se corresponden a un cambio no determinista de alguna variable monitoreada (esto se modela usando la elección no determinista que Promela provee). Cada tabla define una función total (ver Subsección 3.5.1), por lo que su traducción es casi directa utilizando sentencias `if` que Promela provee. Al final de cada iteración del ciclo, el estado nuevo pasará a ser el actual en la siguiente iteración. Es por esto, que al final del ciclo se actualizan las variables con su nuevo valor (por ejemplo, a `mWaterPres` le asignamos `mWaterPresP`). Puede encontrar los detalles de la traducción en [Bharadwaj and Heitmeyer, 1999]. La Figura 37 muestra una versión simple de la traducción a Promela de la especificación SCR del SIS de la Sección 3.5.

Nuestro objetivo aquí es tratar de encontrar una ejecución concreta en *Spec* para $[\sigma']$. Para esto, lo que vamos a hacer es alimentar las tablas con los eventos de entrada de σ', en vez de que cambie no-deterministicamente cualquier variable, siguiendo el orden en el que aparecen en σ'.

Sea $\sigma' = s_0\ e_0\ s_1\ e_1 \ldots e_{m-1}\ s_m$. Lo que haremos es definir una variable entera `pc` (program counter) que indica cuál es el siguiente evento de entrada de σ' que debemos ejecutar (replicando los cambios sobre las variables numéricas). Note que los eventos de entrada en σ' indican cuál es la variable monitoreada que cambia, pero no consideran la relación NAT original. Por lo que nuestra codificación utiliza `pc` para indicar qué variable monitoreada debe cambiar, pero el valor que dicha variable obtenga es no-determinista, acorde la relación original NAT lo indique (como muestra la Figura 37). De esta manera nos aseguramos que cualquier ejecución que Spin encuentre, será una ejecución concreta en *Spec*.

```
1  #define Low 900; Permit 4000; TooLow 0; Permitted 1; High 2
2  /* Variables del Modelo */
3  bool mBlock, mBlockP = Off; bool mReset, mResetP = On;
4  short waterPres, waterPresP = 14; byte mcPressure, mcPressureP = TooLow;
5  bool tOverridden, tOverriddenP = 0;bool cSafetyInjection, cSafetyInjectionP;
6  init{
7  do
8  :: atomic{
9  /* un cambio no-determinista en una variable monitoreada */
10 if
11 :: /* cambia mBlock */
12 :: /* cambia mReset */
13 :: if /* cambia waterPres acorde la variacion de NAT */
14   ::waterPres <= 4999 -> waterPresP = waterPres + 1
15   ::waterPres >= 1 -> waterPresP = waterPres - 1
16   :: ...
17   ::waterPres <= 4990 -> waterPresP = waterPres + 10
18   ::waterPres >= 10 -> waterPresP = waterPres - 10
19   fi;
20 :: else -> break
21 fi;
22 d_step{
23 /* definiciones de tablas */
24 /*  Tabla de transicion de modos mcPressure */
25 if :: mcPressure == TooLow ->
26        if
27        :: (waterPresP >= Low) && (!(waterPres >= Low)) -> mcPressureP = Permitted;
28        ::else skip;
29        fi;
30   :: mcPressure == Permitted ->
31        if
32        :: (waterPresP < Low) && (!(waterPres < Low)) -> mcPressureP = TooLow;
33        :: (waterPresP >= Permit) && (!(waterPres >= Permit)) -> mcPressureP = High;
34        ::else skip;
35        fi;
36   :: mcPressure == High ->
37        if
38        :: (waterPresP < Permit) && (!(waterPres < Permit)) -> mcPressureP = Permitted;
39        ::else skip;
40        fi;
41 fi;
42 /*  tabla de eventos tOverridden */
43 /*  tabla de condicion cSafetyInjection */
44 /* actualizacion de variables */
45 }/* end of d_step */
46 }/* end of atomic */
47 od  /*end of do*/
48 }
```

Figura 37: Especificación PROMELA para el SIS.

Basandonos en la codificación de [Bharadwaj and Heitmeyer, 1999], la especificación Promela que generamos para σ' tiene la siguiente estructura general:

```
1  /* Variables del Modelo */
2  int pc = 0; //program counter
3  int mVar;  //variable monitoreada a cambiar
4  do
5  ::
6     /* asignar a mVar el evento de entrada a ejecutar, segun el pc */
7     /* un cambio no-determinista en la variable monitoreada mVar */
8     /* definiciones de tablas */
9     /* actualizacion de variables */
10    /* actualizacion del valor de pc */
11    /* chequear si encontramos la ejecucion concreta */
12 od
```

Como puede observar, lo primero que hacemos es indicar en la variable `mVar` cuál es la variable monitoreada a modificar respecto del valor de `pc` (es decir, respecto a cuál es el siguiente evento de entrada a ejecutar en σ'). El cambio de la variable monitoreada que indica `mVar` es realizado de manera no determinista, respetando la relación NAT original (como pudo observar en la Figura 37). Luego, los cambios se propagan a las variables dependientes acorde las definiciones de las tablas, y las variables se actualizan como ya explicamos.

Lo siguiente que hacemos es actualizar el valor de `pc`, siempre y cuando el estado siguiente computado tenga el mismo modo de la siguiente región de σ'. Es decir, suponiendo que `pc` $= i$, antes de incrementar `pc` a $i+1$, primero vamos a chequear que el estado computado tiene el mismo modo de la región abstracta s_{i+1} de σ'. Note que cualquier ejecución que no cumple con esta condición no nos interesa, porque estamos buscando una ejecución concreta para σ' (podemos indicarle a Spin que descarte las traza no deseadas con el comando `break`). Para replicar los cambios sobre las variables monitoreadas numéricas, vamos a utilizar la elección no determinista de Spin. Es decir, cuando `pc` $= i$ e $e_i \in \sigma'$ es un evento de entrada que modifica una variable numérica, puede que `pc` quede con el mismo valor (o se incremente como explicamos antes), modelando de esta manera todas las posibles replicaciones del evento e_i (es decir, con esto modelamos los e_i* de $[\sigma']$ cuando cambian variables numéricas).

Finalmente, chequeamos si hemos encontrado o no una ejecución concreta para σ'. Para esto, utilizamos el comando `assert` de Spin codificando la siguiente expresión: `assert (pc=m -> !`$[s_m]$`)`. Si Spin encuentra una violación a este `assert`, quiere decir que encontró una ejecución que llegó hasta la última posición de σ' (`pc=m`) y el estado computado cae dentro de la última región $[s_m]$.

Por lo tanto, esa ejecución es una ejecución concreta para σ', y nuestro técnica finaliza retornando un contraejemplo concreto para la propiedad.

A modo de ejemplo, considere el contraejemplo retornado para el SIS en *Spec'* de la Figura 35. En este caso queremos ver si el modo `Permitted` es alcanzable. Luego de relajar NAT, el contraejemplo retornado fue el siguiente: $\sigma' =< \texttt{TooLow} > (\texttt{mWaterPres}, 14, 900) < \texttt{Permitted} >$. La Figura 38 muestra la codificación en Promela que realiza nuestra técnica de este contraejemplo. Spin tratará mediante la ejecución sucesiva de cambios en la variable `mWaterPres`, llegar al modo `Permitted` a partir del modo `TooLow`. Para este caso, Spin puede generar dicha ejecución en pocos segundos.

5.4.3 *Fase de* Refinamiento

Como mencionamos previamente, cuando una ejecución abstracta σ' de *Spec'* no representa ninguna ejecución concreta de *Spec*, primero debemos encontrar la causa de insatisfactibilidad de σ', para luego sintetizar un predicado que nos permita refinar *Spec'* y remover σ' desde su comportamiento.

Denotemos $\sigma'(..i) = s_0\ e_0\ s_1 \ldots s_{i-1}\ e_{i-1}\ s_i$ como el i-ésimo prefijo de σ', para algún $i \leq m$. Con el objetivo de detectar la causa de insatisfactibilidad de σ', nuestra técnica realiza los siguientes pasos.

Primero, para i desde 1 hasta m, chequeamos si existe al menos una traza concreta para $[\sigma'(..i)]$, usando Spin como explicamos en la sección anterior. Sea k, con $1 \leq k \leq m$, la *primer* posición detectada para la cuál el prefijo $[\sigma'(..k)]$ no posee una ejecución concreta. Como para $[\sigma'(..k-1)]$ si existe al menos un traza concreta, pero para $[\sigma'(..k)]$ no, vamos a considerar que el problema de la insatisfactibilidad de σ' posiblemente se encuentre en la transición $s_{k-1}\ e_{k-1}\ s_k$.

Luego, vamos a utilizar ALV, un model checker simbólico utilizado para analizar especificaciones SCR (ver Subsección 2.5.3), para chequear si dicha transición contiene o no alguna ejecución concreta en *Spec*. Es decir, debemos verificar que no es posible en *Spec*, partiendo de algún estado de $[s_{k-1}]$, ejecutar un número repetido de veces el evento e_{k-1}, alcanzar un estado en $[s_k]$. Note que no podemos utilizar a Spin para esta tarea, ya que el estado inicial debe satisfacer una fórmula que caracteriza $[s_{k-1}]$, mientras que ALV nos provee un mecanismo simple para hacerlo.

Finalmente, si no existe una traza concreta en *Spec* para $[s_{k-1}]e_{k-1} * [s_k]$, sintetizamos un predicado $Pred = [s_{k-1}] \wedge e_{k-1} \rightarrow \neg[s_k]$, el cuál asegura que la ejecución del evento e_{k-1} en *Spec'* comenzando en s_{k-1} no puede conducirnos a un estado representado por s_k. Agregar $Pred$ a *Spec'*, nos asegura que el

```
1  /* Variables del Modelo */
2  int pc = 0; //program counter
3  int mVar;  //variable monitoreada a cambiar
4  init{
5  do
6  :: atomic{
7  /* asignar a mVar el evento de entrada a ejecutar, segun el pc */
8  if
9  :: pc ==0 -> mVar = c_waterPres
10 ::else -> break
11 fi;
12 /* un cambio no-determinista en la variable monitoreada mVar */
13 if
14 :: (mVar==c_mBlock) ->
15    /* cambia mBlock */
16 :: (mVar == c_mReset) ->
17    /* cambia mReset */
18 :: (mVar == c_waterPres) ->
19   if /* cambia waterPres acorde la variacion de NAT */
20   ::waterPres <= 4999 -> waterPresP = waterPres + 1
21   ::waterPres >= 1 -> waterPresP = waterPres - 1
22   :: ...
23   ::waterPres <= 4990 -> waterPresP = waterPres + 10
24   ::waterPres >= 10 -> waterPresP = waterPres - 10
25   fi;
26 :: else -> break
27 fi;
28 d_step{
29 /* definiciones de tablas */
30 /*  Tabla de transicion de modos mcPressure */
31 /*  tabla de eventos tOverridden */
32 /*  tabla de condicion cSafetyInjection */
33 /* actualizacion de variables */
34 /* actualizacion del valor de pc */
35 if
36 :: pc ==0 -> skip
37 :: pc ==0 && (mcPressure==Permitted) -> pc++
38 fi;
39 /* chequear si encontramos la ejecucion concreta */
40 assert (pc=1 -> !(mcPressure==Permitted))
41 }/* end of d_step */
42 }/* end of atomic */
43 od  /*end of do*/
44 }
```

Figura 38: Especificación PROMELA para el Ejemplo Motivador del SIS.

mismo contraejemplo abstracto espurio σ' no volverá a aparecer, y el proceso de verificación puede comenzar nuevamente con la nueva *Spec'* refinada.

Por otro lado, si $[s_{k-1}]e_{k-1} * [s_k]$ tiene al menos una traza, nuestro método de refinamiento *falla* y no podemos remover el contraejemplo abstracto σ' del comportamiento de *Spec'*. En este caso, nuestro enfoque resulta ser incompleto.

Nuestra elección de utilizar ALV para esta tarea se debe a varias razones. ALV ha mostrado ser muy útil para verificar la validez de propiedades sobre especificaciones SCR [Bultan and Heitmeyer, 2006; Bultan and Heitmeyer, 2008]. Como nuestra técnica de refinamiento asume que la transición $s_{k-1}\ e_{k-1}\ s_k$ tiene grandes posibilidades de ser insatisfactible, ALV resulta ser la mejor opción para probarlo. Sin embargo, ALV presenta serios problemas de escalabilidad para generar los contraejemplos cuando la propiedad no es válida [Bultan and Heitmeyer, 2006; Bultan and Heitmeyer, 2008]. Es por esto que seteamos un timeout para el tiempo de análisis de ALV (1 minuto). Cumplido ese tiempo, si ALV no ha respondido, entonces el proceso de refinamiendo falla (aunque nuevamente nuestro método asume que estos serán muy pocos casos, como nuestros resultados experimentales lo muestran).

Note que la poca capacidad de ALV para generar contraejemplos de especificaciones SCR, no lo convierte en buen candidato para tareas como la generación de casos de tests. Luego, como veremos en la sección de resultados experimentales, ALV no será considerado entre los model checkers contra los cuales nos comparamos, para este tipo de tarea.

La traducción de una especificación SCR a Action Language (Subsección 2.5.3)), el lenguaje de ALV, puede encontrarse en [Bultan and Heitmeyer, 2006; Bultan and Heitmeyer, 2008].

Ejemplo de Refinamiento para el SIS

A modo de ejemplo, considere la siguiente situación del SIS, tomada de nuestra evaluación experimental, en la que necesitamos aplicar nuestra técnica de refinamiento. El contraejemplo a analizar en este caso es el siguiente:

$\sigma' =< \texttt{TooLow} > (\texttt{mWaterPres}, 14, 4000) < \texttt{Permitted} \wedge (\texttt{mWaterPres} >= \texttt{Permit}) >$

Note que σ' muestra una ejecución en la que SIS pasa del modo `TooLow` al modo `Permitted`, pero la variable entera `mWaterPres` es mayor que `Permit`. Es decir, el estado alcanzado no es posible en la especificación original *Spec* por varias razones: primero, si `mWaterPres>=Permit` entonces el modo debería ser `High`; y segundo, tampoco es posible alcanzar el `High` desde `TooLow`, sin pasar antes por `Permitted`.

Por las razones mencionadas, Spin no podrá generar una ejecución concreta para σ'. Luego, nuestra técnica buscará probar utilizando ALV, que no es posible a partir de `TooLow` y mediante los sucesivos cambios de la variable `mWaterPres`, alcanzar un estado que satisfaga (`Permitted` $\wedge$ `mWaterPres` $>=$ `Permit`). Para esto, la anterior transición es codificada en ACTION LANGUAGE como muestra la Figura 39. ALV puede probar en pocos segundos la insatisfactibilidad de dicha transición. Luego, nuestra técnica de refinamiento produce el siguiente predicado *Pred* con el que refinaremos *Spec'* para que esta misma violación espuria no vuelva a ocurrir:

$$Pred = (\texttt{mcPressure=TooLow} \wedge \texttt{mWaterPres}' \neq \texttt{mWaterPres} \rightarrow \neg(\texttt{mcPressure'=Permitted} \wedge \texttt{mWaterPres'} >= \texttt{Permit}))$$

5.4.4 *Análisis Iterativo*

Las fases de *Abstracción Lazy*, *Concretización* y *Refinamiento* de la Figura 36 se corresponden a una sola iteración de nuestro enfoque. Estas fases son iterativamente aplicadas hasta que no se detectan más violaciones en *Spec'* en la fase de *Abstracción Lazy*, lo que significa que la especificación SCR original *Spec* satisface la propiedad *P*, o hasta que la fase de *Concretización* logra generar un contraejemplo concreto en *Spec* violanto *P*. Además puede ocurrir que alcancemos un punto donde nuestro método de *Refinamiento* falle, y no seamos capaces de remover un contraejemplo espurio de *Spec'*, por lo que nuestra técnica será inconclusa en estos casos.

Discutamos acerca de la *correctitud* de nuestra técnica. A lo largo de la Sección 5.3 discutimos y demostramos que nuestro algoritmo de Abstracción Lazy es correcto. Además, el Teorema 5.1 asegura que, despúes de relajar la relación NAT, la especificación obtenida *Spec'* es una abstracción conservativa de la especificación original *Spec*. Luego, si nuestro algoritmo de abstracción verifica la validez de *P* sobre *Spec'*, por las razones mencionadas, la propiedad también será válida en *Spec*. Note además, que en caso de que *Spec'* sea refinada con algún predicado *Pred*, nuestro método de refinamiento asegura remover de *Spec'*, sólo transiciones que no son realizables en *Spec*. Por lo tanto, la refinada especificación *Spec'* sigue siendo una abstracción conservativa de *Spec*, luego de agregarle *Pred*.

Por otro lado, si nuestro algoritmo de abstracción retorna un contraejemplo abstracto sobre *Spec'*, la fase de Concretización tratará de reproducirlo sobre la especificación original *Spec*, considerando las variaciones originales de las variables numéricas. Luego, si Spin logra encontrar una ejecución concreta

```
MODULE main
VAR --variables del Modelo
DEFINE -- constantes del Modelo
-- traduccion para cSafetyInjection
cSafetyInjection := case
 mcPressure = Permitted | mcPressure = High: Off;
 mcPressure = TooLow: case tOverridden : Off;
                          ! tOverridden : On;
    esac;
esac;
ASSIGN
-- traduccion para mcPressure
next(mcPressure) :=
case mcPressure = TooLow: case
  (next(waterPres) >= Low) & (!(waterPres >= Low)) : Permitted;
  TRUE : mcPressure;
esac;
 mcPressure = Permitted: case
  (next(waterPres) < Low) & (!(waterPres < Low)) : TooLow;
  (next(waterPres) >= Permit) & (!(waterPres >= Permit)) : High;
  TRUE : mcPressure;
esac;
 mcPressure = High: case
  (next(waterPres) < Permit) & (!(waterPres < Permit)) : Permitted;
  TRUE : mcPressure;
esac;
esac;
-- traduccion para tOverridden
next(tOverridden) :=
case (mcPressure = TooLow | mcPressure = Permitted) :
  case
  ((next(mcPressure) = High) & (!(mcPressure = High))) : FALSE;
  ((next(mReset) = On) & (!(mReset = On))) : FALSE;
  ((next(mBlock) = On) & (!(mBlock = On))) & (mReset = Off) : TRUE;
  TRUE : tOverridden;
  esac;
 mcPressure = High:
  case
  (next(mcPressure) != mcPressure) : FALSE;
  TRUE : tOverridden;
  esac;
esac;
-- transicion a ejecutar
TRANS
 (!(next(waterPres) = waterPres) & (next(waterPres) - waterPres) <= 10
 & (waterPres - next(waterPres)) <= 10 & next(mBlock) = mBlock & next(mReset) = mReset) ;
INIT -- estado desde el que parte la transicion
(mcPressure=TooLow) & (waterPres=14)
LTLSPEC -- estado a chequear si es alcanzable
G(!( (mcPressure=Permitted) & (waterPres>=Permit)))
```

Figura 39: Especificación ACTION LANGUAGE para refinar el SIS.

(guiado por el contrajemplo abstracto) que viola la propiedad P, podemos asegurar que esta ejecución será un contraejemplo concreto para P en *Spec*.

5.5 SOBRE EL ANÁLISIS AUTOMÁTICO DE ESPECIFICACIONES SCR

El método SCR a sido ampliamente aplicado para describir requisitos de software [K. Heninger and Shore, 1978; Heitmeyer and McLean, 1983; van Schouwen et al., 1993], y es el formalismo bajo una amplia gama de análisis (semi-)automáticos: type-checking, chequeos de consistencia (disyunción y completitud), simulación y verificación [Heitmeyer et al., 2005].

En este trabajo, proponemos utilizar nuestra técnica automática, recién presentada en Sección 5.4, para atacar dos problemas complejos sobre las especificaciones SCR: la verificación de *invariantes* de estado y transición; y la generación de *casos de tests*. A pesar de que existen varias técnicas automáticas que atacan estos problemas (como el SCR Toolset), en general fallan cuando deben analizar especificaciones grandes y complejas, como mostraremos en la siguiente sección.

5.5.1 *Verificación de Invariantes*

Particularmente, centraremos la verificación en un cierto tipo de propiedades de safety conocidas como *invariantes*. Sea *Spec* una especificación SCR, y $\Sigma = (S, S_0, E^m, T)$ el LTS asociado a *Spec*. Un *invariante de estado* es un predicado P sobre el conjunto de estados S, que debe ser válido sobre todos los estados alcanzables de Σ (es decir, $Reach(\Sigma) \subseteq [P]$). Luego, para verificar la validez de un invariante de estado, podemos computar los estados alcanzables del sistema (o una sobre-aproximación de estos) y chequear si todos satisfacen P. Por otro lado, un *invariante de transición* es un predicado P_T sobre $S \times S$ que debe ser válido sobre toda transición que el sistema pueda realizar. Como las transiciones factibles del sistema sólo conducen a estados alcanzables, es posible verificar la validez de P_T mientras se computa $Reach(\Sigma)$ (o una sobre-aproximación del mismo).

Nuestro interés por este tipo de propiedades de safety, se debe a que es muy común en las especificaciones SCR de la literatura, encontrar este tipo de propiedades. A modo de ejemplo, a continuación listamos posibles invariantes, cuya validez deberemos de verificar, sobre el Safety Injection System, tomadas de [Bultan and Heitmeyer, 2008]:

1. $\texttt{tOverridden} \Rightarrow \texttt{mcPressure} \neq \texttt{High} \wedge \texttt{mReset} = \texttt{Off}$
2. $\texttt{mReset} = \texttt{On} \wedge \texttt{mcPressure} \neq \texttt{High} \Rightarrow \neg\texttt{tOverridden} \wedge \texttt{cSafetyInjection} = \texttt{On}$
3. $\texttt{mReset} = \texttt{On} \wedge \texttt{mcPressure} \neq \texttt{High} \Rightarrow \neg\texttt{tOverridden}$
4. $\texttt{mReset} = \texttt{On} \wedge \texttt{mcPressure} = \texttt{TooLow} \Rightarrow \texttt{cSafetyInjection} = \texttt{On}$

5.5.2 *Generación de Casos de Tests*

Debido a su naturaleza formal, las especificaciones de requisitos SCR pueden ser utilizadas automáticamente para generar casos de tests, tal como se describe en [Gargantini and Heitmeyer, 1999]. Intuitivamente, un caso de test es una ejecución (finita) de la especificación SCR que cubre cierta funcionalidad. Más precisamente, un caso de test es una secuencia de eventos de entrada, cuya ejecución, produce un conjunto respectivo de estados (computados acorde las definición de las tablas SCR), alcanzando un estado particular que cubre cierta funcionalidad. Luego, estos casos de tests pueden ser utilizados para *validar* el comportamiento del software que implementa dicha especificación SCR. Para esto, podemos chequear si, para un caso de test, el software produce las mismas salidas esperadas cuando se ejecutan los eventos de entradas correspondientes.

Note aquí la importancia de construir casos de tests para la especificación SCR original (en vez de abstracciones manuales de la especificación): los casos de tests generados pueden ser directamente usados para contrastar el sistema con su especificación. De otra manera, si utilizaramos una abstracción manual de la especificación para generar los casos de tests, estos deberían ser adaptados (probablemente de forma manual) a las unidades del sistema real, una tarea costosa, difícil, y propensa a errores.

Para poder utilizar nuestra técnica para generar casos de tests para especificaciones SCR, vamos a seguir [Beyer et al., 2004; Fraser and Gargantini, 2009; Gargantini and Heitmeyer, 1999] donde diferentes técnicas basadas en model checking fueron introducidas. Primero, debemos construir todos los *predicados de test* correspondiente al criterio de cobertura de interés. Cada uno de estos predicados de test caracterizan una clase de equivalencia particular de casos de test, en el criterio de cobertura correspondiente. Así, para cada predicado de test P, generamos una *propiedad trampa* $TP = \neg P$. Luego, invocamos a nuestro enfoque con la especificación SCR y TP como entradas. Si nuestra técnica genera un contraejemplo para TP, entonces hemos encontrado una ejecución que alcanza un estado que satisface P, es decir, un *caso de test* para P. En otro caso, P es un predicado de test que no es realizable.

Diferentes criterios de cobertura se han propuesto para especificaciones SCR. Para evaluar nuestra técnica, vamos a considerar los siguientes criterios:

- Cobertura de Tabla (T): Toda celda es alcanzable.
- *Cobertura de Partición de Modos* (SM): Si una celda se refiere a varios modos, luego la celda es cubierta por cada modo por separado.
- División de Desigualdades (DS): la expresiones con desigualdades son cubiertas con ambos casos de la desigualdad. Por ejemplo, $\geq$ es dividido en $>$ y $=$.
- Cobertura de Límites (B): cubre los valores bordes de las desigualdades. Por ejemplo, $x > C$ (con x y C enteros) es dividido en $x = C + 1$ y $x > C + 1$.
- Cobertura de Condición-Decisión Modificada (MCDC): Cada literal (condición) se considera que de manera independiente puede modificar el valor de una expresión (decisión) de la cual forma parte.

Por ejemplo, considere la Tabla de Eventos para el término `tOverridden` definida en la Tabla 4. Para satisfacer el criterio de Cobertura de Tabla, un caso de test debe ser construido para cada evento de la tabla (distinto al evento never):

1. `@F(mcPressure=High) WHEN mcPressure = High`
2. `@T(mBlock=On) WHEN mReset=Off`
   `AND ((mcPressure=TooLow) OR (mcPressure=Permitted))`
3. `@T(mcPressure=High) OR @T(mReset=On)`
   `AND ((mcPressure=TooLow) OR (mcPressure=Permitted))`
4. y un caso de test para cuando `tOverridden` no cambia (un evento diferente a los precedentes ocurre).

Para cubrir el predicado de test 1, debemos generar una ejecución que comience en modo `TooLow` hasta alcazar el modo `High` (pasando a travéz del modo `Permitted`), produciendo cambios en la variable `mWaterPres`. Finalmente, debe retornar al modo `Permitted`, así de esa manera la condición `@F(mcPressure=High) WHEN mcPressure = High` es satisfecha. Veamos una posible ejecución del SIS que cubriría este predicado de test:

$$
\begin{aligned}
\sigma = &< \texttt{TooLow}, \texttt{mWaterPres=14}, \ldots > \\
&(\texttt{mWaterPres}, 14, 24) < \texttt{TooLow}, \texttt{mWaterPres=24}, \ldots > \ldots \\
&(\texttt{mWaterPres}, 894, 904) < \texttt{Permitted}, \texttt{mWaterPres=904}, \ldots > \ldots \\
&(\texttt{mWaterPres}, 3094, 4004) < \texttt{High}, \texttt{mWaterPres=4004}, \ldots > \ldots \\
&(\texttt{mWaterPres}, 4004, 3094) < \texttt{Permitted}, \texttt{mWaterPres=3094}, \ldots >
\end{aligned}
$$

5.6 EVALUACIÓN EXPERIMENTAL Y DISCUSIÓN

Nuestra evaluación experimental tiene tres objetivos. En la Subsección 5.6.1, proveemos una evaluación de nuestro algoritmo de Abstracción Lazy para especificaciones SCR, presentado en la Sección 5.3. Los resultados evidencian los importantes beneficios de localizar por modos los predicados de abstracción, y modularizar la relación de transición, para el análisis de especificaciones SCR. Más adelante, comparamos nuestra técnica completa, descripta en Sección 5.4, con técnicas del estado del arte para la generación automática de casos de test y verificación de invariantes, en las Subsecciones 5.6.2 y 5.6.3, respectivamente.

Los casos de estudios seleccionados son modelos comúnmente disponibles en la literatura: Cruise Control System (`ccs`), Safety Injection System (`sis`) usado a lo largo de este capítulo, aircraft's autopilot (`autopilot`), y un protocolo para sobrepasar automóviles inteligentes (`car3prop`). [Fraser and Gargantini, 2009] compara el desempeño de varios model checkers en el análisis de especificaciones SCR, usando versiones manualmente simplificadas (mayormente usando constantes y rangos numéricos más chicos) de los modelos SCR que usaremos como casos de estudio. En la Subsección 5.6.1, donde evaluamos nuestro algoritmo de Abstracción Lazy, utilizamos las especificaciones simplificadas de [Fraser and Gargantini, 2009]. Por otro lado, para evaluar el desempeño de nuestra técnica completa respecto de las técnicas basadas en model checking, usaremos las versiones originales (más grandes) de las especificaciones SCR, es decir, las especificaciones al nivel de abstracción de las descripciones textuales de cada sistema. Típicamente, las especificaciones originales tienen variables numéricas con grande rangos, por lo que las constantes son mucho más grandes que las empleadas en [Fraser and Gargantini, 2009]. Por ejemplo, la especificación `autopilot reduced` contiene variables enteras con rangos entre $[0 \ldots 500]$, mientras que la especificación `autopilot` original tiene las mismas variables pero con rango $[0 \ldots 10000]$.

Como mencionamos al comienzo de este capítulo, uno de nuestros principales objetivos fue desarrollar una técnica que sea capaz de lidiar con el nivel de detalle original de las especificaciones SCR, ya que nos brindará grandes beneficios para las tareas de *validación* - comparación de la especificación con el comportamiento esperado por el usuario - y *verificación* - evaluar si la implementación del sistema cumple con la especificación de los requisitos.

Todos los experimentos fueron ejecutados sobre un procesador de 2.6GHz Intel Core 2 Duo con 3GB de RAM (2.5GB como memoria máxima para la herramienta de análisis), corriendo en GNU/Linux 3.0. Puede encontrar y descargar nuestra herramienta y los casos de estudios aquí mencionados desde `http://dc.exa.unrc.edu.ar/staff/rdegiovanni/case-studies/SCR_Analysis.zip`.

5.6.1 *Evaluación del Algoritmo de Abstracción Lazy para SCR*

El objetivo en esta sección es evaluar los beneficios de dos características importantes de nuestro algoritmo de Abstracción Lazy presentado en la Sección 5.3: los modos son usados para localizar los predicados de abstracción, y la función de transformación SCR es modularizada acorde a las dependencias entre las entidades (ver Subsección 5.3.3). Note que en esta sección no estamos evaluando nuestro enfoque completo (por ejemplo, no consideramos el paso de relajar la relación NAT de la especificación).

Con este objetivo en mente, evaluamos nuestro algoritmo de Abstracción usando diferentes configuraciones:

- (LA): Abstracción Lazy Estándar, los predicados de soporte son locales a regiones y la función de transformación no es modularizada.
- (LA + m.): LA más abstracción de modo explícito, usandolos para localizar los predicados de abstracción.
- (LA + mt.): LA más la función de transformación SCR modularizada.
- (LA + m. + mt.): Nuestro algoritmo de Abstracción Lazy, el cuál usa los modos para localizar los predicados de abstracción, y modulariza la función de transformación acorde las dependencias entre las entidades.

La comparación experimental en este caso, consiste en realizar generación automática de casos de tests para tres casos de estudio (`sis reduced`, `car3prop` and `autopilot reduced`), usando todos los criterios de cobertura de la Sección 5.5. Note que estos modelos son versiones reducidas de las originales, es decir, con constantes y rangos numéricos más pequeños, las utilizadas en [Fraser

and Gargantini, 2009] (`car3prop` no es reducida ya que no posee variables numéricas).

La Tabla 7 resume los resultados de estos experimentos. La primer fila de la tabla muestra el nombre del caso de estudio, y entre paréntesis, el numero de predicados de test a ser cubiertos (para todos los criterios de cobertura considerados). La columna *runs* indica el número de veces que una técnica tuvo que ser ejecutada para generar los casos de test necesarios para cubrir todos los predicados de test, para todos los criterios. La columna *c/i/e* muestra el número correspondiente de predicados de test *c*ubiertos (aquellos para los cuales un caso de test fue generado), *i*rrealizables (aquellos para los cuales las técnicas los identifican como inviables), y *e*rrores (aquellos para los cuales las técnicas son inconclusas). Para los predicados de test cubiertos, indicamos entre paréntesis el número de trazas que se necesitaron para cubrirlos, ya que una misma ejecución puede cubrir varios predicados de test. La columna *time* indica cuantos segundos le tomó en total a la técnica generar los casos de test para todos los predicados de test considerados. Hay que remarcar que, cuando cualquiera de las técnicas se ejecuta por más de una hora, es detenida y el predicado de test correspondiente es marcado como erróneo.

Modelo	car3prop (498 TPs.)			sis reduced (91 TPs.)			autopilot reduced (409 TPs.)		
	runs	c/i/e	time	runs	c/i/e	time	runs	c/i/e	time
LA	110	402(14)/90/**6**	33620	19	80(4)/11/**0**	8496	21	395(7)/10/**4**	23497
LA + m.	114	401(17)/96/**1**	8858	21	80(10)/11/**0**	5319	81	347(19)/10/**52**	113401
LA + mt.	118	402(22)/74/**22**	69197	20	80(9)/11/**0**	13032	51	389(31)/10/**10**	22201
LA + m. + mt.	119	402(29)/96/**0**	1795	23	80(12)/11/**0**	4724	54	399(44)/10/**0**	3951

Tabla 7: Evaluación de nuestro algoritmo de Abstracción Lazy para SCR.

Los resultados en Tabla 7 indican que nuestro algoritmo de Abstracción Lazy para SCR (La + m. + mt.) es 6x más rápido que Abstracción Lazy Estándar (LA). Más aún, nuestro algoritmo puede cubrir todos los predicados de test, mientras su competidor no puede hacerlo. Esto ocurre porque la abstracción estándar debe manejar demasiados predicados de abstracción, o debe re-descubrir muchos predicados (ver más abajo), haciendo al modelo abstracto muy costoso de construir (dentro del límite de 1 hora); y le es más dificultoso refinar los predicados adecuados para remover contraejemplos espurios cuando la abstracción no es de modo explícito y la función de transformación no fue modularizada.

En nuestra técnica, los predicados de soporte son locales a los modos en vez de regiones. Previamente hemos argumentado que, debido a la estructura de las tablas, las regiones con el mismo modo tienden a compartir los predicados de soporte. Para validar esa hipótesis, hemos elegido de manera aleatoria algunos predicados de soporte, y medimos, para abstracción estándar y nuestra técnica,

el número de nodos visitados y el números de predicados descubiertos. Además, para el caso de Abstracción Lazy Estándar (LA), medimos el número máximo de veces que un mismo predicado de soporte es "re-descubierto" para diferentes regiones con el mismo modo. Estos resultados pueden observarse en la Tabla 8.

CS	LA			LA + m. + mt.	
	nodes	predicates	most repeated	nodes	predicates
car3prop P1	41	94	30	76	11
car3prop P2	48	75	33	75	8
sis reduced P1	916	158	50	936	65
sis reduced P2	113	25	8	113	17
autopilot reduced P1	1037	126	21	1192	32
autopilot reduced P2	965	126	21	1120	32

Tabla 8: Predicados locales a modos vs. Predicados locales a regiones.

Claramente, para analizar especificaciones SCR utilizando abstracción, es conveniente localizar los predicados usando los modos, ya que nos permite reducir considerablemente el número de predicados necesarios para construir el modelo abstracto, y por supuesto, reducir también la cantidad de invocaciones al proceso que refina la abstracción. Estos resultados, son una de las principales razones por las cuales nuestro algoritmo de abstración tiene un mejor comportamiento que la Abstracción Lazy Estándar, mostrado en Tabla 7. Es importante remarcar que los modelos que involucran variables numéricas con grandes rangos, como `sis`, son aquellos en donde el algoritmo estándar de abstracción Lazy invierte más esfuerzo en re-descubrir predicados de soporte.

5.6.2 *Resultados Experimentales para Generación de Casos de Test*

En esta sección, evaluamos y comparamos nuestro enfoque completo con técnicas del estado del arte para la generación automática de casos de tests para especificaciones SCR, en general, basadas en model checking. Recientemente, una evaluación sobre las capacidades de varios model checkers para esta tarea fué presentada en [Fraser and Gargantini, 2009]. Aquí, comparamos los model checkers usados en [Fraser and Gargantini, 2009] con nuestra técnica, sobre los casos de estudio ya mencionados: `ccs`, `sis`, `autopilot` y `car3prop`. Es importante remarcar que, para los experimentos de esta sección, consideramos las especificaciones en su nivel de detalle original, manteniendo los rangos numéricos reales, sin ser reducidos manualmente. Utilizaremos varios model checkers con diferentes características (por ejemplo, algunos son model checkers de estados explícitos, otros son simbólicos) para la evaluación: Spin, NuSMV,

Cadence SMV y SAL. Corrimos estas herramientas con numerosas variedades de configuraciones, por lo que sólo vamos a reportar los mejores resultados producidos por cada una. Cuando una herramienta no es mencionada para algún caso de estudio, significa que su desempeño fue significativamente peor que los otros model checkers.

La Tabla 9 muestra los resultados para estos experimentos. En la tabla, nuestra técnica es referida como "SCR Abs. Lazy", mientras que cada model checker es referido por su nombre. La tabla muestra, para cada caso de estudio y técnica, el número de veces que la herramienta fue invocada (*runs*), el número de predicados de test cubiertos (c), los irrealizables (i), y los no cubiertos (e) -cuando la técnica falla en generar el caso de test o probarlo como irrealizable. Como en la sección experimental anterior, aquí se utilizan todos los criterios de cobertura mencionados en la Sección 5.5 para producir los predicados de test. El número total de predicados de test es especificado en la tabla, junto al nombre del caso de estudio. La columna *time* indica el tiempo total (en segundos) de ejecución de cada herramienta. Para nuestro enfoque SCR Abs. Lazy, también distinguimos entre el tiempo consumido por el algoritmo de Abstracción Lazy (Abs. Time), el tiempo requerido por model checking para concretizar los contraejemplos abstractos (M.C. Conc), y el tiempo de model checking para refinar la abstracción para remover violaciones espurias (M.C. Ref.). Para estos experimentos, configuramos un timeout de análisis para cada predicado de test de 5 minutos, y un tiempo total de análisis a cada herramienta de 5 horas. La elección de esta configuración se debe a que utilizaremos las especificaciones originales (grande y complejas), por lo que muchos de los model checkers demorarían varios días en dar una respuesta (en caso de tener éxito en su análisis).

Discutamos los resultados experimentales presentados en la Tabla 9. Primero, note que nuestra técnica es capaz de generar casos de tests para casi todos los predicados de test (el único error se observa en el modelo `autopilot`), incluso cuando los model checkers fallan. Esto es más evidente en los casos de estudios más grandes, como el `ccs` y `autopilot`, donde los modelos contienen muchas variables monitoreadas con dominios numéricos muy grandes.

En el caso de estudio más grande, el modelo `ccs`, nuestra técnica no falla para ningún caso de estudio, mientras que casi todos los model checkers consistentemente fallan en su análisis debido al enorme espacio de estados que deben explorar. Spin es el único model checker que puede generar casos de test para varios predicados de test (cerca del 25 %). Sin embargo, Spin sólo logra hacerlo para los predicados de test "fáciles", es decir, aquellos que corresponden a tests que pueden ser cubiertos explorando sólo una pequeña parte del espacio de estados (note que Spin nunca marca un predicado de test como irrealizable, ya

	Runs	Tests (c/i/e)	Time	Abs. Time	M.C. Conc.	M.C. Ref.
autopilot (409 test predicates)						
Spin	169	288(48)/0/**121**	2351	-	-	-
Cad. SMV	378	36(5)/0/**373**	timeout	-	-	-
SAL/SMC	296	116(3)/0/**293**	timeout	-	-	-
SCR Abs. Lazy	234	398(223)/10/**1**	1390	227	848	313
ccs (582 test predicates)						
Spin	582	164/0/**418**	3456	-	-	-
Cad. SMV	582	0/0/**582**	timeout	-	-	-
SCR Abs. Lazy	457	482(357)/100/**0**	4657	1402	1602	1472
sis (91 test predicates)						
Spin	19	80(8)/11/**0**	146	-	-	-
NuSMV	27	80(16)/11/**0**	995	-	-	-
Cad. SMV	31	80(20)/11/**0**	420	-	-	-
SAL/SMC	27	80(16)/11/**0**	38	-	-	-
SCR Abs. Lazy	70	80(59)/11/**0**	156	13	111	30
car3prop (498 test predicates)						
NuSMV	142	402(46)/96/**0**	261	-	-	-
Cad. SMV	160	402(64)/96/**0**	78	-	-	-
SAL/BMC	133	402(37)/81/**15**	56	-	-	-
SCR Abs. Lazy	460	402(364)/96/**0**	2509	2059	450	0

Tabla 9: Comparación de nuestro enfoque con varios model checkers, para la generación de casos de tests para especificaciones SCR.

que tendría que explorar todo el espacio de estados). Es importante remarcar que Spin es muy rápido para generar casos de test, pero como es un model checker de estados explícitos, generalmente se queda sin memoria muy rápido si el modelo es muy grande.

La segunta especificación más grande, el `autopilot`, muestra un comportamiento similar al `ccs`, aunque los model checkers logran cubrir más predicados de test para este caso (cerca del 70 % Spin). A pesar de que nuestra técnica es claramente superior en este caso de estudio, este es el único caso en el cuál nuestro enfoque reporta un *error*: se queda sin tiempo (más de 5 minutos) tratando de concretizar un contraejemplo abstracto (ejecutando Spin). Similar al `ccs`, note que ningún model checker logra marcar un predicado de test como irrealizable para el `autopilot`, debido nuevamente al enorme espacio de estados a explorar.

Por otro lado, para las especificaciones más pequeñas, algunos model checkers pueden ser más rápidos que nuestro enfoque. Por ejemplo, como el modelo `sis` contiene tres variables monitoreadas, de las cuales sólo una es numérica (con un rango no muy grande, de 0 a 5000), no representa demasiados problemas para los model checkers. En particular, SAL (usando bounded model checking) es

más rápido que nuestra técnica, pero nuestro tiempo sigue siendo comparable con los model checkers restantes. Para el caso del `car3prop`, todos los model checkers son bastante más rápidos que nuestro enfoque. Es importante remarcar que el modelo de `car3prop` no tiene variables numéricas, pero si tiene un grán número de variables monitoreadas. Luego, por un lado, la ausencia de variables numéricas permiten a los model checkers explorar el espacio de estados completo muy eficientemente. Por otro lado, nuestro enfoque no se beneficia de relajar la relación NAT (ya que no hay variables numéricas), y debido a las características de este modelo, debe introducir un número muy alto de predicados de abstracción, empeorando aún más los tiempos comparados a los model checkers. De todas formas, a diferencia de los casos en que los model checker fallan, nuestro enfoque es capaz de lidiar con este modelo exitosamente (aunque menos eficientemente).

Finalmente, desde la Tabla 9 observamos que el tiempo de abstracción es mucho menor que el tiempo invertido en model checking (con la única excepción del modelo `car3prop`). Esto nos da evidencias para dar soporte a nuestra hipótesis inicial, que establece que muchas de las propiedades interesantes pueden ser probadas sin considerar los valores específicos de las variables numéricas. Sin embargo, para las propiedades en las cuales el valor preciso de las variables numéricas es importante, debemos utilizar model checking para generar los casos de tests concretos y refinar la abstracción, un proceso mucho más costoso que nuestro algorítmo de abstracción.

5.6.3 *Resultados Experimentales para Verificación*

Ahora nos concentraremos en mostrar que nuestro enfoque también puede ser utilizado para eficientemente verificar propiedades sobre especificaciones SCR, y generar contraejemplos en caso de que la propiedad no sea válida. Diferentes herramientas del estado del arte que provee el SCR Toolset [Heitmeyer et al., 2005] y el model checker (de estados infinitos) ALV [Bultan and Heitmeyer, 2008] han sido utilizados para verificar invariantes de estado y transición para especificaciones SCR. En sus trabajos, han reportado que estas técnicas requieren *reducir manualmente* las especificaciones, para reducir el espacio de estados a explorar, y así sus técnicas puedan lograr analizar exitosamente las especificaciones SCR. Sin embargo, estas especificaciones reducidas manualmente no se encuentran públicamente disponibles, y por lo tanto, no pudimos experimentar directamente sobre esas especificaciones, lo que sería una evaluación más apropiada.

Luego, nuestro objetivo en esta sección es evidenciar que nuestro enfoque puede verificar propiedades sin la necesidad de abstracciones manuales, sin realizar una comparación con las técnicas mencionadas. Note que no tenemos en

cuenta técnicas basadas en demostradores de teoremas, ya que nos centramos en técnicas completamente automáticas para verificación.

Los invariantes que consideramos han sido tomados de [Heitmeyer et al., 2005; Bultan and Heitmeyer, 2008; Bharadwaj and Heitmeyer, 1997]: 11 para el modelo `ccs`, 4 para el `sis` y 2 para el `autopilot`, respectivamente. La Tabla 10 reporta que 9 de las propiedades del `ccs`, 3 del `sis` y las 2 del `autopilot`, fueron verificadas como invariantes válidos. Note que nuestro enfoque demandó 9 segundos en el peor caso para verificar una propeidad en el modelo más grande, el `ccs`. La Tabla 11 reporta 2 propieades inválidas para el `ccs` y 1 para el `sis`. Nuestra técnica requirió 12 segundos para generar el contraejemplo más complejo en el modelo `ccs`, y sólo 1 segundo para el `sis`. Es importante mencionar que ALV puede verificar como invariantes las 9 propiedades del `ccs` en pocos segundos, pero no es útil para generar los contraejemplos, incluso reduciendo manualmente el modelo, su desempeño es considerablemente peor que nuestro enfoque [Bultan and Heitmeyer, 2008]. Por otro lado, Spin puede generar eficientemente contraejemplos para el `sis` y algunos para el `ccs`, pero en general falla cuando analiza algunas propiedades sobre el `ccs` (requiriendo aplicar abstracciones manuales [Heitmeyer et al., 2005]).

Model	Average Time	Shortest Time	Longest Time
`ccs` (9)	1.44 sec	0.10 sec	9 sec
`sis` (3)	0.66 sec	0.10 sec	2 sec
`autopilot` (2)	1 sec	1 sec	1 sec

Tabla 10: Verificación de Invariantes.

Model	Average Time	Shortest Time	Longest Time
`ccs` (2)	7.5 sec	3 sec	12 sec
`sis` (1)	1 sec	-	-

Tabla 11: Generación de Contraejemplos.

5.7 TRABAJOS RELACIONADOS

Las notaciones tabulares, en particular SCR, tienen asociadas herramientas de soporte que proveen diferentes tipos de análisis, como por ejemplo: chequeo de

sintaxis, análisis de consistencia, y la verificación de propiedades [Atlee and Gannon, 1991; Bultan and Heitmeyer, 2008; Heitmeyer et al., 2005]. En particular, varias de estas herramientas nos permiten realizar verificación mediante model chekcing, pero usualmente requieren abstracciones manuales o no escalan de buena manera. Con respecto a testing, el simulador del SCR Toolset [Heitmeyer et al., 2005] permite a los desarrolladores cargar escenarios específicos, los cuales son provistos en un principio por los ingenieros, y chequear si ciertas aserciones, son violadas o no en las ejecuciones particulares descriptas por dichos escenarios. Gargantini y Heitmeyer [Gargantini and Heitmeyer, 1999] usaron un model checker para automáticamente obtener casos de tests que recorrieran diferentes modos del sistema. Recientemente, Fraser y Gargantini [Fraser and Gargantini, 2009] realizaron una extensa comparasión entre varios model checkers (simbólicos, acotados, de estados explicícitos, etc.) para automáticamente generar casos de tests a partir de especificaciones tabulares, analizando la cobertura alcanzada y los problemas de escalabilidad que surgieron. Con respecto a la generación de casos de tests, la evaluación de nuestra técnica se basa en los casos de estudio analizados por Fraser y Gargantini, y la comparación se realiza sobre los mismos model checkers considerados en [Fraser and Gargantini, 2009]. Bultan y Heitmeyer [Bultan and Heitmeyer, 2006] han utilizado el model checker de estados infinitos ALV para analizar especificaciones tabulares. Como explicamos en la Subsección 5.4.3, ALV ha mostrado ser útil para verificar que una propiedad es válida, pero poco eficiente para generar contraejemplos. Por lo que no es una buena opción para generar casos de tests. En particular, para la verificación de propiedades, ALV necesita reducir manualmente algunos modelos para poder generar contraejemplos. Sin embargo, estas especificaciones reducidas manualmente no están publicamente disponibles, por lo que nuestra comparación se basa en resultados publicados (por ejemplo, en [Bultan and Heitmeyer, 2008]) en vez de compararla con corridas efectivas de la herramienta. En la Sección 5.6 realizamos una extensa comparación de nuestra técnica con la mayoría de los trabajos recién mencionados, mostrando que nuestra técnica es mucho más eficiente que los otros enfoques, a la hora de analizar especificaciones de requisitos de gran tamaño.

A nuestro conocimiento, técnicas automáticas de abstracción no son aplicadas con frecuencia para propósitos relacionados al testing. En [Bharadwaj and Heitmeyer, 1999] aplican abstracción a especificaciones SCR, con el propósito de mejorar la escalabilidad de tareas basadas en model checking. La técnica de abstracción de Bharadwaj y Heitmeyer, consiste en remover variables irrelevantes para la propiedad de interés (slicing), y la transformación de variables "internas" en variables monitoreadas, con el objetivo de remover detalles de algunas variables (en especial, las numéricas). Sin embargo, en ese caso, sólo proveen resultados experimentales de analizar algunas propiedades sobre dos

especificaciones SCR pequeñas. En [Heitmeyer et al., 2005] ellos evalúan sus técnicas de abstracción sobre varios ejemplos, mencionando la necesidad de realizar diferentes abstracciones manuales ad-hoc para que el análisis de algunas propiedades finalice correctamente al usar model checking. Como mostramos en la Subsección 5.6.3, nuestra técnica es capaz de lidiar con estas mismas propiedades de forma completamente automática.

La técnica presentada en este capítulo extiende y mejora nuestro trabajo previo, presentado en [Degiovanni et al., 2011]. La principal diferencia reside en el manejo de las variables numéricas, que como hemos explicado, traen complicaciones importantes a los algoritmos automáticos de abstracción. En particular, en [Degiovanni et al., 2011] propusimos una heurística que consiste en dividir el dominio de las variables numéricas en rangos (computados acorde al tamaño del dominio y la variación de la variable) para localizar los predicados de soporte de la abstracción, y así lograr convergencia en muchos casos. Sin embargo, la eficiencia de esa heurística es muy poco efectiva, si la comparamos a los resultados experimentales que hemos obtenido con la técnica actual, presentada en esta tesis.

Numerosos enfoques exitosos para la verificación y generación de casos de test han sido propuestos. Algunos de estos están basados en SMT solving, abstracción, combinaciones y variantes. La mayoría de estos trabajos, se centran en el *análisis de código fuente* en vez de especificaciones de requisitos. En particular, *Abstracción Lazy* y el refinamiento de abstracciones basadas en interpolación han sido utilizadas para automáticamente generar casos de tests que nos conducen a locaciones alcanzables de un programa, con aplicaciones exitosas en drivers y programas de seguridad crítica [Beyer et al., 2004]. Otros enfoques similares a este tipo son [Henzinger et al., 2004; Chaki et al., 2004]. Nuestro técnica se basa en la presentada en [Beyer et al., 2004], la cuál utiliza abstracción Lazy [Henzinger et al., 2002] para la generación de casos de tests, pero aplica sobre especificaciones de requisitos. A diferencia de los dominios de los programas donde las abstracciones son exitosamente aplicadas [Clarke et al., 2005], las especificaciones de requisitos no son “control intensive”, por lo que este trabajo contribuye un novedoso dominio de aplicación. Nuestra algoritmo de abstracción ha sido especialmente adaptado a especificaciones de requisitos, explotando sus características particulares, para hacer que la abstracción sea aplicable de manera eficiente al análisis de especificaciones SCR.

Otras técnicas automáticas, como Pex [Tillmann and De Halleux, 2008] y JPF [Visser et al., 2004], son exitosamente aplicadas a la generación de casos de tests *de caja blanca* para programas en .NET y Java, respectivamente. Estas herramientas aplican sobre código fuente, y se basan sobre *ejecución simbólica*

en vez de abstracción por predicados con refinamiento automático, como es nuestro caso.

5.8 RESUMEN

En este capítulo hemos identificado un número de características inherentes a las especificaciones tabulares de requisitos, que pueden ser explotadas para mejorar el análisis automático de este tipo de especificaciones. Luego, presentamos un enfoque novedoso para el análisis de especificaciones SCR, aplicando un proceso de abstracción de dos niveles, que explotan las características identificadas, y un método de refinamiento basado en model checking. Para mostrar los beneficios de nuestro enfoque, lo comparamos con técnicas del estado del arte para la generación de casos de tests y la verificación de propiedades de especificaciones SCR, sobre varios casos de estudio que ha sido ampliamente utilizados en la literatura. Nuestro enfoque mostró mejor eficiencia y escalabilidad que las técnicas existentes, y fué capaz de analizar todas las especificaciones de nuestros casos de estudio en su nivel de detalle original, a diferencia de los otros enfoques que fallan en varios casos, más notablemente en aquellos cuyas especificaciones contienen variables numéricas con rangos muy grandes.

La importancia de las tareas de validación y verificación, motiva la necesidad de lidiar con especificaciones grandes. Por un lado, es importante que los casos de tests generados mantengan el nivel de detalle esperado por el usuario, y utilizado en la implementación del sistema. De otra manera, cada caso de tests tendría que ser manualmente adaptado por el ingeniero para que sea útil en el contexto original, un proceso lento, tedioso y propenso a errores. Por otro lado, las propiedades probadas como válidas en modelos reducidos manualmente, no tienen garantías de ser válidas en el modelo original. Además, los contraejemplos reportados deberán ser adaptados al nivel de detalle original, para que puedan ser depurados.

A pesar de que abstracción ha sido exitosamente usada para mejorar el análisis de model checking, en general es aplicada en dominios “control intensive” [Clarke et al., 2005]. Sin embargo, las especificaciones de requisitos no son “control intensive”, por lo que constituyen un dominio de aplicación desafiante para aplicar análisis basado en abstracción. Como muestran nuestros experimentos, la aplicación directa de Abstracción Lazy en el contexto de especificaciones de requisitos no tiene un buen desempeño, mientras que nuestra variante, que explota las características de las tablas, muestra sus beneficios.

6

Conclusiones y Trabajos Futuros

En las últimas décadas los métodos formales han ganado una influencia notable dentro de la Ingeniería de Requisitos. Tal es el caso de las metodologías orientadas a objetivos y las notaciones tabulares, que han logrado gran aceptación y se han aplicado sobre numerosos casos prácticos. El uso de lenguajes formales permite, entre otras cosas, eliminar todo tipo de ambigüedades sobre las especificaciones de requisitos, y las hace adecuadas para el análisis automático, por ejemplo, para el chequeo de consistencia y completitud de los requisitos. Sin embargo, otras áreas de la Ingeniería de Software, como la verificación automática de programas, han sabido explotar en mayor medida el poder de los mecanismos de análisis asociados a los métodos formales, como SAT Solving e interpolación.

En esta tesis, mostramos que es posible aprovechar este poder de análisis que proveen los métodos formales a lo largo de todo el proceso de ingeniería de requisitos, contribuyendo a la elaboración y construcción de requisitos de software (etapa temprana del proceso de requisitos), y a la validación y verificación de las especificaciones de requisitos construidas (etapa tardía del proceso de requisitos). Mostramos además, que nuestras técnicas no sólo logran mejores niveles de escalabilidad que las técnicas relacionadas en el estado del arte del área, sino que además pueden aplicarse a casos más generales (como es el caso de nuestra técnica de operacionalización de objetivos, que puede lidiar con un amplio conjunto de propiedades de liveness).

6.1 Conclusiones

La principal contribución de esta tesis es el desarrollo de dos novedosas técnicas *automáticas* que, mediante la manipulación lógica de fórmulas, asisten al ingeniero en tareas específicas llevadas a cabo a lo largo del proceso de ingeniería de requisitos. Nuestra primera técnica ayuda al ingeniero a resolver la incompletitud de los requisitos operacionales, garantizando que la especificación obtenida satisface un conjunto de propiedades deseables. Por otro lado, la segunda técnica nos permite analizar si el comportamiento descripto por nuestras

especificaciones de requisitos cumplen con las expectativas del cliente (validación) y se corresponde con la implementación del sistema (verificación). Puede notarse que, en general, nuestra primera técnica es más aplicable durante la etapa temprana del proceso de ingeniería de requisitos, donde el ingeniero debe lidiar y resolver la parcialidad de los requisitos; mientras que la segunta técnica, requiere una descripción más precisa y acabada de los requisitos, por lo que es aplicable en la etapa tardía de la ingeniería de requisitos.

En el Capítulo 4 presentamos una técnica para la *operacionalización automática de objetivos*. Básicamente, nuestra técnica utiliza model checking para detectar la incompletitud de un Modelo Operacional respecto a un conjunto de objetivos, y de manera automática, combinando interpolación y SAT solving, refina las condiciones requeridas necesarias para garantizar la satisfacción de los objetivos. Esta técnica propone dos contribuciones destacadas. Primero, nuestro enfoque es completamente automático, a diferencia de los existentes que son manuales [Letier and van Lamsweerde, 2002b] o a lo sumo semi-automáticos [Alrajeh et al., 2009], requiriendo la asistencia del usuario en el proceso de refinamiento. Segundo, nuestra técnica es capaz de lidiar con un amplio rango de propiedades de *liveness*, mientras que los enfoque previos para la operacionalización de objetivos [Letier and van Lamsweerde, 2002b; Alrajeh et al., 2009] sólo aplican a objetivos de safety y a un tipo muy particular de propiedades de progreso, que es el progreso del tiempo (*Time Progress*). Además demostramos que nuestra técnica es correcta y garantiza terminación cuando tratamos con objetivos de safety. Más aún, brindamos una metodología que el ingeniero de requisitos puede seguir, para acercarse lo más posible a la solución óptima, en la Sección 4.6.

En el Capítulo 5 presentamos una técnica automática para el análisis de especificaciones de requisitos, aplicada a tareas ligadas a la *validación* y *verificación* de requisitos. En particular, mostramos que nuestra técnica puede ser utilizada para generar casos de tests y verificar propiedades sobre las especificaciones de requisitos. Básicamente, la técnica aplica un proceso de *abstracción automático*, demostrado ser correcto, logrando notables mejoras en la eficiencia y escalabilidad respecto a las técnicas existentes. La principal contribución de esta técnica es que puede lidiar con el nivel de detalle original de las especificaciones, sin someterlas a reducciones manuales o fallar en el proceso de análisis cuando otras técnicas lo hacen. Esto permite, entre otras cosas, que los casos de tests generados (o las violaciones detectadas) puedan ser directamente contrastados con las expectativas del usuario (validación) y con el comportamiento real del sistema implementado (verificación).

6.2 TRABAJOS FUTUROS

Tenemos varias líneas de trabajos futuros. Por un lado, planeamos aplicar nuestra técnica de operacionalización de objetivos sobre casos de estudio más grandes, que nos permitan evaluar si existen problemas de escalabilidad, para así realizar las mejoras necesarias. Además, planeamos investigar cuál es la relación precisa entre las operacionalizaciones obtenidas por interpolación con aquellas obtenidas con programación lógica inductiva (ILP). En particular, estamos interesados en analizar una posible noción de operacionalización "más general", y evaluar si el refinamiento basado en interpolación nos permite alcanzar dicha operacionalización. Además, planeamos investigar una potencial complementación entre interpolación e ILP, para la operacionalización de objetivos.

Por otro lado, estamos estudiando la posibilidad de utilizar interpolación para la detección y resolución de conflictos a nivel de objetivos. Intuitivamente, podemos pensar a un interpolante sobre dos objetivos inconsistentes, como un obstáculo, es decir, una condición cuya validez garantiza que no podrán ser satisfechos ambos objetivos a la vez.

Finalmente, debido a que nuestras técnicas recaen fuertemente en el cómputo de interpolantes, planeamos evaluar mecanismos alternativos que computan interpolantes, para analizar si algún algoritmo particular de interpolación es más adecuado para nuestros propósitos.

Bibliografía

[Alrajeh et al., 2009] Alrajeh, D., Kramer, J., Russo, A., and Uchitel, S. (2009). Learning operational requirements from goal models. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 265–275, Washington, DC, USA. IEEE Computer Society. 8, 11, 38, 73, 74, 117, 124, 141, 142, 193

[Alrajeh et al., 2013] Alrajeh, D., Kramer, J., Russo, A., and Uchitel, S. (2013). Elaborating requirements using model checking and inductive learning. *IEEE Trans. Softw. Eng.*, 39(3):361–383. 11, 105, 109, 116, 117, 141

[Alrajeh et al., 2012] Alrajeh, D., Kramer, J., van Lamsweerde, A., Russo, A., and Uchitel, S. (2012). Generating obstacle conditions for requirements completeness. In *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland*, pages 705–715. 5

[Alur and Henzinger, 1994] Alur, R. and Henzinger, T. A. (1994). A really temporal logic. *J. ACM*, 41(1):181–203. 34

[Anton, 1997] Anton, A. I. (1997). *Goal Identification and Refinement in the Specification of Software-based Information Systems*. PhD thesis, Atlanta, GA, USA. UMI Order No. GAX97-35409. 7, 140

[Atlee and Gannon, 1991] Atlee, J. and Gannon, J. (1991). State-based model checking of event-driven system requirements. *SIGSOFT Softw. Eng. Notes*, 16(5):16–28. 9, 189

[Baier and Katoen, 2008] Baier, C. and Katoen, J.-P. (2008). *Principles of Model Checking*. The MIT Press. 91

[Bak et al., 2011] Bak, K., Czarnecki, K., and Wasowski, A. (2011). Feature and meta-models in clafer: Mixed, specialized, and coupled. In *Proceedings of the Third International Conference on Software Language Engineering*, SLE'10, pages 102–122, Berlin, Heidelberg. Springer-Verlag. 5

[Bell and Thayer, 1976] Bell, T. E. and Thayer, T. A. (1976). Software requirements: Are they really a problem? In *Proceedings of the 2Nd International Conference on Software Engineering*, ICSE '76, pages 61–68, Los Alamitos, CA, USA. IEEE Computer Society Press. 2

[Beyer et al., 2004] Beyer, D., Chlipala, A. J., Henzinger, T. A., Jhala, R., and Majumdar, R. (2004). Generating tests from counterexamples. In *Proceedings of the 26th International Conference on Software Engineering*, ICSE '04, pages 326–335, Washington, DC, USA. IEEE Computer Society. 179, 190

[Bharadwaj and Heitmeyer, 1997] Bharadwaj, R. and Heitmeyer, C. (1997). Applying the scr requirements method to a simple autopilot. In *In Proc. Fourth NASA Langley Formal Methods Workshop (LFM97), NASA Langley Research*. 146, 188

[Bharadwaj and Heitmeyer, 1999] Bharadwaj, R. and Heitmeyer, C. L. (1999). Model checking complete requirements specifications using abstraction. *Automated Software Engineering*, 6(1):37–68. 9, 169, 170, 172, 189

[Biere et al., 1999] Biere, A., Cimatti, A., Clarke, E. M., and Zhu, Y. (1999). Symbolic model checking without bdds. In *Proceedings of the 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, TACAS '99, pages 193–207, London, UK, UK. Springer-Verlag. 6, 22, 92

[Blackburn et al., 2006] Blackburn, P., Benthem, J. F. A. K. v., and Wolter, F. (2006). *Handbook of Modal Logic, Volume 3 (Studies in Logic and Practical Reasoning)*. Elsevier Science Inc., New York, NY, USA. 33

[Boehm and Papaccio, 1988] Boehm, B. W. and Papaccio, P. N. (1988). Understanding and controlling software costs. *IEEE Trans. Softw. Eng.*, 14(10):1462–1477. 2, 47

[Bradley and Manna, 2007] Bradley, A. R. and Manna, Z. (2007). *The Calculus of Computation: Decision Procedures with Applications to Verification*. Springer-Verlag New York, Inc., Secaucus, NJ, USA. 94

[Brooks, 1987] Brooks, Jr., F. P. (1987). No silver bullet essence and accidents of software engineering. *Computer*, 20(4):10–19. 2

[Bruttomesso et al., 2008] Bruttomesso, R., Cimatti, A., Franzén, A., Griggio, A., and Sebastiani, R. (2008). The mathsat 4 smt solver. In *Proceedings of the 20th International Conference on Computer Aided Verification*, CAV '08, pages 299–303, Berlin, Heidelberg. Springer-Verlag. 23, 157, 165

[Bruttomesso et al., 2007] Bruttomesso, R., Cimatti, A., FranzÃľn, A., Griggio, A., Hanna, Z., Nadel, A., Palti, A., and Sebastiani, R. (2007). A lazy and layered smt($\mathcal{BV}$) solver for hard industrial verification problems. In Damm, W. and Hermanns, H., editors, *CAV*, volume 4590 of *Lecture Notes in Computer Science*, pages 547–560. Springer. 23

[Bultan, 2000] Bultan, T. (2000). Action language: A specification language for model checking reactive systems. In *Proceedings of the 22Nd International Conference on Software Engineering*, ICSE '00, pages 335–344, New York, NY, USA. ACM. 33

[Bultan and Heitmeyer, 2006] Bultan, T. and Heitmeyer, C. (2006). Analyzing tabular requirements specifications using infinite state model checking. In *Proceedings of the Fourth ACM and IEEE International Conference on Formal Methods and Models for Co-Design, 2006. MEMOCODE '06. Proceedings.*, MEMOCODE '06, pages 7–16, Washington, DC, USA. IEEE Computer Society. 175, 189

[Bultan and Heitmeyer, 2008] Bultan, T. and Heitmeyer, C. L. (2008). Applying infinite state model checking and other analysis techniques to tabular requirements specifications of safety-critical systems. *Design Autom. for Emb. Sys.*, 12(1-2):97–137. 9, 149, 167, 175, 178, 187, 188, 189

[Bultan and Yavuz-Kahveci, 2001] Bultan, T. and Yavuz-Kahveci, T. (2001). Action language verifier. In *Automated Software Engineering, 2001. (ASE 2001). Proceedings. 16th Annual International Conference on*, pages 382–386. 43

[Butler, 1996] Butler, R. W. (1996). An introduction to requirements capture using pvs: Specification of a simple autopilot. Technical report. 9

[Cabodi et al., 2006] Cabodi, G., Murciano, M., Nocco, S., and Quer, S. (2006). Stepping forward with interpolants in unbounded model checking. In *2006 International Conference on Computer-Aided Design (ICCAD'06), November 5-9, 2006, San Jose, CA, USA*, pages 772–778. 19

[Chaki et al., 2004] Chaki, S., Clarke, E. M., Groce, A., Jha, S., and Veith, H. (2004). Modular verification of software components in c. *IEEE Trans. Software Eng.*, 30(6):388–402. 190

[Cimatti et al., 2010] Cimatti, A., Griggio, A., and Sebastiani, R. (2010). Efficient generation of craig interpolants in satisfiability modulo theories. *ACM Trans. Comput. Logic*, 12(1):7:1–7:54. 19, 23

[Clarke et al., 2003] Clarke, E., Grumberg, O., Jha, S., Lu, Y., and Veith, H. (2003). Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM*, 50(5):752–794. 6, 22, 140

[Clarke, 2008] Clarke, E. M. (2008). 25 years of model checking. chapter The Birth of Model Checking, pages 1–26. Springer-Verlag, Berlin, Heidelberg. 39

[Clarke and Emerson, 1981] Clarke, E. M. and Emerson, E. A. (1981). Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs*, pages 52–71. 39

[Clarke et al., 1986] Clarke, E. M., Emerson, E. A., and Sistla, A. P. (1986). Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst.*, 8(2):244–263. 39

[Clarke et al., 2005] Clarke, E. M., Gupta, A., Jain, H., and Veith, H. (2005). Model checking: Back and forth between hardware and software. In Meyer, B. and Woodcock, J., editors, *VSTTE*, volume 4171 of *Lecture Notes in Computer Science*, pages 251–255. Springer. 190, 191

[Clarke et al., 1999] Clarke, Jr., E. M., Grumberg, O., and Peled, D. A. (1999). *Model Checking*. MIT Press, Cambridge, MA, USA. 39

[Cockburn, 2000] Cockburn, A. (2000). *Writing Effective Use Cases*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition. 4, 47

[Cook, 1971] Cook, S. A. (1971). The complexity of theorem-proving procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, STOC '71, pages 151–158, New York, NY, USA. ACM. 17

[Courtois and Parnas, 1993] Courtois, P.-J. and Parnas, D. L. (1993). Documentation for safety critical software. In Basili, V. R., DeMillo, R. A., and Katayama, T., editors, *ICSE*, pages 315–323. IEEE Computer Society / ACM Press. 60, 125

[Craig, 1957a] Craig, W. (1957a). Linear reasoning. A new form of the herbrand-gentzen theorem. *J. Symb. Log.*, 22(3):250–268. 19

[Craig, 1957b] Craig, W. (1957b). Three uses of the herbrand-gentzen theorem in relating model theory and proof theory. *J. Symb. Log.*, 22(3):269–285. 20

[Dardenne et al., 1993] Dardenne, A., van Lamsweerde, A., and Fickas, S. (1993). Goal-directed requirements acquisition. In *SCIENCE OF COMPUTER PROGRAMMING*, pages 3–50. 6, 7, 34, 38, 48, 51, 52, 54, 72, 73, 98, 140

[Davis et al., 1962] Davis, M., Logemann, G., and Loveland, D. (1962). A machine program for theorem-proving. *Commun. ACM*, 5(7):394–397. 18

[Davis and Putnam, 1960] Davis, M. and Putnam, H. (1960). A computing procedure for quantification theory. *J. ACM*, 7(3):201–215. 18

[de Roever and Hooman, 1989] de Roever, W. P. and Hooman, J. (1989). Design and verification in real-time distributed computing: an introduction to compositional methods. In *Protocol Specification, Testing and Verification IX, Proceedings of the IFIP WG6.1 Ninth International Symposium on Protocol Specification, Testing and Verification, Enschede, The Netherlands, 6-9 June, 1989*, pages 37–56. 106

[Degiovanni et al., 2014] Degiovanni, R., Alrajeh, D., Aguirre, N., and Uchitel, S. (2014). Automated goal operationalisation based on interpolation and sat solving. In *ICSE*, pages 129–139. 12, 73, 117

[Degiovanni et al., 2011] Degiovanni, R., Ponzio, P., Aguirre, N., and Frias, M. F. (2011). Abstraction based automated test generation from formal tabular requirements specifications. In *TAP*, pages 84–101. 12, 146, 151, 190

[DeMarco, 1979] DeMarco, T. (1979). Classics in software engineering. chapter Structured Analysis and System Specification, pages 409–424. Yourdon Press, Upper Saddle River, NJ, USA. 4, 47

[D'Ippolito et al., 2011] D'Ippolito, N., Braberman, V., Piterman, N., and Uchitel, S. (2011). Synthesis of live behaviour models for fallible domains. pages 211–220. IEEE. 141

[D'Ippolito et al., 2010] D'Ippolito, N. R., Braberman, V., Piterman, N., and Uchitel, S. (2010). Synthesis of live behaviour models. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '10, pages 77–86, New York, NY, USA. ACM. 141

[Fraser and Gargantini, 2009] Fraser, G. and Gargantini, A. (2009). An evaluation of model checkers for specification based test case generation. In *ICST*, pages 41–50. 9, 11, 149, 167, 179, 181, 182, 184, 189

[Fuxman et al., 2004] Fuxman, A., Liu, L., Mylopoulos, J., Pistore, M., Roveri, M., and Traverso, P. (2004). Specifying and analyzing early requirements in tropos. *Requirements Engineering*, 9(2):132–150. 8, 140

[Fuxman et al., 2001] Fuxman, A., Pistore, M., Mylopoulos, J., and Traverso, P. (2001). Model checking early requirements specifications in tropos. In *Requirements Engineering, 2001. Proceedings. Fifth IEEE International Symposium on*, pages 174–181. 8, 140

[Gargantini and Heitmeyer, 1999] Gargantini, A. and Heitmeyer, C. (1999). Using model checking to generate tests from requirements specifications. *SIGSOFT Softw. Eng. Notes*, 24(6):146–162. 5, 9, 147, 179, 189

[Ghezzi et al., 2002] Ghezzi, C., Jazayeri, M., and Mandrioli, D. (2002). *Fundamentals of Software Engineering*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2nd edition. 2, 5, 47

[Giannakopoulou and Magee, 2003] Giannakopoulou, D. and Magee, J. (2003). Fluent model checking for event-based systems. In *ESEC SIGSOFT FSE*, pages 257–266. 36

[Graf and Saïdi, 1997] Graf, S. and Saïdi, H. (1997). Construction of abstract state graphs with pvs. In *Proceedings of the 9th International Conference on Computer Aided Verification*, CAV '97, pages 72–83, London, UK, UK. Springer-Verlag. 157, 158, 159

[Gunter et al., 2000] Gunter, C. A., Gunter, E. L., Jackson, M., and Zave, P. (2000). A reference model for requirements and specifications. *IEEE Softw.*, 17(3):37–43. 45, 47, 70

[Harel and Politi, 1998] Harel, D. and Politi, M. (1998). *Modeling Reactive Systems with Statecharts: The Statemate Approach*. McGraw-Hill, Inc., New York, NY, USA, 1st edition. 5

[Heitmeyer et al., 1998] Heitmeyer, C., Kirby, J., and Labaw, B. (1998). Applying the scr requirements method to a weapons control panel: An experience report. In *Proceedings of the Second Workshop on Formal Methods in Software Practice*, FMSP '98, pages 92–102, New York, NY, USA. ACM. 9

[Heitmeyer et al., 2005] Heitmeyer, C. L., Archer, M., Bharadwaj, R., and Jeffords, R. D. (2005). Tools for constructing requirements specifications: the scr toolset at the age of ten. *Comput. Syst. Sci. Eng.*, 20(1). 5, 9, 11, 34, 38, 144, 145, 149, 167, 178, 187, 188, 189, 190

[Heitmeyer et al., 1996] Heitmeyer, C. L., Jeffords, R. D., and Labaw, B. G. (1996). Automated consistency checking of requirements specifications. *ACM Trans. Softw. Eng. Methodol.*, 5(3):231–261. 5, 7, 59, 61

[Heitmeyer and McLean, 1983] Heitmeyer, C. L. and McLean, J. D. (1983). Abstract requirements specification: A new approach and its application. *IEEE Trans. Softw. Eng.*, 9(5):580–589. 59, 144, 178

[Henzinger et al., 2004] Henzinger, T. A., Jhala, R., Majumdar, R., and McMillan, K. L. (2004). Abstractions from proofs. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '04, pages 232–244, New York, NY, USA. ACM. 164, 190

[Henzinger et al., 2002] Henzinger, T. A., Jhala, R., Majumdar, R., and Sutre, G. (2002). Lazy abstraction. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '02, pages 58–70, New York, NY, USA. ACM. 152, 158, 161, 163, 190

[Hoare, 1985] Hoare, C. A. R. (1985). *Communicating Sequential Processes*. Prentice-Hall. 30

[Holzmann, 1991] Holzmann, G. J. (1991). *Design and Validation of Computer Protocols*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA. 31

[Holzmann, 2004] Holzmann, G. J. (2004). *The SPIN Model Checker - primer and reference manual*. Addison-Wesley. 31, 42, 169

[IEEE, 1998] IEEE (1998). Ieee recommended practice for software requirements specifications. 4, 5

[Jackson and Vaziri, 2000] Jackson, D. and Vaziri, M. (2000). Finding bugs with a constraint solver. In *Proceedings of the 2000 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '00, pages 14–25, New York, NY, USA. ACM. 6, 22

[Jackson, 1995] Jackson, M. (1995). The world and the machine. In *Proceedings of the 17th International Conference on Software Engineering*, ICSE '95, pages 283–292, New York, NY, USA. ACM. 3, 45

[Jalote, 2005] Jalote, P. (2005). *An Integrated Approach to Software Engineering*. Texts in Computer Science. Springer. 2, 5, 47

[Jhala, 1999] Jhala, R. (1999). *Lazy Abstraction*. PhD thesis. 159

[K. Heninger and Shore, 1978] K. Heninger, J. Kallander, D. P. and Shore, J. (1978). Software requirements for the a-7e aircraft. Technical report. 9, 59, 144, 178

[Kautz and Selman, 1992] Kautz, H. and Selman, B. (1992). Planning as satisfiability. In *IN ECAI-92*, pages 359–363. Wiley. 6, 22

[Kazhamiakin et al., 2004] Kazhamiakin, R., Pistore, M., and Roveri, M. (2004). Formal verification of requirements using spin: A case study on web services. In *SEFM*, pages 406–415. IEEE Computer Society. 38

[Keller, 1976] Keller, R. M. (1976). Formal verification of parallel programs. *Commun. ACM*, 19(7):371–384. 26

[Kesten et al., 1993] Kesten, Y., Manna, Z., Manna, Z., Pnueli, A., Mcguire, H., and Pnueli, A. (1993). A decision algorithm for full propositional temporal logic. pages 97–109. Springer-Verlag. 40

[Krajícek, 1997] Krajícek, J. (1997). Interpolation theorems, lower bounds for proof systems, and independence results for bounded arithmetic. *Journal of Symbolic Logic*, 62:457–486. 21

[Kramer et al., 1983] Kramer, J., Magee, J., Sloman, M., and Lister, A. (1983). CONIC: an integrated approach to distributed computer control systems. *Computers and Digital Techniques, IEE Proceedings E*, 130(1):1+. 24, 75, 117

[Lamport, 1980] Lamport, L. (1980). "sometimeïs sometimes "not never": On the temporal logic of programs. In *Proceedings of the 7th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '80, pages 174–185, New York, NY, USA. ACM. 34, 43

[Letier, 2001] Letier, E. (2001). *Reasoning about Agents in Goal-Oriented Requirements Engineering*. PhD thesis. 110, 136

[Letier, 2002] Letier, E. (2002). Goal-oriented elaboration of requirements for a safety injection control system. Technical report. 117, 125

[Letier and Heaven, 2013] Letier, E. and Heaven, W. (2013). Requirements modelling by synthesis of deontic input-output automata. In *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*, pages 592–601. 5, 141

[Letier et al., 2005] Letier, E., Kramer, J., Magee, J., and Uchitel, S. (2005). Fluent temporal logic for discrete-time event-based models. In Wermelinger, M. and Gall, H., editors, *ESEC/SIGSOFT FSE*, pages 70–79. ACM. 38, 81

[Letier et al., 2008] Letier, E., Kramer, J., Magee, J., and Uchitel, S. (2008). Deriving event-based transition systems from goal-oriented requirements models. *Automated Software Engg.*, 15(2):175–206. 5, 59, 76, 77, 80, 81, 82, 83, 85, 86, 102, 105

[Letier and van Lamsweerde, 2002a] Letier, E. and van Lamsweerde, A. (2002a). Agent-based tactics for goal-oriented requirements elaboration. In *Proceedings of the 24th International Conference on Software Engineering*, ICSE '02, pages 83–93, New York, NY, USA. ACM. 8

[Letier and van Lamsweerde, 2002b] Letier, E. and van Lamsweerde, A. (2002b). Deriving operational software specifications from system goals. In *SIGSOFT FSE*, pages 119–128. 8, 38, 50, 54, 56, 57, 73, 78, 193

[Letier and van Lamsweerde, 2004] Letier, E. and van Lamsweerde, A. (2004). Reasoning about partial goal satisfaction for requirements and design engineering. *SIGSOFT Softw. Eng. Notes*, 29(6):53–62. 8

[Lewis, 1918] Lewis, C. I. (1918). *A survey of symbolic logic*. University of California Press. Republished by Dover, 1960. 33

[Li and Somenzi, 2006] Li, B. and Somenzi, F. (2006). Efficient abstraction refinement in interpolation-based unbounded model checking. In *Tools and Algorithms for the Construction and Analysis of Systems, 12th International Conference, TACAS 2006 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2006, Vienna, Austria, March 25 - April 2, 2006, Proceedings*, pages 227–241. 19

[Magee and Kramer, 2006] Magee, J. and Kramer, J. (2006). *Concurrency - state models and Java programs (2. ed.)*. Wiley. 8, 30, 31, 41, 76, 81, 102, 105

[Maiden and Alexander, 2004] Maiden, N. and Alexander, I. (2004). *Scenarios, stories, use cases : through the systems development life-cycle*. J. Wiley and sons, Chichester. 4, 47

[Manna and Pnueli, 1992] Manna, Z. and Pnueli, A. (1992). *The Temporal Logic of Reactive and Concurrent Systems*. Springer-Verlag New York, Inc., New York, NY, USA. 10, 34, 35, 38, 39, 74, 85, 109, 110

[Manna and Pnueli, 1995] Manna, Z. and Pnueli, A. (1995). *Temporal verification of reactive systems: safety*. Springer-Verlag New York, Inc., New York, NY, USA. 34, 35, 38

[McMillan, 2003] McMillan, K. L. (2003). Interpolation and sat-based model checking. In *CAV*, pages 1–13. 19, 21, 22, 140, 164

[Mcmillan, 2005] Mcmillan, K. L. (2005). Applications of craig interpolants in model checking. In *In TACASâĂŹ2005: Tools and Algorithms for the Construction and Analysis of Systems, LNCS 3440*, pages 1–12. Springer. 19, 74, 140

[McMillan, 2005] McMillan, K. L. (2005). An interpolating theorem prover. *Theor. Comput. Sci.*, 345(1):101–121. 24

[Miller and Shanahan, 1999] Miller, R. and Shanahan, M. (1999). The event calculus in classical logic - alternative axiomatisations. *Electron. Trans. Artif. Intell.*, 3(A):77–105. 36

[Milner, 1980] Milner, R. (1980). *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer. 30

[Mitchell, 2008] Mitchell, B. (2008). Characterizing communication channel deadlocks in sequence diagrams. *IEEE Trans. Software Eng.*, 34(3):305–320. 5

[Mitchell, 1997] Mitchell, T. (1997). *Machine Learning (Mcgraw-Hill International Edit)*. McGraw-Hill Education (ISE Editions), 1st edition. 8, 140

[Mylopoulos et al., 1992] Mylopoulos, J., Chung, L., and Nixon, B. (1992). Representing and using nonfunctional requirements: A process-oriented approach. *IEEE Trans. Softw. Eng.*, 18(6):483–497. 8, 140

[Naur et al., 1969] Naur, P., Randell, B., Bauer, F., and Committee, N. S. (1969). *Software engineering: report on a conference sponsored by the NATO Science Committee, Garmisch, Germany, 7th to 11th October 1968*. Scientific Affairs Division, NATO. 2

[Owicki and Lamport, 1982] Owicki, S. and Lamport, L. (1982). Proving liveness properties of concurrent programs. *ACM Trans. Program. Lang. Syst.*, 4(3):455–495. 39

[Parnas and Madey, 1995] Parnas, D. L. and Madey, J. (1995). Functional documents for computer systems. *Science of Computer Programming*, 25:41–61. 45, 46, 59

[Pnueli, 1977] Pnueli, A. (1977). The temporal logic of programs. In *FOCS*, pages 46–57. 34, 35, 39

[Prior, 1957] Prior, A. N. (1957). *Time and Modality*. Oxford University Press. 34

[Pudlák, 1997] Pudlák, P. (1997). Lower bounds for resolution and cutting plane proofs and monotone computations. *Journal of Symbolic Logic*, 62:981–998. 21

[Regis et al., 2015] Regis, G., Degiovanni, R., D'Ippolito, N., and Aguirre, N. (2015). Specifying event-based systems with a counting fluent temporal logic. In *ICSE, To appear*. 12

[Rolland et al., 1998] Rolland, C., Souveyet, C., and Achour, C. B. (1998). Guiding goal modelling using scenarios. *IEEE Transaction on Software Engineering*, 24:1055–1071. 8, 140

[Rumbaugh et al., 1999] Rumbaugh, J., Jacobson, I., and Booch, G., editors (1999). *The Unified Modeling Language Reference Manual*. Addison-Wesley Longman Ltd., Essex, UK, UK. 4, 47

[Scilingo et al., 2014] Scilingo, G., Novaira, M. M., and Degiovanni, R. (2014). Analyzing behavioural scenarios over tabular specifications using model checking. In *LAFM*, pages 71–76. 12

[Scilingo et al., 2013] Scilingo, G., Novaira, M. M., Degiovanni, R., and Aguirre, N. (2013). Analyzing formal requirements specifications using an off-the-shelf model checker. In *CLEI*, pages 1–9. 12

[Sistla, 1999] Sistla, A. P. (1999). Safety, liveness and fairness in temporal logic. In *Formal Aspect of Computing*, pages 495–511. 91

[Sommerville, 2006] Sommerville, I. (2006). *Software Engineering: (Update) (8th Edition) (International Computer Science)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA. 2, 5, 47

[Stephan et al., 1996] Stephan, P. R., Brayton, R. K., and Sangiovanni-Vincentelli, A. L. (1996). Combinational test generation using satisfiability. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 15(9):1167–1176. 6, 22

[Stevens et al., 1974] Stevens, W. P., Myers, G. J., and Constantine, L. L. (1974). Structured design. *IBM Syst. J.*, 13(2):115–139. 4, 47

[Tillmann and De Halleux, 2008] Tillmann, N. and De Halleux, J. (2008). Pex: White box test generation for .net. In *Proceedings of the 2Nd International Conference on Tests and Proofs*, TAP'08, pages 134–153, Berlin, Heidelberg. Springer-Verlag. 190

[Uchitel et al., 2007] Uchitel, S., Brunet, G., and Chechik, M. (2007). Behaviour model synthesis from properties and scenarios. In *Proceedings of the 29th International Conference on Software Engineering*, ICSE '07, pages 34–43, Washington, DC, USA. IEEE Computer Society. 38

[Uchitel et al., 2003] Uchitel, S., Kramer, J., and Magee, J. (2003). Synthesis of behavioral models from scenarios. *IEEE Trans. Software Eng.*, 29(2):99–115. 8

[Van et al., 2004] Van, H. T., van Lamsweerde, A., Massonet, P., and Ponsard, C. (2004). Goal-oriented requirements animation. In *12th IEEE International Conference on Requirements Engineering (RE 2004), 6-10 September 2004, Kyoto, Japan*, pages 218–228. 5

[van Lamsweerde, 2001] van Lamsweerde, A. (2001). Goal-oriented requirements engineering: A guided tour. In *Proceedings of the Fifth IEEE International Symposium on Requirements Engineering*, RE '01, pages 249–, Washington, DC, USA. IEEE Computer Society. 49

[van Lamsweerde, 2008] van Lamsweerde, A. (2008). Requirements engineering: From craft to discipline. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, SIGSOFT '08/FSE-16, pages 238–249, New York, NY, USA. ACM. 5

[van Lamsweerde, 2009] van Lamsweerde, A. (2009). *Requirements Engineering - From System Goals to UML Models to Software Specifications*. Wiley. 5, 49, 59

[van Lamsweerde and Willemet, 1998] van Lamsweerde, A. and Willemet, L. (1998). Inferring declarative requirements specifications from operational scenarios. *IEEE Trans. Software Eng.*, 24(12):1089–1114. 8, 140

[van Schouwen et al., 1993] van Schouwen, A. J., Parnas, D. L., and Madey, J. (1993). Documentation of requirements for computer systems. In *RE*, pages 198–207. 59, 144, 178

[Visser et al., 2004] Visser, W., Păsăreanu, C. S., and Khurshid, S. (2004). Test input generation with java pathfinder. *SIGSOFT Softw. Eng. Notes*, 29(4):97–107. 190

[von Wright, 1951a] von Wright, G. (1951a). *An Essay in Modal Logic*. North-Holland publishing Company. 34

[von Wright, 1951b] von Wright, G. H. (1951b). Deontic logic. *Mind*, 60:1–15. 34

[Weidenhaupt et al., 1998] Weidenhaupt, K., Pohl, K., Jarke, M., and Haumer, P. (1998). Scenario usage in system development: A report on current practice. In *ICRE*, pages 222–. IEEE Computer Society. 4

[Whitten and Bentley, 2007] Whitten, J. L. and Bentley, L. D. (2007). *Systems Analysis and Design Methods*. McGraw-Hill, Inc., New York, NY, USA, 7 edition. 4

[Yavuz-Kahveci et al., 2005] Yavuz-Kahveci, T., Bartzis, C., and Bultan, T. (2005). Action language verifier, extended. In *Proceedings of the 17th International Conference on Computer Aided Verification*, CAV'05, pages 413–417, Berlin, Heidelberg. Springer-Verlag. 43

[Yu, 1997] Yu, E. S. K. (1997). Towards modeling and reasoning support for early-phase requirements engineering. In *Proceedings of the 3rd IEEE International Symposium on Requirements Engineering*, RE '97, pages 226–, Washington, DC, USA. IEEE Computer Society. 7, 72, 140

[Zave and Jackson, 1997] Zave, P. and Jackson, M. (1997). Four dark corners of requirements engineering. *ACM Trans. Softw. Eng. Methodol.*, 6(1):1–30. 45, 54