Facultad de Matemática, Astronomía y Física
Universidad Nacional de Córdoba

# Análisis de Refinamientos entre Sistemas de Transiciones Modales basado en SAT

Carolina Inés Dania
Director: Dr. Nazareno Aguirre

16 de diciembre de 2008

# Resumen


Desde tiempos previos a la llamada *crisis del software* se ha reconocido que la complejidad y el tamaño de los sistemas de software demanda metodologías sistemáticas de desarrollo. El objetivo de éstas es permitir crear, diseñar y mantener (éxitosamente) software de calidad y de gran escala (y en los tiempos estipulados). En busca de proveer garantías del correcto funcionamiento del software, surgieron una variedad de técnicas y metodologías de desarrollo con sólidas bases matemáticas y lógicas.

Los sistemas de transición de estados (LTS), y la amplia mayoría de sus variantes constituyen un formalismo adecuado para la caracterización del comportamiento operacional de sistemas, incluyendo sistemas reactivos, concurrentes y distribuidos. En particular, los sistemas de transiciones modales (MTS) permiten descripciones parciales de sistemas, las cuales son útiles en etapas tempranas del desarrollo de software. Las relaciones de refinamiento entre MTS son centrales a esta idea. Éstas permiten identificar las especificaciones que más se acercan a la implementación del sistema.

El objetivo de este trabajo es equipar a los MTS con herramientas de análisis automático o semi-automático para poder estudiar a éstos objetos y a las relaciones de refinamiento entre ellos.



**Classificación:** D.2.4 Software/Program Verification, F.3.2 Semantics of Programming Languages.

**Palabras Claves:** LTS, MTS, Bisimulaciones, Refinamientos, Implementaciones, SAT solver, Alloy, MTSA.

*La vida es mucho mejor desde que se es licenciado.*<br>
Lic. Matías Bordese

# Agradecimientos

A mi vieja, mi viejo, Marcos y Flor por ser quienes siempre me apoyaron.

A mis abuelas, tios y primos por estar en cada momento que los necesité.

A Pedro por estar siempre en cada instante de mi vida.

Al Naza, no sólo por ser mi director sino por ser un amigo.

A mis amigos J, el flaco, Juli, Mati, Nati, Caro, Waldo, Lau, el recientemente Dr. King, Rafa, Santi, Ale, Fran, Carlos, Germán, Diego, Joshep y César K.

A mis otros amigos que están lejos, Matthi, Ceci y Xavi que no van a poder hacerme ningún daño.

A mis compañeros de laburo y a mis jefes.

A Pichu por los tips sobre uso de memoria y a Fernando por prestarme a Shiva.

Y a todos los que no estoy nombrando y que seguramente me lo reprocharán luego.

# Índice general

**1. Introducción** **11**

**2. Conceptos preliminares** **15**
2.1. Sistemas de Transiciones Etiquetadas . . . . . . . . . . . . . . . . . . 15
2.2. Simulaciones y Bisimulaciones . . . . . . . . . . . . . . . . . . . . . . 16
2.3. Sistemas de Transiciones Modales . . . . . . . . . . . . . . . . . . . . 20
2.4. Refinamientos . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 21
2.5. Implementaciones . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 23

**3. Modelado de MTS y LTS en Alloy** **27**
3.1. Definición de Universos . . . . . . . . . . . . . . . . . . . . . . . . . 27
3.2. Transiciones y funciones auxiliares . . . . . . . . . . . . . . . . . . . . 29
3.3. Relaciones entre LTS y MTS . . . . . . . . . . . . . . . . . . . . . . . 31
3.3.1. Simulaciones y Bisimulaciones . . . . . . . . . . . . . . . . . 31
3.3.2. Refinamientos . . . . . . . . . . . . . . . . . . . . . . . . . . 32
3.3.3. Implementaciones . . . . . . . . . . . . . . . . . . . . . . . . 32
3.4. Análisis de algunas propiedades elementales . . . . . . . . . . . . . . . 33
3.5. Algunas mejoras al modelado . . . . . . . . . . . . . . . . . . . . . . . 35
3.5.1. Simulaciones y Bisimulaciones . . . . . . . . . . . . . . . . . 36
3.5.2. Refinamientos . . . . . . . . . . . . . . . . . . . . . . . . . . 37
3.5.3. Implementaciones . . . . . . . . . . . . . . . . . . . . . . . . 37
3.6. Comparación de diversas herramientas en el aná-lisis de bisimulaciones y refinamientos . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 38
3.7. Ejemplos de otros análisis con Alloy Analyzer . . . . . . . . . . . . . 41

**4. Predicados de Alto Orden sobre MTS y LTS** **47**
4.1. Preservación de implementaciones en Alloy . . . . . . . . . . . . . . 48
4.1.1. Análisis de propiedades con cuantificación de Alto Orden . . 51
4.1.2. Limitaciones en el Análisis mediante Alloy Analyzer . . . . . 51
4.2. Mínimo conjunto de Relaciones . . . . . . . . . . . . . . . . . . . . . 52
4.2.1. En busca de relaciones representantes . . . . . . . . . . . . . 52
4.2.2. Minimizando relaciones con Alloy Analyzer . . . . . . . . . 53
4.3. Generación de LTS/MTS . . . . . . . . . . . . . . . . . . . . . . . . . 55
4.4. Metodología de Análisis . . . . . . . . . . . . . . . . . . . . . . . . . 56
4.5. Algunos Ejemplos de contraejemplos . . . . . . . . . . . . . . . . . . 58

4.6. Estimación de tiempo de análisis . . . . . . . . . . . . . . . . . . . . 59
4.7. Mejora al tiempo de análisis . . . . . . . . . . . . . . . . . . . . . . . 59
4.8. Una alternativa utilizando DynAlloy . . . . . . . . . . . . . . . . . . 59
4.8.1. Análisis de la propiedad mediante DynAlloy . . . . . . . . . 60

**5. Conclusiones y trabajo futuro** **61**
5.1. Conclusiones . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 61
5.2. Trabajo Futuro . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 62

**A. La herramienta MTSA** **63**
A.1. LTSA - Labeled Transition System Analyzer . . . . . . . . . . . . . . 63
A.2. MTSA - Modal Transition System Analyzer . . . . . . . . . . . . . . 64
A.3. MTSChecker . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 65

**B. Las herramientas Alloy y DynAlloy** **67**
B.1. Alloy . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 67
B.1.1. Gramática y semántica de Alloy . . . . . . . . . . . . . . . . 67
B.1.2. Signaturas . . . . . . . . . . . . . . . . . . . . . . . . . . . . 68
B.1.3. Fórmulas y declaraciones . . . . . . . . . . . . . . . . . . . . 68
B.1.4. Funciones, hechos, aserciones y predicados . . . . . . . . . . 69
B.1.5. Corridas y chequeos . . . . . . . . . . . . . . . . . . . . . . 70
B.1.6. Skolemización de Relaciones . . . . . . . . . . . . . . . . . . 70
B.2. DynAlloy . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 71
B.2.1. Gramática y semántica de DynAlloy . . . . . . . . . . . . . . 71
B.2.2. Predicados . . . . . . . . . . . . . . . . . . . . . . . . . . . . 71
B.3. KodKod . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 71
B.3.1. Sintáxis abstracta de KodKod . . . . . . . . . . . . . . . . . 72
B.4. Cotas . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 73

**C. Ejemplos** **75**
C.1. MTSA . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 75
C.2. MTSChecker . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 77
C.3. Alloy . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 79
C.4. KodKod . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 82

**D. Mínimo conjunto de Relaciones en Alloy** **87**

**E. Código** **95**
E.1. Disco Ram . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 95
E.2. Script de generación de MTS y análisis de nuestra propiedad . . . . . 96
E.2.1. Es conexo . . . . . . . . . . . . . . . . . . . . . . . . . . . . 100
E.2.2. Antecendente de la propiedad . . . . . . . . . . . . . . . . . 101
E.2.3. Consecuente de la propiedad . . . . . . . . . . . . . . . . . . 101
E.3. KodKod . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 102
E.3.1. Clase Transition . . . . . . . . . . . . . . . . . . . . . . . . 102
E.3.2. Clase LTS . . . . . . . . . . . . . . . . . . . . . . . . . . . . 107
E.3.3. Clase MTS . . . . . . . . . . . . . . . . . . . . . . . . . . . 109
E.3.4. Clase Rel . . . . . . . . . . . . . . . . . . . . . . . . . . . . 111
E.3.5. Ejemplos de (bi)simulaciones . . . . . . . . . . . . . . . . . 112
E.3.6. Ejemplos de refinamientos . . . . . . . . . . . . . . . . . . . 116
E.3.7. Ejemplos de implementaciones . . . . . . . . . . . . . . . . . 120

# 1 Introducción

Desde tiempos previos a la llamada *crisis del software* se ha reconocido que la complejidad y el tamaño de los sistemas de software, cuyo campo de aplicación ha crecido significativamente respecto de sus aplicaciones en cálculo numérico de los inicios de la computación, demanda metodologías sistemáticas de desarrollo [GJM02]. Por supuesto, el objetivo de éstas es permitir crear, diseñar y mantener (éxitosamente) software de calidad y de gran escala (y en los tiempos estipulados) [Som00]. Uno de los problemas más importantes asociados con la construcción de software es la corrección del mismo, es decir, en qué medida el software construido satisface los requisitos (funcionales o no) establecidos durante las etapas tempranas del desarrollo. En busca de proveer garantías del correcto funcionamiento del software, surgieron una variedad de técnicas y metodologías de desarrollo con sólidas bases matemáticas y lógicas conocidas como *métodos formales* [Dil94][Win90]. Debido a su naturaleza, la aplicación de métodos formales requiere gran experiencia y conocimientos, sobre todo en lo concerniente a matemática y lógica, por lo cual su aplicación resulta costosa en la práctica. Esto ha provocado que su principal aplicación se limite a sistemas críticos [Sto96] (es decir, sistemas de software/hardware cuyo mal funcionamiento o falla puede causar daños de magnitud, como la pérdida de vidas humanas, etc), aunque claramente los beneficios que sus técnicas proveen son relevantes a todo tipo de software. Por este motivo, las metodologías de desarrollo más utilizadas en la actualidad son informales, en el sentido de que las notaciones y procesos utilizados no cuentan con semántica formal o precisamente descripta en algún formalismo matemático (por ejemplo, la amplia mayoría de las notaciones abarcadas por UML [BRJ98], o las notaciones asociadas a metodologías más antiguas como las reportadas en [Ing87]). Más aún, el *testing*, claramente una técnica informal, sigue siendo en la actualidad la técnica para la garantía de corrección (funcional) del software más ampliamente utilizada en la práctica.

Otra de las razones por las cuales se dificulta la adopción de métodos formales en la práctica es que, en varios casos, las notaciones utilizadas en los ámbitos formal e informal son de distinta naturaleza. Un ejemplo claro de esto es el hecho de que, en notaciones informales como UML, las descripciones suelen interpretarse como *parciales* (que se completarán sucesivamente a medida que se gane conocimiento sobre el sistema a desarrollar), mientras que en notaciones formales las descripciones suelen ser *totales*, es decir, requieren contar con la información completa sobre el comportamiento del sistema para su formalización. Este es el caso particular de los *sistemas de transición de estados* (LTS), y la amplia mayoría de sus variantes. Los LTS constituyen un formalismo adecuado para la caracterización del comportamiento operacional

de sistemas, incluyendo sistemas reactivos, concurrentes y distribuidos (más aún, existen extensiones a la noción clásica de sistema de transición de estados que incorporan elementos probabilistas o de tiempo real, para poder describir sistemas con este tipo de componentes). Si bien los LTS constituyen un formalismo adecuado para describir el comportamiento de sistemas, éstos requieren contar con información *total* sobre el comportamiento de un sistema. Como resultado de esta observación, y ante la necesidad de poder describir *parcialmente* el comportamiento de un sistema, para así poder acercar la naturaleza de estas descripciones formales a la naturaleza más ampliamente difundida en descripciones informales, y para, por ejemplo, dar lugar a posibles implementaciones posteriores o para describir familias de productos [FUB06], surgieron en los últimos años variantes de los LTS que admiten parcialidad en la descripción [LT88, LX90, Fis06]. Una de estas variantes, los *sistemas de transición modales*, ha sido equipada con nociones de refinamiento e implementación, e incluso poseen herramientas de soporte.

Últimamente se ha reconocido que contar con herramientas de análisis automático o semi-automático es un elemento de gran importancia para contribuir a la utilización de métodos formales en la práctica. Es el objetivo de este trabajo precisamente éste, equipar a los MTS, como formalismo de descripción parcial de sistemas, con una herramienta de análisis potente. Existe actualmente una herramienta de este tipo, MTSA [DFCU08]; esta herramienta permite, dados un par de MTS, chequear si éstos son bisimilares bajo alguna de varias nociones diferentes de bisimulación [Mil89, vGW96]. La implementación de estos chequeos en MTSA se basa en variantes de los algoritmos tradicionales para chequeo de bisimilitud [DPP01, GV90]. Nuestro enfoque será diferente al empleado por el MTSA. Proponemos en este trabajo lo siguiente:

- la utilización de SAT solver [1] para el chequeo de existencia de relaciones de refinamiento/implementación entre MTS,
- el aprovechamiento de mecanismos avanzados de análisis basado en SAT, como la explotación de información parcial de modelos, para acelerar el proceso de análisis,
- el chequeo de cierto tipo de "meta propiedades" de MTS y las relaciones de refinamiento asociadas.

Por supuesto, debido a que pretendemos utilizar SAT para el análisis de relaciones de refinamiento e implementación entre MTS, la opción más directa sería la utilización de la lógica proposicional como lenguaje para la codificación de LTS/MTS, y la directa utilización de alguno de los muchos SAT solvers disponibles (por ejemplo, Berkmin [GN07], Minisat [ES03], Zchaff [MFM04], etc). Por el contrario, nuestra propuesta se enfoca en la utilización de un lenguaje de especificaciones de más alto nivel, llamado Alloy [Jac03], como lenguaje intermedio para la aplicación de SAT. Este lenguaje posee varias características que lo hacen adecuado para esta tarea, tales como:

- el lenguaje Alloy, fuertemente basado en la noción de relación, permite describir MTS, LTS y las relaciones entre ellos de manera declarativa y clara,
- algunas técnicas importantes de aceleración en el análisis basado en SAT se encuentran disponibles sólo a través de Alloy, y no son soportadas directamente por SAT solvers (por ejemplo, el aprovechamiento de información sobre cotas de relaciones y la aceleración mediante la eliminación de modelos isomorfos),

[1] Un SAT solver es una herramienta que permite responder automáticamente si una fórmula proposicional es satisfactible

- el Alloy Analyzer ofrece una traducción de conceptos necesarios en la descripción de máquinas de estados (LTS, MTS) que máximiza el aprovechamiento de los SAT solvers como herramientas de análisis.

Como mostraremos en este trabajo, el análisis basado en SAT es una opción viable y de gran utilidad para el estudio de MTS. Si bien en los estudios comparativos realizados la alternativa provista por el MTSA resultó ser más eficiente, el análisis realizado utilizando SAT solving se mantuvo en general en el orden de unos pocos segundos. Además, debido a la declaratividad del lenguaje de especificaciones utilizado, la herramienta resulta ser más versátil que MTSA, pues puede utilizarse no sólo para el chequeo de refinamientos, sino también para tareas relacionadas tales como la comparación de diferentes alternativas en relaciones de refinamiento/simulación (por ejemplo, en el estudio de adaptaciones de definiciones de estos conceptos, o en la generación de casos que satisfagan simultáneamente las condiciones de diferentes relaciones de simulación/refinamiento), etc.

A continuación presentamos algunos conceptos preliminares que serán necesarios en el desarrollo de la tesis. El resto de este trabajo está organizado de la siguiente manera. En el capítulo 3, presentamos una caracterización de MTS/LTS y relaciones de refinamiento e implementación en Alloy. Mostramos también algunos ejemplos, comparación de tiempos con otras herramientas, y algunas limitaciones del mecanismo de análisis asociado a Alloy con respecto al chequeo de propiedades de MTS/LTS.

En el capítulo 4 nos embarcamos en un intento de análisis de propiedades de alto orden de MTS, motivados por una propiedad particular: la completitud de refinamiento fuerte con respecto a implementaciones. Observaremos en este capítulo algunas limitaciones intrínsecas a la propiedad, que es expresable en Alloy pero no analizable por Alloy Analyzer. Discutiremos varias variantes de expresión de esta propiedad, y una simplificación al problema para la generación de candidatos a contraejemplos de la propiedad. También veremos como una extensión al lenguaje Alloy, llamada DynAlloy, contribuye en esta tarea de análisis.

Finalmente, en el capítulo 5 expondremos nuestras conclusiones sobre el trabajo, y en el capítulo 6 describiremos algunas líneas de trabajo futuro.

# 2 Conceptos preliminares

En este capítulo proveemos las definiciones de algunos conceptos necesarios en el desarrollo de este trabajo. Comenzaremos reproduciendo las definiciones de las máquinas de estados utilizadas, como así también de las diferentes relaciones entre éstas que estudiaremos, y analizaremos, en esta tesis.

## 2.1 Sistemas de Transiciones Etiquetadas

**Definición 2.1.1.** Un *sistema de transiciones etiquetadas (LTS - labelled transition systems)* es una estructura $\mathsf{P} = (S, L, \longrightarrow, s_0)$ donde $S$ es un conjunto finito de estados, $L$ es un conjunto finito de etiquetas observables más una etiqueta no observable (o silenciosa) llamada $\tau$, $\longrightarrow \subseteq (S \times L \times S)$ es una relación de transición entre estados, y $s_0 \in S$ es el estado inicial.

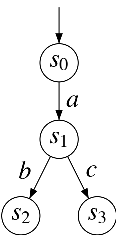



Figura 2.1: Ejemplo de un sistema de transiciones etiquetadas

Un sistema de transiciones etiquetadas $\mathsf{P} = (S, L, \longrightarrow, s_0)$ permite modelar el comportamiento de un sistema de la siguiente manera:

- el conjunto $S$ representa el espacio de todos los estados posibles que el sistema puede asumir. El estado inicial $s_0 \in S$ representa el estado a partir del cual se inicia la ejecución en el sistema.

- las etiquetas en $L$ representan las acciones que el sistema puede ejecutar, o los eventos a los que éste puede responder (el evento $\tau$ es simplemente utilizado para modelar acciones privadas del sistema, para las cuales no se quiere asociar eventos observables).

- las transiciones en $\longrightarrow$ representan el flujo de ejecución del sistema, es decir cómo se progresa de un estado a otro en el sistema a través de la ejecución de acciones, o la ocurrencia de eventos. Notemos que, dado que $\longrightarrow$ es una *relación*, estos modelos admiten naturalmente el no determinismo.

Para clarificar la definición anterior, consideremos el LTS que se muestra gráficamente en la Figura 2.1. Podemos ver el conjunto de estados $S = \{s_0, s_1, s_2, s_3\}$, el de acciones $L = \{a, b, c\}$ y el conjunto de transiciones permitidas. El estado inicial, $s_0$, está marcado mediante un arco entrante al nodo, sin origen. Asi, por ejemplo, si estamos en el estado $s_0$ podemos movernos al estado $s_1$ realizando la acción $a$.

Dados un LTS $\mathsf{P} = (S, L, \longrightarrow, s_0)$ y $s, s' \in S$, denotamos mediante $s \xrightarrow{l} s'$, el hecho de que $s$ transiciona por $l$ a $s'$ en $\mathsf{P}$, si y sólo si $(s, l, s') \in\longrightarrow$. De igual manera, escribimos $s \xrightarrow{\hat{l}} s'$ para denotar que $s \xrightarrow{l} s'$ o, $l = \tau$ y $s = s'$. Usamos $s \stackrel{l}{\Longrightarrow} s'$ para denotar $s(\xrightarrow{\tau})^* \xrightarrow{l} (\xrightarrow{\tau})^* s'$. Por último, utilizamos $s \stackrel{\hat{l}}{\Longrightarrow} s'$ para describir $s(\xrightarrow{\tau})^* \xrightarrow{l} (\xrightarrow{\tau})^* s'$ o, $l = \tau$ y $s = s'$.

Además dado un LTS $\mathsf{P} = (S, L, \longrightarrow, s_0)$, denotamos mediante $\mathit{Reach}(\mathsf{P})$ al conjunto de todos los estados de $S$ que son alcanzables (en cero o más pasos) a partir del estado $s_0$ mediante $\longrightarrow$.

## 2.2 Simulaciones y Bisimulaciones

Una manera de describir a los sistemas es a través del comportamiento. Quizás la manera más simple de dar semántica a LTS es mediante las ejecuciones que éstos admiten, que es justamente la *semántica de trazas* (debido a que las ejecuciones pueden representarse mediante trazas (secuencias) de etiquetas). La semántica formal de LTS mediante conjunto de trazas permite realizar numerosas tareas de análisis, entre las que podemos destacar la comparación de comportamientos de diferentes LTS. Esto permite intentar dar respuesta a preguntas tales como ¿son éstas descripciones equivalentes?, o ¿es esta descripción más fina que esta otra?, o ¿admite este LTS todas las ejecuciones de esta otra? Responder este tipo de preguntas tiene infinidad de aplicaciones prácticas, tales como reducción de LTS (minimización), o la definición de relaciones de refinamiento.

Veamos cuándo dos sistemas son equivalentes en la semántica de trazas. Según la semántica de trazas, dos sistemas son equivalentes si permiten el mismo conjunto de observaciones, donde una observación consiste simplemente en una secuencia de acciones realizadas por el sistema consecutivamente. Diremos que $\sigma = a_1 a_2 \ldots a_n$ es una traza generada por un LTS $\mathsf{P} = (S, L, \longrightarrow, s_0)$, si existen estados $s_1, s_2, \ldots, s_n \in S$ tal que $s_0 \xrightarrow{a_1} s_1 \xrightarrow{a_2} s_2 \xrightarrow{a_3} \ldots \xrightarrow{a_{n-1}} s_{n-1} \xrightarrow{a_n} s_n$. Denotamos $tr(\mathsf{P})$ al conjunto de trazas generadas por $\mathsf{P}$.

Sean $\mathsf{P}$ y $\mathsf{Q}$ sistemas. Consideremos el conjunto de trazas generado por cada uno de ellos. El sistema $\mathsf{Q}$ es más expresivo que el sistema $\mathsf{P}$ si el conjunto de trazas de $\mathsf{P}$ está incluido en el conjunto de trazas de $\mathsf{Q}$, esto es $tr(\mathsf{P}) \subseteq tr(\mathsf{Q})$.

Consideremos por ejemplo los sistemas P y P' de la figura 2.2, el conjunto de trazas generado por el sistema P es $tr(\mathsf{P}) = \{\varepsilon, a, ab, ac\}$, mientras que el generado por el sistema P' es $tr(\mathsf{P'}) = \{\varepsilon, a, ab, ac, ad\}$. Luego, $tr(\mathsf{P}) \subseteq tr(\mathsf{P'})$.

Si observamos detenidamente la figura 2.2, si realizamos la acción $a$ en el sistema P, podemos luego elegir realizar la acción $b$ o $c$. En el sistema P', en cambio, dependiendo de qué acción $a$ realicemos, quedaremos en una situación en la que sólo

Figura 2.2: LTS P y P’, donde P’ tiene mayor expresividad que P.

podemos realizar la acción $b$, o en su defecto en la que sólo podemos realizar la acción $c$. En otras palabras, el sistema P’ tiene menos expresividad respecto del sistema P. Esta diferencia no es detectada a nivel de trazas, lo cual nos lleva a buscar otra forma de describir las posibles ejecuciones de un sistema. El instrumento fundamental para comparar comporamiento de LTS es la noción de *simulación*, y sus derivados, como la bisimulación. La idea bajo el concepto de simulación es que dado que P decide realizar una acción, entonces el sistema P’ puede imitarlo realizando la misma acción. Si el sistema P’ puede imitar toda acción realizada por P, entonces diremos que P’ simula a P o que P es simulado por P’.

La noción de simulación no es más que una relación particular sobre el conjunto de estados de los sistemas involucrados.

**Definición 2.2.1.** Una *simulación fuerte (strong simulation)* es una relación $\mathsf{r} \in S \times S'$ entre dos LTS, $\mathsf{P} = (S, L, \longrightarrow, s_0)$ y $\mathsf{P}' = (S', L', \longrightarrow', s_0')$, donde se cumple que $(s_0, s_0') \in \mathsf{r}$ y para todo $s_1, s_2 \in \mathit{Reach}(\mathsf{P})$, $s_1' \in \mathit{Reach}(\mathsf{P}')$ y $l \in S$,

$$s_1 \xrightarrow{l} s_2 \text{ y } (s_1, s_1') \in \mathsf{r} \text{ implica,}$$

$$s_1' \xrightarrow{l}{}' s_2' \text{ y } (s_2, s_2') \in \mathsf{r}, \text{ para algún } s_2' \in \mathit{Reach}(\mathsf{P}').$$

Estamos interesados además en la equivalencia de sistemas. A nivel de trazas, podemos observarlo mediante el conjunto de trazas que cada sistema genera. Si ambos sistemas generan el mismo, podemos decir que son equivalentes. Pero, al igual que en el caso anterior, estamos interesados en poder diferenciar la expresividad de los sistemas en cuanto a la libertad de elección de eventos a realizar, y es por eso que un método observacional no nos alcanza. Debemos tener en cuenta la estructura de los sistemas. El concepto de bisimulación nos ayudará con esta descripción. La idea se basa en que dados dos sistemas P y P’, queremos que, cada vez que uno realice una acción, el otro pueda imitarlo. En este caso, el rol de qué sistema realiza o imita la acción cambia durante la ejecución.

En la siguiente definición, usaremos $\breve{\mathsf{r}}$ para denotar la conversa, o transpuesta, de una relación binaria $\mathsf{r}$.

**Definición 2.2.2.** Una *bisimulación fuerte (strong bisimulation)* es una relación $\mathsf{r} \in S \times S'$ entre dos LTS, $\mathsf{P} = (S, L, \longrightarrow, s_0)$ y $\mathsf{P}' = (S', L', \longrightarrow', s_0')$, tal que cumple que $\mathsf{r}$ es una simulación fuerte entre $\mathsf{P}$ y $\mathsf{P}'$ y $\breve{\mathsf{r}}$ es una simulación fuerte entre $\mathsf{P}'$ y $\mathsf{P}$.

Consideremos los ejemplos de la figuras 2.3 y 2.4. Veamos un caso donde se da una bisimulación fuerte entre los sistemas y uno en el que no.

Hasta ahora, la acción $\tau$ no ha recibido un tratamiento diferenciado del resto de las acciones en las definiciones de simulación y bisimulación. En otras palabras, $\tau$

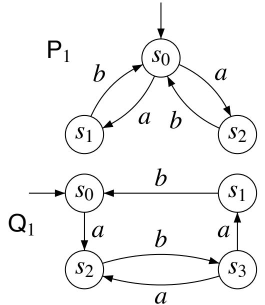



Figura 2.3: Existe una bisimulación fuerte entre $\mathsf{P}_1$ y $\mathsf{Q}_1$. Esto es, hay al menos una relación tal que satisface nuestra definición. Una posible relación es, $\mathsf{r} = \{(s_0, s_0), (s_0, s_3), (s_1, s_1), (s_1, s_2), (s_2, s_2)\}$.

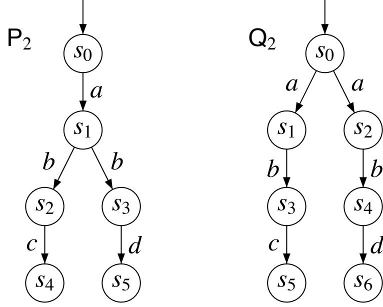



Figura 2.4: No existe una bisimulación fuerte entre $\mathsf{P}_2$ y $\mathsf{Q}_2$. Tenemos que $(s_0, s'_0) \in \mathsf{r}$, $\mathsf{Q}_2$ decide hacer $a$, $\mathsf{P}_2$ lo imita con la única acción $a$ disponible. Luego $(s_1, s'_1) \in \mathsf{r}$. Ahora $\mathsf{P}_2$ decide realizar $b$ y luego $d$, cosa que $\mathsf{Q}_2$ no puede imitar.

ha recibido el mismo tratamiento que cualquier otra acción observable en nuestros sistemas. ¿Cuál es el rol de las acciones no observables en la (bi)simulación? Nos interesa modelar una noción de bisimulación teniendo en cuenta que $\tau$ es una acción no observable o silenciosa. Esto significa que si un sistema realiza una acción $\tau$, el otro sistema no ve este comportamiento y viceversa. Una forma natural de observarlo es mediante las ya definidas funciones $\Longrightarrow$ y $\longrightarrow$. La idea es utilizar $\Longrightarrow$ en lugar de $\longrightarrow$ en nuestras definiciones anteriores. Esto nos va a permitir modelar la acción visible en nuestro sistema cuando ocurra una secuencia de acciones silenciosas y en medio de éstas, la acción visible en el otro sistema. Un ejemplo es el de la figura 2.5, en el cual podemos observar que cuando la acción $b$ ocurra en el sistema P, el sistema P' la va a imitar realizando las acciones $\tau$ y $b$.

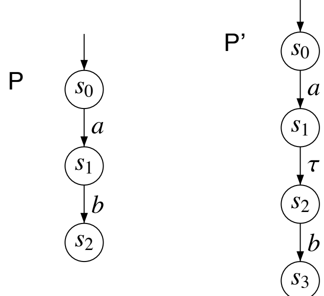



Figura 2.5: LTS P y P', donde P' imita a P con la ayuda de acciones no observables.

**Definición 2.2.3.** Una *simulación débil (weak simulation)* es una relación $\mathsf{r} \in S \times S'$

entre dos LTS, $\mathsf{P} = (S, L, \longrightarrow, s_0)$ y $\mathsf{P}' = (S', L', \longrightarrow', s'_0)$ donde se cumple que $(s_0, s'_0) \in \mathsf{r}$ y para todo $s_1, s_2 \in Reach(\mathsf{P})$, $s'_1 \in Reach(\mathsf{P}')$ y $l \in S$,

$$s_1 \xrightarrow{l} s_2 \text{ y } (s_1, s'_1) \in \mathsf{r} \text{ implica}$$
$$s'_1 \overset{\hat{l}}{\Longrightarrow}{}' s'_2 \text{ y } (s_2, s'_2) \in \mathsf{r}, \text{ para algún } s'_2 \in Reach(\mathsf{P}').$$

**Definición 2.2.4.** Una *bisimulación débil (weak bisimulation)* es una relación $\mathsf{r} \in S \times S'$ entre dos LTS, $\mathsf{P} = (S, L, \longrightarrow, s_0)$ y $\mathsf{P}' = (S', L', \longrightarrow', s'_0)$, tal que cumple que $\mathsf{r}$ es una simulación débil entre $\mathsf{P}$ y $\mathsf{P}'$ y $\breve{\mathsf{r}}$ es una simulación débil entre $\mathsf{P}'$ y $\mathsf{P}$.

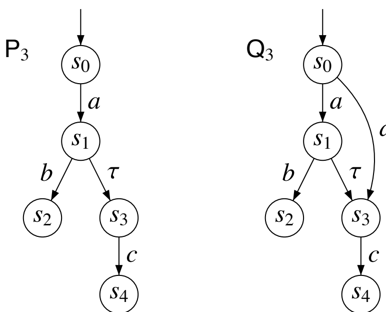



Figura 2.6: Existe una bisimulación débil entre $\mathsf{P}_3$ y $\mathsf{Q}_3$. Esto es, hay al menos una relación tal que satisface nuestra definición. Una posible relación es, $\mathsf{r} = \{(s_0, s_0), (s_1, s_1), (s_2, s_2), (s_3, s_3), (s_4, s_4)\}$.

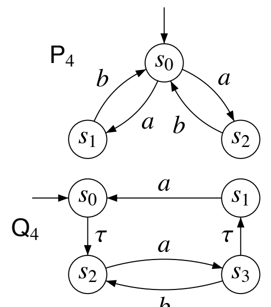



Figura 2.7: No existe una bisimulación débil entre $\mathsf{P}_4$ y $\mathsf{Q}_4$, ya que $\mathsf{P}_4$ decide hacer $a$, luego $\mathsf{Q}_4$ lo imita con $a$, por lo tanto $(s_1, s_3) \in \mathsf{r}$. Ahora $\mathsf{Q}_4$ realiza $\tau$, por lo que $(s_1, s_3) \in \mathsf{r}$ ya que $\mathsf{P}_4$ no observa esta acción. Luego, $\mathsf{Q}_4$ decide hacer $a$, y $\mathsf{P}_4$ no puede imitarlo.

Consideremos la figura 2.7, vemos un ejemplo donde dos LTS no son bisimilarmente débiles, pero en la figura 2.6, estos LTS son bisimilarmente débiles. Con la intención de diferenciar el comportamiento de sistemas desde un punto de vista más estructural, introducimos la noción de (bi)simulación ramificada. Intuitivamente, la idea es que dos sistemas sean bisimilares bajo esta nueva relación de bisimulación "ramificada" si y sólo si éstos poseen la misma ramificación.

**Definición 2.2.5.** Una *simulación ramificada (branching simulation)* es una relación $\mathsf{r} \in S \times S'$ entre dos LTS, $\mathsf{P} = (S, L, \longrightarrow, s_0)$ y $\mathsf{P}' = (S', L', \longrightarrow', s'_0)$ donde se cumple que $(s_0, s'_0) \in \mathsf{r}$ y para todo $s_1, s_2 \in Reach(\mathsf{P})$, $s'_1 \in Reach(\mathsf{P}')$ y $l \in S$,

$$s_1 \xrightarrow{l} s_2 \text{ y } (s_1, s'_1) \in \mathsf{r} \text{ implica}$$
$$s'_1 \overset{\hat{\tau}}{\Longrightarrow}{}' s'_2 \overset{\hat{l}}{\longrightarrow}{}' s'_3 \text{ y } (s_1, s'_2), (s_2, s'_3) \in \mathsf{r}, \text{para algún } s'_2, s'_3 \in Reach(\mathsf{P}').$$

**Definición 2.2.6.** Una *bisimulación ramificada (branching bisimulation)* es una relación $\mathsf{r} \in S \times S'$ entre dos LTS, $\mathsf{P} = (S, L, \longrightarrow, s_0)$ y $\mathsf{P}' = (S', L', \longrightarrow', s'_0)$, tal que cumple que $\mathsf{r}$ es una simulación ramificada entre $\mathsf{P}$ y $\mathsf{P}'$ y $\breve{\mathsf{r}}$ es una simulación ramificada entre $\mathsf{P}'$ y $\mathsf{P}$.

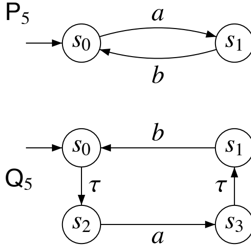



Figura 2.8: Existe al menos una relación de bisimulación ramificada entre $\mathsf{P}_5$ y $\mathsf{Q}_5$. Una posible relación es $\mathsf{r} = \{(s_0, s_0), (s_0, s_2), (s_1, s_1), (s_1, s_3)\}$.

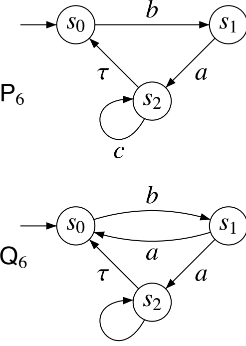



Figura 2.9: No existe una bisimulación ramificada entre $\mathsf{P}_6$ y $\mathsf{Q}_6$ ya que supongamos que $\mathsf{P}_6$ realiza $b$, luego $\mathsf{Q}_6$ lo imita con $b$, luego $\mathsf{P}_6$ hace $a$ y $\mathsf{Q}_6$ lo imita nuevamente. Ahora los roles cambian y $\mathsf{Q}_6$ decide hacer $c$, y $\mathsf{P}_6$ no lo puede imitar ya que se encuentra en el estado $s_0$ y dicha acciones no está habilitada.

Notemos que una relación que es (bi)simulación fuerte también es (bi)simulación ramificada, y similarmente una (bi)simulación ramificada es una (bi)simulación débil. Sin embargo las recíprocas no son ciertas. Consideremos el ejemplo dado de la figura 2.8 entre estos LTS se satisface la bisimulación ramificada, pero no se satisface la bisimulación fuerte. Por otro lado, en el ejemplo dado por la figura 2.6 tenemos que se satisface la bisimulación débil, pero no la bisimulación branching.

## 2.3 Sistemas de Transiciones Modales

Los LTS son adecuados como interpretación del comportamiento de sistemas cuando uno cuenta con información *total* sobre los mismos. Por el contrario, cuando uno sólo tiene información parcial sobre el comportamiento de un sistema, la versión tradicional de LTS deja de ser adecuada. Existen numerosas razones por las cuales es importante poder modelar sistemas parciales. Entre éstas está el hecho de que, en etapas tempranas del proceso de desarrollo, es usual conocer sólo parcialmente el comportamiento del sistema a desarrollar. Contar con modelos formales en estas situaciones nos permite razonar sobre varios aspectos ligados a estos modelos parciales, tales como la correcta correspondencia de implementaciones (modelos totales especificados, por ejemplo, mediante LTS) con respecto a un modelo parcial, o las formas válidas de incorporación de detalles (complementación de un modelo parcial con información

adicional sobre su comportamiento) en un modelo parcial, es decir, relaciones de refinamiento entre éstas.

Una de las variantes de LTS que incorpora cierta noción de incertidumbre, necesaria para modelar sistemas parciales, son los MTS, que describimos formalmente a continuación. El lector interesado en más detalles al respecto, puede consultar [LT88].

**Definición 2.3.1.** Un *sistema de transiciones modales (MTS - modal transition systems)* es una estructura $\mathsf{P} = (S, L, \longrightarrow_r, \longrightarrow_p, s_0)$ donde $\longrightarrow_r \subseteq \longrightarrow_p$, $\mathsf{P} = (S, L, \longrightarrow_r, s_0)$ es un LTS representado por transiciones requeridas del sistema y $\mathsf{P} = (S, L, \longrightarrow_p, s_0)$ es un LTS representado por transiciones posibles (pero no necesariamente requeridas) del sistema.

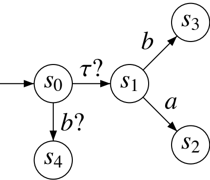



Figura 2.10: Ejemplo de un sistema de transiciones modales

Un sistema de transiciones modales $\mathsf{P} = (S, L, \longrightarrow_r, \longrightarrow_p, s_0)$ permite modelar el comportamiento de un sistema de la siguiente manera:

- el conjunto $S$ representa, al igual que para LTS, el espacio de todos los estados posibles que el sistema puede asumir. De igual manera, el estado inicial $s_0 \in S$ representa el estado a partir del cual se inicia la ejecución en el sistema.
- las etiquetas en $L$ representan las acciones que el sistema puede ejecutar, o los eventos a los que éste puede responder (el evento $\tau$ es simplemente utilizado para modelar acciones privadas del sistema, para las cuales no se quiere asociar eventos observables, al igual que en LTS).
- las transiciones en $\longrightarrow_r$ representan el flujo de ejecución del sistema que es *requerido*, es decir que toda realización de $\mathsf{P}$ debe admitir. Nuevamente, el hecho de que $\longrightarrow_r$ es una relación indica que $\longrightarrow_r$ puede codificar no determinismo.
- las transiciones en $\longrightarrow_p$ representan el flujo de ejecución del sistema que es *probable*, es decir que cada realización de $\mathsf{P}$ tiene la libertad de admitir o no. En cierto sentido, las transiciones probables limitan el grado de libertad con que el modelo parcial asociado a $\mathsf{P}$ puede completarse.

En la Figura 2.10 podemos ver el conjunto de estados $S = \{s_0, s_1, s_2, s_3, s_4\}$, el de acciones $L = \{\tau, a, b\}$ y el conjunto de transiciones permitidas. Las transiciones probables están indicadas con un signo de interrogación. Asi, por ejemplo, si estamos en el estado $s_0$ podemos movernos al estado $s_1$ realizando la acción $\tau?$, que es una acción probable en nuestro sistema.

## 2.4 Refinamientos

Hemos visto que la noción de traza, o ejecución, no es suficiente como elemento para dar semántica a LTS. Como consecuencia de esto, han surgido diferentes nociones de (bi)simulación, algunas de las cuales hemos reproducido. Así como existen

buenas razones prácticas para contar con mecanismos para comparar LTS respecto de su semántica, estas mismas razones se aplican al caso de especificaciones parciales a través de MTS. En otras palabras, es importante contar con nociones similares a las de (bi)simulación vistas, pero que tengan en cuenta la parcialidad de MTS. La noción que introduciremos a continuación es la que permite incorporar detalle a un MTS, esencialmente a través de la “transformación” de transiciones probables en requeridas, o la eliminación de transiciones probables.

Dado que un MTS cuenta con dos relaciones de transición sobrecargamos la notación definida y denotamos por *Reach*(P) el conjunto de estados alcanzables en $\mathsf{P} = (S, L, \longrightarrow_r, \longrightarrow_p, s_0)$ a partir de $s_0$ mediante $(\longrightarrow_r \cup \longrightarrow_p)$.

**Definición 2.4.1.** Un *refinamiento fuerte (strong refinement)* es una relación $\mathsf{r} \in S \times S'$ entre dos MTS, $\mathsf{P} = (S, L, \longrightarrow_r, \longrightarrow_p, s_0)$ y $\mathsf{P}' = (S', L', \longrightarrow'_r, \longrightarrow'_p, s'_0)$ donde se cumple que:

$(s_0, s'_0) \in \mathsf{r}$ y para todo $s_1, s_2 \in Reach(\mathsf{P})$, $s'_1 \in Reach(\mathsf{P}')$ y $l \in L$,

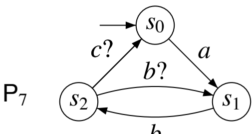
$$(s_1, s'_1) \in \mathsf{r} \text{ y } s1 \xrightarrow{l}_r s_2 \text{ implica}$$

$$s'_1 \xrightarrow{l}{}'_r s'_2 \text{ y } (s_2, s'_2) \in \mathsf{r}, \text{ para algún } s'_2 \in Reach(\mathsf{P})' \text{ y}$$

para todo $s'_1, s'_2 \in Reach(\mathsf{P}')$, $s_1 \in Reach(\mathsf{P})$ y $l \in L'$,

$$(s'_1, s_1) \in \Upsilon \text{ y } s'_1 \xrightarrow{l}{}'_p s'_2 \text{ implica}$$

$$s_1 \xrightarrow{l}_p s_2 \text{ y } (s'_2, s_2) \in \Upsilon, \text{ para algún } s_2 \in Reach(\mathsf{P}).$$



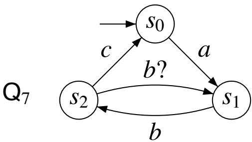




Figura 2.11: Existe un refinamiento fuerte entre $\mathsf{P}_7$ y $\mathsf{Q}_7$. La relación es $\mathsf{r} = \{(s_0, s_0), (s_1, s_1), (s_2, s_2)\}$.
Pero no existe un refinamiento fuerte entre $\mathsf{Q}_7$ y $\mathsf{P}_7$. Veamos que $\mathsf{Q}_7$ realiza, $a$, $b$ y $c$. Luego, $\mathsf{P}_7$ lo imita con $a$, $b$ y no puede realizar $c$ ya que no es una transición requerida.

En la figura 2.11 podemos observar cómo $\mathsf{P}_7$ es un refinamiento de $\mathsf{Q}_7$, y $\mathsf{Q}_7$ no es un refinamiento de $\mathsf{P}_7$.

Al igual que cuando introducimos bisimulación fuerte en LTS, no hemos dicho nada hasta ahora de las acciones $\tau$ en MTS. Estamos interesados, en considerar estas acciones como no observables. Introducimos la noción de refinamiento débil para poder describir este comportamiento.

**Definición 2.4.2.** Un *refinamiento débil (weak refinement)* es una relación $\mathsf{r} \in S \times S'$ entre dos MTS, $\mathsf{P} = (S, L, \longrightarrow_r, \longrightarrow_p, s_0)$ y $\mathsf{P}' = (S', L', \longrightarrow'_r, \longrightarrow'_p, s'_0)$ se cumple que:

$(s_0, s'_0) \in \mathsf{r}$ y para todo $s_1, s_2 \in Reach(\mathsf{P})$, $s'_1 \in Reach(\mathsf{P}')$ y $l \in S$,

$$(s_1, s_1') \in \mathsf{r} \text{ y } s_1 \xrightarrow{l}_{r} s_2 \text{ implica}$$

$$s_1' \overset{\hat{l}}{\Longrightarrow}{}'_{r} s_2' \text{ y } (s_2, s_2') \in \mathsf{r}, \text{ para algún } s_2' \in \mathit{Reach}(\mathsf{P}') \text{ y}$$

para todo $s_1', s_2' \in \mathit{Reach}(\mathsf{P}')$, $s_1 \in \mathit{Reach}(\mathsf{P})$ y $l \in S'$,

$$(s_1', s_1) \in \breve{\mathsf{r}} \text{ y } s_1' \xrightarrow{l}{}'_{r} s_2' \text{ implica}$$

$$s_1 \overset{\hat{l}}{\Longrightarrow}_{r} s_2 \text{ y } (s_2', s_2) \in \breve{\mathsf{r}}, \text{ para algún } s_2 \in \mathit{Reach}(\mathsf{P}).$$

Analicemos las figuras 2.12 y 2.13; veamos dos ejemplos los cuales en uno se satisface la propiedad de refinamiento débil, mientras que en el otro no.

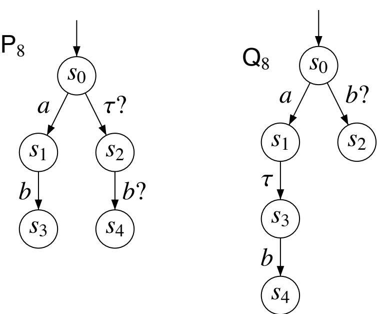



Figura 2.12: Existe un refinamiento débil entre $\mathsf{P}_8$ y $\mathsf{Q}_8$. Una relación que satiface es, $\mathsf{r} = \{(s_0, s_0), (s_1, s_1), (s_1, s_3), (s_3, s_4), (s_4, s_2)\}$.

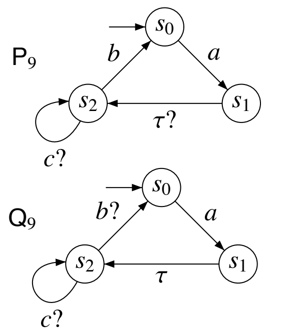



Figura 2.13: No existe un refinamiento débil entre $\mathsf{P}_9$ y $\mathsf{Q}_9$, notemos que $\mathsf{P}_9$ realiza $a$, entonces $\mathsf{Q}_9$ la imita con $a$, luego si $\mathsf{P}_9$ realiza $\tau$, $\mathsf{Q}_9$ se queda quieta, porque si hace $\tau$ entonces habilita $b$ que como es requerida, $\mathsf{Q}_9$ no la puede imitar. Ahora si $\mathsf{Q}_9$ decide hacer $c?$, entonces $\mathsf{P}_9$ la tiene que imitar con $\tau$ y $c$, lo cual la habilita para hacer $b$ y $\mathsf{Q}_9$ no la puede imitar porque $b$ es probable en $\mathsf{Q}_9$

## 2.5 Implementaciones

Hemos descripto dos tipos de sistemas; los LTS que permiten descripciones totales y los MTS que permiten descripciones parciales de los sistemas. Ahora estamos interesados en relacionar ambas descripciones. Por ejemplo queremos poder corroborar si una descripción total de un sistema es una representación de una descripción parcial del mismo, o poder corroborar que forma puede tener mi sistema total, a partir de sólo una descripción parcial. En otras palabras queremos ver si dada una especificación de un sistema, hemos llegado a su correcta implementación o si dada una implementación del sistema, ésta es posible a partir de dicha especificación. Donde la especificación es una descripción parcial. Basándonos en los conceptos anteriores, tomaremos una descripción parcial, esto es un MTS y una descripción total, un LTS; y verificaremos si entre éstos existe alguna relación de implementación.

Comenzamos con la definición de implementación fuerte.

**Definición 2.5.1.** Una *implementación fuerte (strong implementation)* es una relación $\mathsf{r} \in S \times S'$ entre un MTS $\mathsf{P} = (S, L, \longrightarrow_r, \longrightarrow_p, s_0)$ y un LTS $\mathsf{P}' = (S', L', \longrightarrow, s'_0)$ el cual es un caso particular de refinamiento fuerte en el cual extendemos el LTS $\mathsf{P}'$ a un MTS $\mathsf{P}' = (S', L', \longrightarrow, \longrightarrow, s'_0)$.

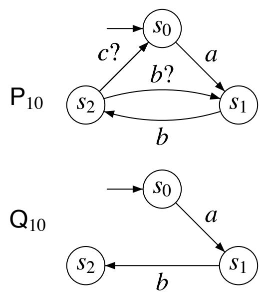



Figura 2.14: Existe una implementación fuerte entre $\mathsf{P}_{10}$ y $\mathsf{Q}_{10}$. La relación es $\mathsf{r} = \{(s_0, s_0), (s_1, s_1), (s_2, s_2)\}$.

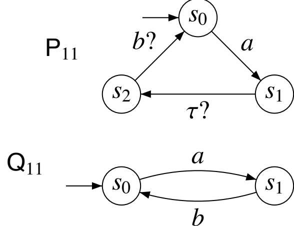



Figura 2.15: No existe una implementación fuerte entre $\mathsf{P}_{11}$ y $\mathsf{Q}_{11}$. La acción $\tau$ es una acción observable para la implementación fuerte.

Al igual que en los casos anteriores mostramos un caso donde se da una implementación fuerte entre un MTS y un LTS, y uno en el que no. Notemos que, por el momento no hemos dicho nada de la acción $\tau$. Ahora estamos interesados nuevamente en considerar a las acciones $\tau$ como acciones sileciosas, por tal motivo introducimos la noción de implementación débil.

**Definición 2.5.2.** Una *implementación débil (weak implementation)* es una relación $\mathsf{r} \in S \times S'$ entre un MTS $\mathsf{P} = (S, L, \longrightarrow_r, \longrightarrow_p, s_0)$ y un LTS $\mathsf{P}' = (S', L', \longrightarrow, s'_0)$ el cual es un caso particular de refinamiento fuerte en el cual extendemos el LTS $\mathsf{P}'$ a un MTS $\mathsf{P}' = (S', L', \longrightarrow, \longrightarrow, s'_0)$.

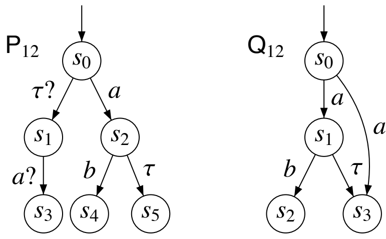



Figura 2.16: Existe una implementación débil entre $\mathsf{P}_{12}$ y $\mathsf{Q}_{12}$. La relación de implementación débil es, $\mathsf{r} = \{(s_0, s_0), (s_2, s_1), (s_4, s_2), (s_5, s_3)\}$.

Vemos en la figura 2.16 y la figura 2.17 un caso en el que se da la implementación weak y un caso en el que no. Por último introducimos la noción de implementación ramificada dada por Fishbein en [Fis06]. La idea es además de considerar a $\tau$ como acciones no observables o silenciosas, poder observar el comportamiento desde un punto de vista más estructural.

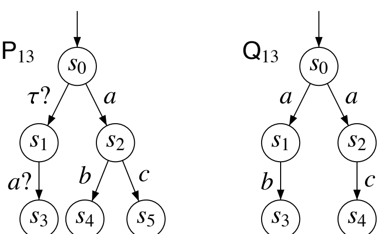



Figura 2.17: No existe una implementación débil entre $\mathsf{P}_{13}$ y $\mathsf{Q}_{13}$ debido a que, supongamos que $\mathsf{P}_{13}$ decide hacer $a$, luego $\mathsf{Q}_{13}$ decide imitarlo y hace $a$ llegando al estado $s_2$. $\mathsf{P}_{13}$ ahora hace $b$ y $\mathsf{Q}_{13}$ no puede imitarlo, ya que no tiene habilitada dicha acción.

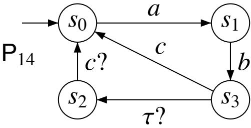
**Definición 2.5.3.** Una *implementación ramificada (branching implementation)* es una relación $\mathsf{r} \in S \times S'$ entre un MTS $\mathsf{P} = (S, L, \longrightarrow_r, \longrightarrow_p, s_0)$ y un LTS $\mathsf{P'} = (S', L', \longrightarrow, s'_0)$ tal que,

$(s_0, s'_0) \in \mathsf{r}$ y para todo $s_1, s_2 \in \mathit{Reach}(\mathsf{P})$, $s'_1 \in \mathit{Reach}(\mathsf{P'})$ y $l \in S$,

$(s'_1, s_1) \in \mathsf{r}$, $s_1 \xrightarrow{l}_r s_2$ implica $s'_i \xrightarrow{\tau} s'_{i+1}$ y $(s_1, s'_{i+1}) \in \mathsf{r}, 1 \leq i+1 < n+1$ y $s'_n \xrightarrow{\hat{l}} s'_{n+1}$ y $(s_1, s'_{n+1}) \in \mathsf{r}$, para algún $s'_2, \ldots, s'_{n+1} \in \mathit{Reach}(\mathsf{P'})$.

para todo $s'_1, s'_2 \in \mathit{Reach}(\mathsf{P'})$ y $s_1 \in \mathit{Reach}(\mathsf{P})$ y $l \in S'$,

$$(s_1, s'_1) \in \breve{\mathsf{r}} \text{ y } s'_1 \xrightarrow{l} s'_2 \text{ implica } s_i \xrightarrow{\tau}_p s_{i+1} \text{ y } (s'_1, s_{i+1}) \in \breve{\mathsf{r}}, 1 \leq i+1 < n+1$$
$$\text{y } s_n \xrightarrow{\hat{l}}_p s_{n+1} \text{ y } (s'_1, s_{n+1}) \in \breve{\mathsf{r}}, \text{para algn } s_2, \ldots, s_{n+1} \in \mathit{Reach}(\mathsf{P}).$$



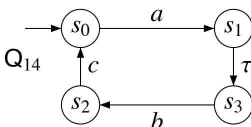




Figura 2.18: Existe una implementación ramificada entre $\mathsf{P}_{14}$ y $\mathsf{Q}_{14}$. Una relación que satisface es $\mathsf{r} = \{(s_0, s_0), (s_2, s_1), (s_4, s_2), (s_5, s_3)\}$

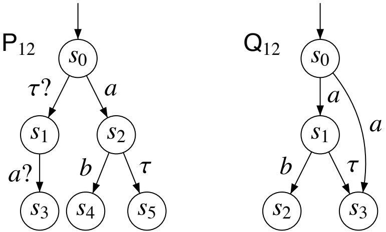



Figura 2.19: No existe una implementación ramificada entre $\mathsf{P}_{12}$ y $\mathsf{Q}_{12}$. Ya que como podemos observar son diferentes estructuras.

# 3 Modelado de MTS y LTS en Alloy

Como describimos anteriormente, la intención de este trabajo es utilizar SAT solving para el análisis de relaciones de refinamiento e implementación entre LTS y MTS. Utilizaremos para esto, como lenguaje de modelado intermedio, el lenguaje de especificaciones Alloy, que nos brinda construcciones al nivel de abstracción adecuado para modelar estos formalismos y las relaciones entre ellos.

Modelar LTS y MTS en Alloy nos permite además aprovechar algunos mecanismos de mejora al análisis basado en SAT, tales como la eliminación de modelos isomorfos [JJD98] y el aprovechamiento de información sobre cotas en la generación de fórmulas proposicionales para analizar mediante SAT solver.

La razón por la cual consideramos a Alloy en el nivel de abstracción adecuado para la especificación de LTS y MTS es que este lenguaje da a la noción de relación, fundamental en las definiciones de LTS y MTS, un rol central. Es también importante que la clausura transitiva, crucial en la definición de relaciones de simulación/refinamiento, es directamente soportada en Alloy. Las caracterizaciones que daremos a continuación son lo suficientemente claras como para que incluso el lector no familiarizado con Alloy las pueda comprender. Referimos sin embargo al Apéndice B para más detalles sobre el lenguaje.

## 3.1 Definición de Universos

Comenzaremos con la definición de los universos necesarios para LTS y MTS. Claramente necesitamos la definición de dominios para estados y etiquetas. Más aún, debemos exigir la existencia de $\tau$, la acción "silenciosa", como etiqueta válida. Para declarar dominios en Alloy, se utilizan signaturas:

```
abstract sig State {}
sig CState extends State {}

abstract sig Action {}
sig CAction extends Action {}
one sig Tau extends Action {}
```

En las definiciones anteriores, cada signatura define un dominio, y signaturas diferentes definen, en principio, dominios disjuntos. La excepción a esta regla la constituyen las signaturas relacionadas por extensión (en el módulo anterior, las signaturas `CState` y `State`, por ejemplo). Cuando una signatura extiende a otra, la primera define

un subdominio de la segunda. Además, los subdominios distintos de una signatura son disjuntos. Notemos la declaración de $\tau$ a través de la signatura Tau. Esto puede confundir al lector no familizarizado con Alloy, pues estamos utilizando signaturas para declarar tanto dominios como elementos (constantes) de los mismos. La razón de esto es que Alloy no permite definir constantes en los modelos: las constantes son parte de las interpretaciones de los modelos, y no de los modelos. Luego, cuando es necesario referirse a elementos particulares, éstos se modelan a través de signaturas unitarias, en las que se fuerza la existencia de exactamente un elemento mediante el modificador "one".

Finalmente, notemos el uso del modificador "abstract" en la declaración de algunas signaturas. Este modificador indica que la signatura no tiene elementos "propios", sino que su dominio asociado estará compuesto exclusivamente por la unión de los dominios de sus subsignaturas. El lector familiarizado con orientación a objetos seguramente observará una relación entre signaturas abstractas y extensión con las nociones de clase abstracta y herencia, repectivamente.

Con las signaturas definidas más arriba, los universos de LTS y MTS pueden caracterizarse directamente, nuevamente mediante signaturas, de la siguiente manera:

```
/* signature LTS: definition of labelled transition system.       */
/*                Actions and states of the LTS are implicitly    */
/*                those mentioned in init and trans.              */
/*                An LTS is then simply an initial state plus a   */
/*                transition relation.                            */
abstract sig LTS {
        init: one State,
        trans: (Action -> State) -> State
}

/* signature MTS: definition of modal transition system. Actions  */
/*                and states of the MTS are implicitly those      */
/*                mentioned in init and t.                        */
/*                An MTS is simply an LTS in which some of the    */
/*                transitions are marked as required              */
abstract sig MTS {
        init: one State,
        trans: (Action -> State) -> State,
        req: (Action -> State) -> State
}
{
        req in trans
}
```

Notemos que, al no considerar el conjunto de estados como parte de un LTS o MTS, algunos LTS o MTS no pueden ser representados. Consideremos por ejemplo el siguiente LTS:

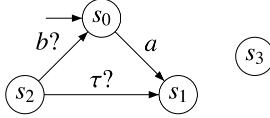



Figura 3.1: Ejemplo de un sistema de transiciones modales no conexo

Como el nodo $s_3$ no es inicial y no está involucrado en la relación de transición, no lo podríamos representar en nuestro modelo de LTS anterior. Sin embargo, esta pérdida no nos afecta para lo que estamos interesados en representar y analizar, y evitamos tener que definir un atributo extra en nuestros LTS y MTS.

En la definición de LTS y MTS también hemos utilizado signaturas, esta vez con atributos asociados. Estas signaturas, además de definir dominios declaran relaciones para cada uno de los distintos campos definidos, del perfil y aridad declaradas. Asi, por ejemplo, `req` es una relación con perfil `MTS` × `Action` × `State` × `State`, total y funcional en el primer argumento. La restricción "las transiciones requeridas son parte del conjunto global de transiciones" se fuerza mediante un hecho (fact) ligado a la signatura MTS.

Notemos que, en estas definiciones, las etiquetas son ubicadas al comienzo en las relaciones de transición, ya sean requeridas o probables. La razón de esto es simplemente para poder "proyectar"fácilmente la relación entre estados, es decir, hacerlas independientes de las etiquetas, mediante la composición relacional, uno de los operadores relacionales disponibles en Alloy.

## 3.2 Transiciones y funciones auxiliares

Con los LTS y MTS definidos, podemos intentar definir las nociones de simulación, bisimulación, refinamiento e implementación. Para simplificar la lectura de especificaciones, introducimos varias definiciones auxiliares que representan varios tipos de "transiciones". Estas definiciones son encapsuladas en predicados Alloy. Un predicado Alloy es simplemente una fórmula parametrizada (es decir, con variables libres), al estilo de los esquemas en Z [Dil94]. Los tipos de transiciones y funciones auxiliares son los siguientes:

- $s_1 \xrightarrow{a} s_2$

```
/* simple arrow: single transition between states                    */
pred arrow
          (t: (Action -> State) -> State, s1, s2: State, a: Action) {
              a -> s1 -> s2 in t
}
```

- $s_1 \xrightarrow{\hat{a}} s_2$

```
/* simple "roofed" arrow: single transition or silent step          */
pred arrowT
          (t: (Action -> State) -> State, s1, s2: State, a: Action) {
              a -> s1 -> s2 in t or (a = Tau  and s1 = s2)
}
```

- $s_1 \overset{a}{\Longrightarrow} s_2$

```
/* Double Arrow                                                     */
/* single transition preceded and succeded by potentially many      */
/*                                                     silent steps */
pred clTArrowClT
          (t: (Action -> State) -> State, s1, s2: State, a: Action) {
              s1 -> s2 in ((*(t[Tau])).(t[a]).(*(t[Tau])))
}
```

- $s_1 \overset{\hat{a}}{\Longrightarrow} s_2$

```
/* Double arrow                                                   */
/* Single transition or silent step , preceded and succeded by    */
/*                                    potentially many silent steps */
pred clTArrowTClT
        (t: (Action -> State) -> State , s1 , s2: State , a: Action) {
            s1 -> s2 in ((*(t[Tau])).(t[a]).(*(t[Tau])))
            or (a = Tau and s1 = s2)
}
```

- la siguiente función toma un conjunto de transiciones y lo complementa con todas las transiciones $\tau$ posibles. Será utilizado para modelar el predicado de simulación débil.

```
/* augTauTransition: "augments" a set of transitions with all      */
/*                                        possible tau transitions */
fun augTauTransition
      (t: (Action -> State) -> State):(Action -> State) -> set State {
          t + (Tau -> (iden & (State -> State)))
}
```

- la siguiente función toma un conjunto de transiciones y un conjunto de estados, y calcula la preimagen del conjunto de estados tomados respecto de la clausura reflexo-transitiva de las transiciones, complementadas con $\tau$:

```
/* preImageClTau: Computes the preimage of ~ set of states C,       */
/*                with respect to the reflexive-transitive closure */
/*                of a set of transitions complemented with silent */
/*                                                           steps */
fun preImageClTau
 (t: (Action -> State) -> State , C: set State): set(State -> State) {
     *((t[Tau]) & (State -> C))
}
```

En las definiciones anteriores hicimos uso de varios elementos sintácticos de Alloy, que explicaremos a continuación. Utilizando las variables y constantes relacionales declaradas como signaturas, campos de signaturas y parámetros de predicados, podemos formar términos relacionales. Podemos hacerlo usando el producto (->), unión (+), intersección (&), composición (·), conversa (~), clausura transitiva (^) y clausura reflexo-transitiva (∗). Con éstos podemos construir fórmulas atómicas mediante inclusión relacional (*in*) e igualdad relacional (=). Los conectivos (! es la negación, *and* la conjunción, *or* la dijunción) y los cuantificadores (*all* y *some* son los cuantificadores universal y existencial, respectivamente) nos permiten formar fórmulas compuestas. Es importante mencionar que todos los términos en Alloy denotan relaciones; por esto, los elementos se representan mediante singletones, es decir, relaciones unarias de cardinal uno (así, la inclusión relacional "in" puede usarse para representar pertenencia). Recomendamos al lector consultar el Apéndice B para más detalle sobre la sintaxis y semántica de Alloy, o referirse a [Jac06].

Nótese la importancia de la clausura reflexo-transitiva en la definición de varias de las transiciones definidas. Es importante hacer notar que, sin el operador relacional de clausura, nos sería imposible definir varias de estas transiciones dado que Alloy es esencialmente un lenguaje de primer orden.

## 3.3 Relaciones entre LTS y MTS

Definidos los LTS, MTS y funciones auxiliares, estamos en condiciones de modelar las definiciones de bisimulación, refinamiento e implementación introducidas anteriormente. La manera de modelarlas en Alloy es mediante predicados, de la misma forma que utilizamos éstos para definir los tipos de transiciones.

### 3.3.1 Simulaciones y Bisimulaciones

- Simulación Fuerte

```
/* States that r is a strong simulation between LTS l1 and LTS l2. */
pred StrongSimulation (r: State -> State, l1, l2: LTS) {
        all a: Action | (~r).(l1.trans[a]) in (l2.trans[a]).(~r)
}
```

- Bisimulación Fuerte

```
/* States that r is a strong bisimulation between LTS l1 and l2      */
pred StrongBisimulation (r: State -> State, l1, l2: LTS) {
        StrongSimulation [r, l1, l2] and
        StrongSimulation [~r, l2, l1]
}
```

- Simulación Ramificada

```
/* States that r is a branching simulation between LTS l1 and l2.   */
pred BranchingSimulation (r: State -> State, l1, l2: LTS) {
        all s1, s2, s3: State | all a: Action |
            (arrow[l1.trans, s1, s2, a] and s1 -> s3 in r)
            implies
            (some s4, s5: State |
                s3 -> s4 in (preImageClTau[l2.trans, s1.r]) and
                arrowT[l2.trans, s4, s5, a] and
                s1 -> s4 + s2 -> s5 in r
            )
}
```

- Bisimulación Ramificada

```
/* States that r is a branching bisimulation between LTS l1 and l2.*/
pred BranchingBisimulation (r: State -> State, l1, l2: LTS) {
         BranchingSimulation [r, l1, l2] and
         BranchingSimulation [~r, l2, l1]
}
```

- Simulación Débil

```
/* States that r is a weak simulation between LTS l1 and LTS l2.    */
pred WeakSimulation (r: State -> State, l1, l2: LTS) {
        all a: Action | (~r).(l1.trans[a]) in
           ((*(l2.trans[Tau])).((augTauTransition[l2.trans])[a]).
                                               (*(l2.trans[Tau])))).(~r)
}
```

- Bisimulación Débil

```
/* States that r is a weak bisimulation between LTS l1 and LTS l2. */
pred WeakBisimulation (r: State -> State , l1, l2: LTS) {
        WeakSimulation [r, l1, l2] and
        WeakSimulation [~r, l2, l1]
}
```

### 3.3.2 Refinamientos

- Refinamiento Fuerte

```
/* States that r is a strong refinement between MTS m1 and MTS m2. */
pred StrongRefinement (r: State -> State, m1, m2: MTS) {
        all a: Action | (~(r)).(m1.req[a]) in m2.req[a].(~(r))
        all a: Action | (~(r)).(m2.trans[a]) in m1.trans[a].(~(r))
}
```

- Refinamiento Débil

```
/* States that r is a weak refinement between MTS m1 and MTS m2.   */
pred WeakRefinement (r: State -> State, m1, m2: MTS) {
        all a: Action | (~r).(m1.req[a]) in
           ((*(m2.req[Tau])).((augTauTransition[m2.req])[a]).
                                          (*(m2.req[Tau]))).(~r)
        all a: Action | (~r).(m2.trans[a]) in
           ((*(m1.trans[Tau])).((augTauTransition[m1.trans])[a]).
                                          (*(m1.trans[Tau]))).(~r)
}
```

### 3.3.3 Implementaciones

- Implementación Fuerte

```
/* States that r is a strong implementation between MTS m and l     */
pred StrongImplementation (r: State -> State, m: MTS, l: LTS) {
        all a: Action | (~(r)).(m.req[a]) in l.trans[a].(~(r))
        all a: Action | (~(r)).(l.trans[a]) in m.trans[a].(~(r))
}
```

- Implementación Débil

```
/* States that r is a weak implementation between MTS m and LTS l. */
pred WeakImplementation (r: State -> State, m: MTS, l: LTS) {
        all a: Action | (~r).(m.req[a]) in
   ((*(l.trans[Tau])).((augTauTransition[l.trans])[a]).
                                          (*(l.trans[Tau]))).(~r)
        all a: Action | (~r).(l.trans[a]) in
   ((*(m.trans[Tau])).((augTauTransition[m.trans])[a]).
                                          (*(m.trans[Tau]))).(~r)
}
```

- Implementación Ramificada

```
/* States that r is a branching implementation between MTS m and l.*/
pred BranchingImplementation (r: State -> State, m: MTS, l: LTS) {
        all s1, s2: State | all a: Action | all s3 : State |
             arrow[m.req, s1, s2, a] and s1 -> s3 in r
             implies
             (some s4, s5: State |
                 s3 -> s4 in (preImageClTau[l.trans, s1.r]) and
```

```
            arrowT[l.trans, s4, s5, a] and s2 -> s5 in r
        )
    all s1, s2: State | all a: Action | all s3 : State |
        arrow[l.trans, s1, s2, a]  and s1 -> s3 in ~r
        implies
        (some s4, s5: State |
            s3 -> s4 in (preImageClTau[m.trans, s1.(~r)]) and
            arrowT[m.trans, s4, s5, a] and s2 -> s5 in ~r
        )
}
```

## 3.4 Análisis de algunas propiedades elementales

Una de las características más importantes de Alloy es que, además de poseer una sintaxis y semántica simple y precisa, provee un mecanismo de análisis completamente automático. Esto evita que el desarrollador necesite manipular expresiones matemáticas en las tareas de análisis (aunque claramente necesitará manipularlas al describir formalmente sus modelos en el lenguaje). El mecanismo de análisis de Alloy se basa en SAT solving, y su facilidad de aplicación (brindada por su completa automatización) tiene un costo importante en el alcance de las respuestas que nos brindan. El mecanismo funciona, básicamente, de la siguiente manera: dada una especificación *Spec*, el analizador acepta fórmulas de la siguiente naturaleza:

- corridas (runs), que se interpreta como escenarios o casos particulares de *Spec*.
- aserciones (assertions), que se interpretan como potenciales consecuencias de *Spec*.

En el caso de las corridas, el analizador intenta construir modelos de

*Spec and run*

(donde run es la corrida provista), mientras que en el caso de las aserciones el analizador busca construir modelos de

*Spec and (not assert)*

(donde assert es la aserción provista).

Para realizar esta búsqueda de modelos de manera automática, se requiere contar con un procedimiento de decisión para la lógica relacional (la lógica subyacente a Alloy). Sin embargo, la lógica relacional es una extensión de la lógica de primer orden, y por lo tanto, no existen algoritmos de decisión para la misma. Por este motivo, el analizador requiere que el usuario provea *cotas* para los dominios. El analizador nos dará una respuesta positiva si encuentra un modelo de la fórmula correspondiente dentro de las cotas dadas; el analizador dará una respuesta negativa en el caso en el cual la fórmula sea *insatisfactible* dentro de las cotas provistas. Por supuesto, que una fórmula no tenga modelos dentro de las cotas provistas no garantiza su insatisfactibilidad general, pues podría tener modelos si ampliamos las cotas. El Alloy Analyzer, la herramienta de análisis de Alloy, entra entonces en la categoría de *model finders* (como MACE [McC03] y PARADOX [Cla03]). La forma en la cual realiza la búsqueda es mediante una traducción de las fórmulas Alloy (con las correspondientes cotas) a una fórmula proposicional, y el uso de SAT solver para comprobar su satisfactibilidad. La utilidad de este mecanismo es justificada por la llamada *hipótesis del alcance pequeño* (small scope hypothesis), que dice básicamente que en la práctica, cuando nuestras fórmulas

corresponden a especificaciones de software, si una propiedad tiene contraejemplos, en general los tiene de tamaño pequeño [Jac06].

El lector puede encontrar más detalles al respecto en el Apéndice B.

Con nuestra especificación de LTS y MTS, y las definiciones de las diferentes relaciones de refinamiento y simulación, ya estamos en condiciones de realizar algunas tareas de análisis sobre algunas propiedades elementales.

Por ejemplo, podemos preguntarnos si existe algún tipo de relación particular entre un par de LTS particulares. Es decir, dados $P_1$ y $Q_1$, preguntarnos, por ejemplo, ¿son $P_1$ y $Q_1$ bisimilares? Esta pregunta se reduce por supuesto a la existencia de una relación entre los estados de $P_1$ y $Q_1$, que los haga bisimilares.

Para analizar esto necesitamos representar a $P_1$ y $Q_1$ en Alloy, complementando nuestra especificación de LTS, MTS y relaciones de simulación/refinamiento. Esto incluye:

1. Definición de los universos particulares de estados y acciones (los usados por $P_1$ y $Q_1$).

2. Caracterización de las estructuras de $P_1$ y $Q_1$ (mediante signaturas).

3. Generación de un predicado asociado a la propiedad a analizar.

Por ejemplo, sean $P_1$ y $Q_1$ los vistos en 2.3, y supongamos que la propiedad a corroborar es bisimulación fuerte. Por lo tanto en Alloy esto nos queda de la siguiente manera:

```
module BisS
open library

one sig S0, S1, S2, S3 extends State {}
one sig A, B extends Action {}

one sig P1 extends LTS {} {
    init = S0
    trans = A -> S0 -> S1 + B -> S1 -> S0 + A -> S0 -> S2
            + B -> S2 -> S0
}

one sig Q1 extends LTS {} {
    init = S0
    trans = A -> S0 -> S2 + B -> S2 -> S3 + A -> S3 -> S2
            + A -> S3 -> S1 + B -> S1 -> S0
}

pred BisS (r: State -> State) {
    P1.init -> Q1.init in r
        and StrongBisimulation [r, P1.trans, Q1.trans]
}

run BisS for 5
```

Este predicado será analizado mediante una corrida, lo cual lo hacemos mediante el comando *run*. Obtenemos en este caso una instancia de r que hace al predicado válido. Suponiendo que no exista una r que haga válido el predicado, entonces obtendremos como respuesta que no existe una instancia tal que satisfaga dicho predicado.

Nótese que, en este caso, la respuesta del analizador es *absoluta*, en el sentido que si $P_1$ y $Q_1$ son bisimilares, el analizador encontrará una bisimulación (y viceversa). En

otras palabras, la cota 5 es suficiente para realizar un análisis completo: si no existen modelos de la propiedad dentro de esa cota no existen de ningún tamaño posible. Por supuesto esto es una excepción y no corresponde al caso general en el uso de Alloy y su analizador.

Otro ejemplo de propiedades que podemos analizar mediante el analizador de Alloy es la siguiente. Supongamos que tenemos una relación entre LTS o MTS, y queremos corroborar si existen instancias de MTS o LTS que no satisfagan las condiciones de esta relación. Siguiendo con el mismo ejemplo de antes, queremos saber si existe un $P_1$ y un $Q_1$, tal que no estén relacionados por la relación de interés definida. La forma de modelar este tipo de problemas es mediante aserciones; en otras palabras, queremos una instancia de $P_1$ y $Q_1$ tal que no satisfagan los requisitos para estar relacionados. Esto lo corroboramos sobre una aserción, utilizando el comando *assert*. Podemos ver este tipo de aserción, para el caso particular en el que la relación de interés es la bisimulación fuerte.

```
module BisS
open library

one sig S0, S1, S2, S3 extends State {}
one sig A, B extends Action {}

one sig P1 extends LTS {}
one sig Q1 extends LTS {}

assert BisS {
    all r: State -> State | P1.init -> Q1.init in r
        implies !StrongBisimulation [r, P1.trans, Q1.trans]
}

check BisS for 5
```

Si obtenemos una instancia de $P_1$, $Q_1$ y $r$, entonces estamos delante de un ejemplo en el cual $P_1$ no es un refinamieno de $Q_1$. En caso que no obtengamos una instancia, es porque para todas las instancias de LTS, es válida dicha aserción, dentro de las cotas provistas.

Es importante notar que otras herramientas como el MTSA [DFCU08] y MTSChecker [Fis06] no pueden analizar este tipo de propiedades; en estas herramientas sólo es posible analizar la existencia de relaciones de simulación/refinamiento para MTS/LTS dados.

## 3.5 Algunas mejoras al modelado

En el capítulo anterior hemos visto cómo a partir de las definiciones de simulaciones definimos las bisimulaciones y refinamientos, y de la misma manera, cómo a partir de las nociones de refinamientos introducimos las definiciones de implementaciones. Como hemos modelado hasta el momento las propiedades, es imposible definirlas en función de las otras, ya que en algunos casos trabajamos con LTS y en otros con MTS. La idea en esta optimización de modelado es trabajar los predicados a nivel de transiciones y olvidarnos que éstos están sumergidos en el contexto de un LTS o MTS. De esta manera logramos trabajar directamente con las transiciones siendo más concreto y más claro en las definiciones y a partir de esto, los predicados más complejos definirlos en función de los más simples. Asi reutilizamos definiciones y las mismas

quedan estructuralmente definidas, esto es, las bisimulaciones y refinamientos en función de las simulaciones, y las implementaciones en función de los refinamientos.

### 3.5.1 Simulaciones y Bisimulaciones

- Simulación Fuerte

```
/* States that r is a strong simulation between transitions      */
/*                                                     t1 and t2 */
pred StrongSimulation
        (r: State -> State, t1, t2: (Action -> State) -> State) {
            all a: Action | (~(r)).(t1[a]) in t2[a].(~(r))
}
```

- Bisimulación Fuerte

```
/* States that r is a strong bisimulation between transitions    */
/*                                                     t1 and t2 */
pred StrongBisimulation
        (r: State -> State, t1, t2: (Action -> State) -> State) {
            StrongSimulation [r,  t1, t2] and
            StrongSimulation [~r, t2, t1]

}
```

- Simulación Ramificada

```
/* States that r is a branching simulation between transitions    */
/*                                                  t1 and t2     */
pred BranchingSimulation
        (r: State -> State, t1, t2: (Action -> State) -> State) {
          all s1, s2, s3: State | all a: Action |
            (arrow[t1, s1, s2, a] and s1 -> s3 in r)
            implies
            (some s4, s5: State |
              s3 -> s4 in (preImageClTau[t2, s1.r]) and
              arrowT[t2, s4, s5, a] and s1 -> s4 + s2 -> s5 in r
            )
}
```

- Bisimulación Ramificada

```
/* States that r is a branching bisimulation between transitions  */
/*                                                  t1 and t2     */
pred BranchingBisimulation
      (r: State -> State, t1, t2: (Action -> State) -> State) {
          BranchingSimulation [r, t1, t2] and
          BranchingSimulation [~r, t2, t1]
}
```

- Simulación Débil

```
/* States that r is a weak simulation between transitions         */
/*                                                     t1 and t2  */
pred WeakSimulation
        (r: State -> State, t1, t2: (Action -> State) -> State) {
      all a: Action | (~r).(t1[a]) in
       ((*(t2[Tau])).((augTauTransition[t2])[a]).(*(t2[Tau]))).(~r)
}
```

- Bisimulación Débil

```
/* States that r is a weak bisimulation between transitions    */
/*                                                   t1 and t2 */
pred WeakBisimulation
        (r: State -> State , t1, t2: (Action -> State) -> State) {
            WeakSimulation [r, t1, t2] and
            WeakSimulation [~r, t2, t1]
}
```

### 3.5.2 Refinamientos

- Refinamiento Fuerte

```
/* States that r is a strong refinement between transitions t1  */
/* (required and maybe) and t2 (required and maybe).            */
pred StrongRefinement
(r: State -> State, t1p, t1r, t2p, t2r: (Action -> State) -> State) {
    StrongSimulation[r, t1r, t2r] and
    StrongSimulation[~r, t2p, t1p]
}
```

- Refinamiento Débil

```
/* States that r is a weak refinement between transitions t1    */
/* (required and maybe) and t2 (required and maybe).            */
pred WeakRefinement
(r: State -> State, t1p, t1r, t2p, t2r: (Action -> State) -> State) {
    WeakSimulation[r, t1r, t2r] and
    WeakSimulation[~r, t2p, t1p]
}
```

### 3.5.3 Implementaciones

- Implementación Fuerte

```
/* States that r is a strong implementation between transitions t1 */
/* (required and maybe) and t2.                                    */
pred StrongImplementation
     (r: State -> State, t1p, t1r, t2: (Action -> State) -> State) {
         StrongRefinement [r, t1p, t1r, t2, t2]
}
```

- Implementación Débil

```
/* States that r is a weak implementation between transitions t1  */
/* (required and maybe) and t2.                                   */
pred WeakImplementation
      (r: State -> State, t1p, t1r, t2: (Action -> State) -> State) {
          WeakRefinement [r, t1p, t1r, t2, t2]
}
```

- Implementación Ramificada

```
/* States that r is a branching implementation between transitions */
/* t1 (required and maybe) and t2.                                 */
pred BranchingImplementation
      (r: State -> State, t1p, t1r, t2: (Action -> State) -> State) {
            (all s1, s2: State | all a: Action | all s3 : State |
                arrow[t1r, s1, s2, a]  and s1 -> s3 in r
```

```
          implies
          (some s4, s5: State |
              s3 -> s4 in (preImageClTau[t2, s1.r]) and
              arrowT[t2, s4, s5, a] and s2 -> s5 in r
          ))
      (all s1, s2: State | all a: Action | all s3 : State |
          arrow[t2, s1, s2, a]  and s1 -> s3 in ~r
          implies
          (some s4, s5: State |
              s3 -> s4 in (preImageClTau[t1p, s1.(~r)]) and
              arrowT[t1p, s4, s5, a] and s2 -> s5 in ~r
          ))
}
```

## 3.6 Comparación de diversas herramientas en el análisis de bisimulaciones y refinamientos

En esta sección realizamos una primera comparación entre el uso de SAT solving para analizar la existencia de refinamientos entre LTS/MTS y otras herramientas, basadas con algoritmos de chequeo de (bi)simulación. Las herramientas a comparar con nuestro enfoque son MTSChecker [Fis06, FUB06], desarrollada por D. Fischbein, y MTSA [DFFU07, FGPA05], una extensión de LTSA que maneja sistemas de transiciones modales desarrollada por D. Fischbein, Nicolás D'Ippolito y Sebastián Uchitel. Estas herramientas analizan la existencia de relaciones de bisimulación o refinamiento (varias variantes de estas relaciones) mediante algoritmos específicos para estas tareas; nuestro enfoque, en cambio, realiza una técnica basada en constraint solving, que en teoría analiza todas las posibilidades (fuerza bruta). Es de esperar entonces que tanto MTSA como MTSChecker funcionen más eficientemente.

Esto puede hacer pensar al lector que la comparación que realizamos es inútil (e incluso que utilizar constraint solving para el análisis de existencia de relaciones de bisimulación/refinamiento, contando con algoritmos para el análisis de estas relaciones, es inútil). Sin embargo, como explicamos anteriormente, existen buenas razones para hacer esto. Por un lado, realizar la comparación nos permitirá analizar la factibilidad del enfoque (i.e. aún cuando los tiempos sean superiores a los obtenidos utilizando algoritmos para chequeo de bisimulación/refinamiento, son los tiempos obtenidos usando SAT solving razonables?). Por otra parte, el uso de Alloy para esta tarea nos brinda una herramienta más versátil, que podemos consultar de manera que los algoritmos específicos para bisimulación y refinamiento no soportan.

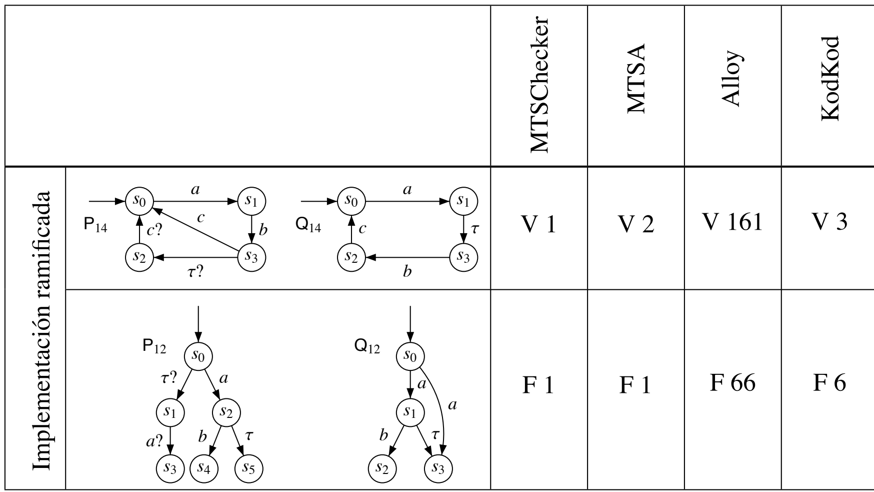
La comparación fue realizada para varios pares de LTS y MTS, analizando bisimulación y refinamiento de distinta naturaleza. Se utilizó para el análisis una Intel Core 2 Duo CPU T5250 1.50GHz, con 2Gb de RAM. Los resultados obtenidos se resumen en la siguiente tabla. Las últimas cuatro columnas indican las respuestas obtenidas por las diferentes herramientas (V en caso de ser bisimilares o existe la relación de refinamiento o implementación, F en otro caso) y el tiempo demandado por correspondiente a cada una de éstas, en milisegundos.

La tabla incluye, como el lector puede apreciar, una columna con tiempos de una herramienta que no mencionamos hasta el momento: KodKod [TJ07]. KodKod es el motor de análisis detrás de Alloy (a partir de su versión 4); toda especificación Alloy es parseada y traducida a KodKod, para luego ser analizada mediante SAT solving. En los tiempos de KodKod (versus los de Alloy) nos ahorramos entonces el tiempos de parsing y generación del modelo KodKod resultante. Es importante notar que el

análisis de especificaciones KodKod sigue siendo mediante constraint solving, es decir, esencialmente mediante el análisis de todas las posibles instancias acotadas por cierto valor provisto por el usuario (en otras palabras, el análisis sigue siendo por fuerza bruta).

| | | MTSChecker | MTSA | Alloy | KodKod |
|---|---|---|---|---|---|
| Bisimulación Fuerte | $P_1$, $Q_1$ | V 12 | V 8 | V 60 | V 6 |
| | $P_2$, $Q_2$ | F 3 | F 5 | F 64 | F 4 |
| Bisimulación Débil | $P_3$, $Q_3$ | V 4 | V 8 | V 144 | V 4 |
| | $P_4$, $Q_4$ | F 4 | F 4 | F 60 | F 8 |
| Bis. ramificada | $P_5$, $Q_5$ | V 2 | V 5 | V 140 | V 4 |
| | $P_6$, $Q_6$ | F 2 | F 4 | F 80 | F 4 |

| | | MTSChecker | MTSA | Alloy | KodKod |
|---|---|---|---|---|---|
| Refinamiento fuerte | $P_7$, $Q_7$ | V 1 | V 2 | V 92 | V 3 |
| | $Q_7$, $P_7$ | F 1 | F 2 | F 48 | F 4 |
| Refinamiento débil | $P_8$, $Q_8$ | V 2 | V 4 | V 184 | V 2 |
| | $P_9$, $Q_9$ | F 1 | F 3 | F 64 | F 1 |
| Implem. fuerte | $P_{10}$, $Q_{10}$ | V 1 | V 2 | V 56 | V 1 |
| | $P_{11}$, $Q_{11}$ | F 1 | F 1 | F 52 | F 2 |
| Implementación débil | $P_{12}$, $Q_{12}$ | V 1 | V 2 | V 100 | V 3 |
| | $P_{13}$, $Q_{13}$ | F 1 | F 2 | F 76 | F 3 |



| | | MTSChecker | MTSA | Alloy | KodKod |
|---|---|---|---|---|---|
| Implementación ramificada | $P_{14}$ / $Q_{14}$ | V 1 | V 2 | V 161 | V 3 |
| | $P_{12}$ / $Q_{12}$ | F 1 | F 1 | F 66 | F 6 |

Como podemos observar en la tabla, los tiempos utilizando Alloy son en algunos casos bastante peores que los de MTSChecker y MTSA. Sin embargo, en todos los casos obtenemos una respuesta en unos pocos segundos mostrando que el análisis usando Alloy es factible. Más aún, cambiando el uso de Alloy por el uso directo de KodKod, los tiempos de análisis bajan significativamente, acercándose a los tiempos de MTSChecker y MTSA.

Nótese además que para este tipo de chequeo, tanto Alloy como KodKod nos dan respuestas certeras, en el sentido que, dada que las relaciones posibles entre LTS y MTS están acotadas, la exploración de relaciones posibles es exhaustiva. Por lo tanto, si existe una relación de refinamiento/bisimulación, está garantizado que Alloy/KodKod la encontrará.

Por último nuestra modelización tiene una utilidad adicional de nuestro modelo de MTS/LTS en Alloy: depuración de algoritmos para el análisis de relaciones de refinamiento, bisimulación o implementación. En efecto, la declaratividad brindada por el lenguaje Alloy hace que las especificaciones de este tipo de relaciones sea directa (de hecho, la especificación no es otra cosa que la expresión de las definiciones de estas relaciones en la sintaxis de Alloy), permitiéndonos contrastar los resultados obtenidos de los algoritmos para chequeo de bisimulación, refinamiento, etc, con las definiciones formales de estas relaciones.

## 3.7 Ejemplos de otros análisis con Alloy Analyzer

Ya hemos visto cómo podemos utilizar Alloy Analyzer para comprobar si, dados dos MTS, existe una relación de refinamiento de cierto tipo entre ellos.

Veremos ahora algunos ejemplos de la versatilidad de Alloy, con otro tipo de consultas que podemos hacer a nuestra caracterización de MTS y refinamientos. Estas consultas, que son directas en Alloy, no pueden realizarse en las herramientas MTSChecker y MTSA.

Mostraremos también ejemplos de consultas expresadas con facilidad, pero que, debido a características del mecanismo de análisis subyacente a Alloy Analyzer, no pueden ser analizadas.

- Todo refinamiento fuerte, es también un refinamiento débil. Más precisamente para

toda relación r, si ésta satisface las condiciones para ser un refinamiento fuerte entre dos MTS cualquiera, entonces la misma satisface las condiciones para ser un refinamiento débil para estos mismos MTS. Caracterizamos esta propiedad en Alloy de la siguiente manera:

```
module SRefImpliesWRef
open library

one sig A, B extends Action {}
one sig S0, S1, S2, S3 extends State {}

one sig m1 extends MTS {}
one sig m2 extends MTS {}

assert SRefImpliesWRef {
    all r: State -> State |
        StrongRefinement[r, m1.trans, m1.req, m2.trans, m2.req]
        implies
        WeakRefinement[r, m1.trans, m1.req, m2.trans, m2.req]
}

check SRefImpliesWRef for 4
```

Esta propiedad es válida, con lo cual Alloy Analyzer no encontrará contraejemplos de la misma, de tamaño acotado por 4 (porque no existen contraejemplos de ningún tamaño).

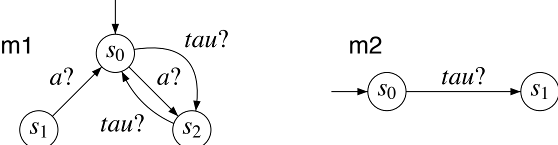
Es importante notar que esta propiedad está cuantificada universalmente sobre los LTS m1 y m2, que son variables libres en la especificación. Por otro lado, estamos acotando el número de estados y acciones de nuestro universo. Evidentemente, esta propiedad no puede analizarse en MTSA y MTSChecker. Alloy Analyzer realiza este análisis con relativa eficiencia, como se muestra en la tabla a continuación.

| Cotas | Tiempos |
|---|---|
| 4 | 2.14s |
| 5 | 3.59s |
| 10 | 3m14s |

- Todo refinamiento débil es también un refinamiento fuerte. Esta propiedad, la recíproca de la anterior, se expresa de la siguiente manera:

```
module WRefImpliesSRef
open library

one sig A, B extends Action {}
one sig S0, S1, S2 extends State {}

one sig m1 extends MTS {}
one sig m2 extends MTS {}

assert WRefImpliesSRef {
    all r: State -> State |
        WeakRefinement[r, m1.trans, m1.req, m2.trans, m2.req]
        implies
        StrongRefinement[r, m1.trans, m1.req, m2.trans, m2.req]
}

check WRefImpliesSRef for 4
```

A diferencia de la propiedad anterior, ésta no es válida, Alloy Analyzer rápidamente encuentra un contraejemplo de la misma, con 4 como cota para los tamaños de los dominios. El contraejemplo encontrado por Alloy consta de los siguientes MTS m1 y m2:




Claramente una de las relaciones que hace válido el refinamiento débil entre m1 y m2, pero que no hace válido el refinamiento fuerte es la siguiente:

$$r = \{(s_0, s_0), (s_1, s_2), (s_2, s_1)\}$$

Nuevamente, y sólo para poder apreciar cuán eficientemente se puede realizar este análisis, proveemos la siguiente tabla en tiempos de análisis de la propiedad:

| Cotas | Tiempos |
|---|---|
| 4 | 3,07s |
| 5 | 7,84s |
| 10 | 1m16s |
| 20 | 2m23s |

- La relación de refinamiento fuerte es transitiva sobre LTS. En otras palabras, para toda terna de LTS l1, l2 y l3, si existen refinamientos fuertes entre l1 y l2, y entre l2 y l3, entonces existe un refinamiento fuerte entre l1 y l3. Esto lo podemos expresar de la siguiente manera:

```
module StrongTrans
open library

one sig A, B extends Action {}
one sig S0, S1, S2 extends State {}

one sig l1 extends LTS {}
one sig l2 extends LTS {}
one sig l3 extends LTS {}

assert StrongTrans {
    all r12, r23: State -> State |
        (l1.init -> l2.init in r12 and
         StrongBisimulation[r12, l1.trans, l2.trans] and
         l2.init -> l3.init in r23 and
         StrongBisimulation[r23, l2.trans, l3.trans])
       implies
         (some r: State -> State |
         l1.init -> l3.init in r and
         StrongBisimulation[r, l1.trans, l3.trans])
}

check StrongTrans for 10
```

Alloy Analyzer no puede analizar esta propiedad debido a que no puede skolemizar relaciones de alto orden. Por el momento asumiremos que esta propiedad no es analizable por Alloy Analyzer, y veremos más adelante cómo atacar este tipo de problemas.

- Otra propiedad interesante para analizar es el *refinamiento observacional común*, introducido en [BCU06]. Este concepto es crucial para la descripción de la *mezcla* (o *merge*) de MTS [BCU06]. Podemos describir al refinamiento observacional común de la siguiente manera. En el contexto de lenguajes de especificaciones que admiten parcialidad, es usual encontrarse con el caso en el cual uno cuenta con dos o más descripciones parciales de diferentes partes de un mismo sistema. El refinamento observacional común nos permite encontrar una única descripción parcial que abarque a estas otras descripciones.

  Para poder definir refinamiento observacional común, necesitamos introducir la noción de interfaz, que identifica las acciones observables asociadas a un MTS (y oculta el resto).

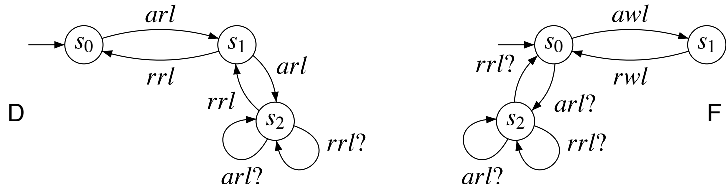
  **Definición 3.7.1.** Una *interfaz* de un MTS $\mathsf{M} = (S, L, \longrightarrow_r, \longrightarrow_p, s_0)$ respecto de un conjunto de acciones X $\subseteq$ Actions, es un nuevo MTS $\mathsf{M}' = (S, X, \longrightarrow_{r'}, \longrightarrow_{p'}, s'_0)$, donde las acciones que no pertenecen a X están ocultas, es decir las acciones que no pertenecen a X son renombradas por Tau en las transiciones. Denotaremos $\mathsf{M}' = \mathsf{M}@L$

  **Definición 3.7.2.** Un *Refinamiento observacional común (COR - Common Observational Refinement)* entre dos MTS $\mathsf{M} = (S, L, \longrightarrow_r, \longrightarrow_p, s_0)$ y $\mathsf{N} = (S', L', \longrightarrow_{r'}, \longrightarrow_{p'}, s'_0)$, es un MTS $\mathsf{P} = (T, O, \longrightarrow_{r''}, \longrightarrow_{p''}, t_0)$, tal que se cumple que $0 \subseteq (L \cup L')$, $\mathsf{M} \preccurlyeq \mathsf{P}@L$ y $\mathsf{N} \preccurlyeq \mathsf{P}@L'$.

  Veamos cómo describimos las funciones `interface`, `COR` y una función auxiliar `Acts` (que devuelve el conjunto de acciones de un MTS) en Alloy:

```
fun interface
  (t: (Action -> State) -> State, X: set Action):
                                    (Action -> State) -> set State{
    t - (((Action-Tau)-X) -> State -> State) +
    (Tau -> ((Action-Tau)-X).t)
}

fun acts (t: (Action -> State) -> State): Action{
  (t.State).State
}

pred COR
 (r1: State -> State, r2: State -> State, m1: MTS, m2: MTS, m: MTS) {
  m1.init -> m.init in r1 and
  m2.init -> m.init in r2 and
  WeakRefinement[r1, m1.trans, m1.req,
    interface[m.trans, acts[m1.req]], interface[m.req, acts[m1.req]]]
  and
  WeakRefinement[r2, m2.trans, m2.req,
    interface[m.trans, acts[m2.req]], interface[m.req, acts[m2.req]]]
}
```

  Utilizaremos ahora un ejemplo de [BCU06], donde se describen dos instancias de un sistema de lectores/escritores. La finalidad será que, mediante Alloy Analyzer, encontremos un refinamiento observacional común de este sistema. Detallamos a continuación los grafos con las dos descripciones parciales:

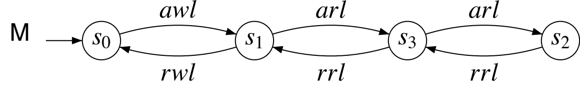



donde `arl` denota `acquireReadLock`, `rrl` denota `releaseReadLock`, `awl` denota `acquireWriteLock` y `rwl` denota `releaseWriteLock`, correspondiéndose así con la nomenclatura utilizada en [BCU06].

Sólo nos resta encontrar un refinamiento común entre estos dos MTS. En Alloy esto lo podemos lograr mediante el uso de predicados, como se muestra a continuación.

```
module COR
open library

one sig ARL, RRL, RWL, AWL extends Action {}
one sig S0, S1, S2 extends State {}

one sig D extends MTS {} {
  init = S0
  req = ARL -> S0 -> S1 + ARL -> S1 -> S2 +
        RRL -> S2 -> S1 + RRL -> S1 -> S0
  trans = req + ARL -> S2 -> S2 + RRL -> S2 -> S2
}

one sig F extends MTS {} {
  init = S0
  req = AWL -> S0 -> S1 + RWL -> S1 -> S0
  trans = req + RRL -> S2 -> S0 + ARL -> S0 -> S2 +
          RRL -> S2 -> S2 +  ARL -> S2 -> S2
}

one sig M extends MTS{}

pred RefCommon (r1: State -> State, r2: State -> State) {
    COR[r1, r2, D, F, M]
}

run RefCommon for 20
```

El resultado obtenido de análisis usando Alloy Analyzer es el siguiente MTS M:




Notemos que el ejemplo anterior constituye un caso de estudio más concreto, y más representativo del tipo de especificaciones mediante MTS que normalmente se encontrarían en la práctica. Mostramos la siguiente tabla con los tiempos de análisis de esta propiedad, para dar al lector información sobre el tiempo requerido de análisis.

| Cotas | Tiempos |
|---|---|
| 4 | 102ms |
| 5 | 114ms |
| 10 | 133ms |
| 20 | 154ms |

# 4 Predicados de Alto Orden sobre MTS y LTS

En el capítulo anterior mostramos algunos tipos de chequeos que podemos realizar utilizando la caracterización de LTS/MTS, (bi)simulaciones y refinamientos. Vimos algunas propiedades que con Alloy no podemos chequear, y también algunos ejemplos que podemos chequear con Alloy pero no con otras herramientas, como MTSA y MTSChecker, y destacamos la versatibilidad de Alloy para estas tareas.

Esta versatibilidad de la cual hablamos sugiere utilizar nuestra caracterización de LTS y MTS para algunas otras tareas. Por ejemplo, durante la investigación de formas adecuadas de caracterizar refinamientos entre MTS se han propuesto varias definiciones, y se han conjeturado propiedades de éstas. Un ejemplo de esto es la recíproca de la preservación de implementaciones por parte de refinamiento fuerte, es decir, la siguiente propiedad:

$$I[\mathsf{M}] \subseteq I[\mathsf{N}] \Rightarrow \mathsf{N} \preceq \mathsf{M}$$

donde $I[\mathsf{M}]$ e $I[\mathsf{N}]$ denotan los conjuntos de implementaciones de los MTS $\mathsf{M}$ y $\mathsf{N}$ respectivamente y $\mathsf{N} \preceq \mathsf{M}$ significa que $\mathsf{N}$ refina a $\mathsf{M}$ o que $\mathsf{M}$ es un refinamiento de $\mathsf{N}$.

Esta propiedad, durante un tiempo se creyó verdadera e incluso se demostró (erróneamente) un teorema que asegura que refinamiento es completo para implementación [Hut05]. Esta propiedad es en realidad falsa. Los contraejemplos que se conocen de la misma son relativamente pequeños [Fis], algunos de ellos son:

donde el conjunto de implementaciones de $\mathsf{M}_1$ y $\mathsf{N}_1$, y $\mathsf{M}_2$ y $\mathsf{N}_2$ son los mismos, pero sin embargo $\mathsf{N}_1 \not\preceq \mathsf{M}_1$ y $\mathsf{N}_2 \not\preceq \mathsf{M}_2$.

La pregunta que podemos hacernos ahora es: ¿podremos utilizar Alloy para "testear" este tipo de propiedades? La utilidad de hacer esto es evidente, ya que nos permitiría analizar la validez, en modelos acotados, de propiedades de este tipo, y en caso de encontrar contraejemplos comprender las causas por las cuales falla.

Desafortunadamente, como veremos más adelante, este tipo de propiedad no se puede analizar directamente con Alloy. La razón de esto es que el antecedente de la propiedad tiene, implicitamente una cuantificación sobre todas las implementaciones del MTS $\mathsf{M}$. Este conjunto de implementaciones obviamente depende de $\mathsf{M}$, e incluso para MTS pequeños puede ser infinito.

Consideremos el caso del siguiente MTS:

$a?$

$\longrightarrow (s_0) \circlearrowleft$

Éste tiene infinitas implementaciones. En particular, la siguiente familia de LTS son todas implementaciones del MTS anterior.

$\mathsf{I}_0 \quad \longrightarrow (s_0)$

$\mathsf{I}_1 \quad \longrightarrow (s_0) \xrightarrow{a} (s_1)$

$\mathsf{I}_2 \quad \longrightarrow (s_0) \xrightarrow{a} (s_1) \xrightarrow{a} (s_2)$

$\vdots \quad \vdots \quad \vdots \quad \vdots \quad \vdots$

$$\mathsf{I}_n = (\{s_i\}^n, \{a\}, \{s_i \rightarrow s_{i+1} \mid 0 \leq i \leq n\}, s_0)$$

Observemos que cada una de las implementaciones es distinta de la otra módulo bisimulación.

La exploración de contraejemplos de las propiedad busca pares de MTS $\mathsf{M}$ y $\mathsf{N}$ tales que $I[\mathsf{M}] \subseteq I[\mathsf{N}]$ y $\mathsf{N} \not\preceq \mathsf{M}$ (i.e., que hacen falsa la implicación). Pero para poder buscar estos contraejemplos necesitamos poder cuantificar MTS $\mathsf{M}$ y $\mathsf{N}$ tales que $I[\mathsf{M}] \subseteq I[\mathsf{N}]$, lo cual requiere, como vimos, chequear un número infinito de implementaciones, en algunos casos. Esta limitación sólo nos permitirá utilizar Alloy Analyzer para buscar *potenciales* contraejemplos, como veremos más adelante. Por cuestiones de presentación nos convendrá trabajar con la contrarecíproca de la propiedad anterior:

$$\mathsf{N} \not\preceq \mathsf{M} \Rightarrow I[\mathsf{M}] \not\subseteq I[\mathsf{N}]$$

## 4.1 Preservación de implementaciones en Alloy

La propiedad mencionada anteriormente puede caracterizarse en Alloy con relativa facilidad, mediante las siguientes aserciones: primero debemos conseguir dos candidatos de MTS tal que no se cumpla que uno es refinamiento del otro, para luego corroborar la relación de sus conjuntos de implementaciones.

```
module Prop
open library

one sig A, B extends Action{}
one sig S0, S1, S2 extends State{}
```

```
one sig m1 extends MTS{}
one sig m2 extends MTS{}

pred NoRef {
    !(some r: State -> State | m1.init -> m2.init in r and
        StrongRefinement[r, m1.trans, m1.req, m2.trans, m2.req])
}

run NoRef for 5
```

Una vez que encontremos dos instancias de MTS que satisfagan esta propieada, estaremos en condiciones de analizar la relación entre sus respectivos conjuntos de implementaciones.

```
module Prop
open library

one sig A, B extends Action{}
one sig S0, S1, S2 extends State{}

/* Notar que en este punto debe instanciarse los MTS encontrados   */
/*                                      con el chequeo anterior */
one sig m1 extends MTS{}{
 trans = ...
 req = ...
}
one sig m2 extends MTS{}{
 trans = ...
 req = ...
}
one sig i extends LTS{}

pred Prop {
    some r2: State -> State | m2.init -> i.init in r2 and
        StrongImplementation[r2, m2.req, m2.trans, i.trans] and
    !(some r1: State -> State | m1.init -> i.init in r1 and
        StrongImplementation[r1, m1.req, m1.trans, i.trans])
}

run Prop for 5
```

Como vemos esta propiedad es claramente expresable en Alloy (como lo demuestra las aserciones anteriores), pero al igual que para algunas propiedades del capítulo anterior, Alloy Analyzer no puede analizarla, debido a que contiene cuantificadores de alto orden no skolemizables (las cuantificaciones sobre relaciones binarias, en este caso).

Una forma de sortear el problema de las cuantificaciones de alto orden, usualmente utilizada en Alloy, es la introducción de una signatura para representar el dominio de alto orden, en nuestro caso las relaciones binarias. Esto lo podemos hacer simplemente de la siguiente manera:

```
sig Relation {
    rel: State -> State
}
```

La propiedad puede entonces, ser expresada de la siguiente manera:

```
module Prop
open library

one sig A, B extends Action{}
one sig S0, S1, S2 extends State{}

fact{
    !(some r: Relation | m1.init -> m2.init in r.rel and
        StrongRefinement[r.rel, m1, m2])
}

one sig m1 extends MTS{}
one sig m2 extends MTS{}

pred NoRef {
    !(some r: Relation | m1.init -> m2.init in r.rel and
        StrongRefinement[r.rel, m1.trans, m1.req, m2.trans, m2.req])
}

run NoRef for 5
```

Una vez que encontremos dos instancias de MTS que satisfagan esta especificación, las representamos (codificamos) en un modelo Alloy y analizamos la siguiente propiedad:

```
module Prop
open library

one sig A, B extends Action{}
one sig S0, S1, S2 extends State{}

fact{
    !(some r: Relation | m1.init -> m2.init in r.rel and
        StrongRefinement[r.rel, m1, m2])
}

/* Notar que en este punto debe instanciarse los MTS encontrados     */
/*                                          con el chequeo anterior */
one sig m1 extends MTS{}{
 req = ...
 trans = ...
}
one sig m2 extends MTS{}{
 req = ...
 trans = ...
}
one sig i extends LTS{}

pred Prop {
    some r2: Relation | m2.init -> i.init in r2.rel and
        StrongImplementation[r2.rel, m2.req, m2.trans, i.trans] and
    !(some r1: Relation | m1.init -> i.init in r1.rel and
        StrongImplementation[r1.rel, m1.req, m1.trans, i.trans])
}

run Prop for 5
```

si el predicado no se satisface entonces, estamos delante de un potencial contraejemplo para nuestra propiedad. Es sólo un contraejemplo potencial porque no exploramos todo el universo de implementaciones, sino solo implementaciones acotadas por las cotas

provistas por el usuario para los dominios de la especificación. Si, por el contrario, la propiedad es satisfecha, el par de MTS provistos no nos sirve, por lo que debemos buscar otro par de MTS para continuar realizando el analisis.

El análisis anterior podría en principio expresarse en una única especificación Alloy, en lugar de realizarlo en dos etapas (una para buscar instancias que satisfagan el antecedente, e intentar que éstas falsifiquen el consecuente). Sin embargo, como veremos más adelante, el análisis de esta propiedad en una sola etapa no produce los resultados que uno esperaría, debido a que la semántica de Alloy no fuerza al analizador la generación de todas las posibles relaciones entre estados. Discutiremos este problema en detalle a continuación.

### 4.1.1 Análisis de propiedades con cuantificación de Alto Orden

Luego de la modificación anterior, podemos usar Alloy Analyzer para chequear nuestra propiedad de interés. Sin embargo, estamos en presencia de una especificación en la cual la signatura (en nuestro caso, Relation) debe representar todos los valores posibles de una estructura compuesta (en nuestro caso, todas las relaciones binarias posibles de estados). Como se explica en la sección 5.3 de [Jac06], esto contradice la semántica de Alloy, según la cual una signatura representa sólo un conjunto de valores. Como se explica en [Jac06], este inconveniente puede ser resuelto forzando, mediante hechos, que la signatura que representa todas las instancias posibles de valores efectivamente esté "poblada" adecuadamente. Este tipo de axioma, denominado **axioma de generación**, en nuestro caso tiene la siguiente forma:

```
fact genRelation {
    some r: Relation | no r.rel
    all r: Relation, s0, s1: State |
        some r':Relation | r'.rel = r.rel + (s0 -> s1)
}
```

### 4.1.2 Limitaciones en el Análisis mediante Alloy Analyzer

Si bien luego de la transformación, y mediante la inclusión de axiomas de generación, podemos analizar la propiedad descripta, existen limitaciones en la cantidad máxima de elementos que se pueden asociar con cada signatura.

Debido al uso que le damos a la signatura *Relation*, necesitamos, tener tantos elementos asociados a esta signatura como relaciones binarias tengamos entre estados. Este número depende claramente del número de estados. Suponiendo que tenemos $i$ estados en nuestros MTS/LTS, y considerando a $n$ el número de posibles asociaciones entre estados tenemos $n = i \times i$; la cantidad de relaciones binarias para $i$ estados está dada por la siguiente expresión:

$$\sum_{i=0}^{n} \binom{n}{i} = 2^n$$

Tomando casos concretos, como por ejemplo 2, 3 y 4 estados, la cantidad de relaciones que necesitamos instanciar son 16, 512 y 65536, respectivamente.

Alloy acepta 255 como cota máxima para las signaturas. Esto nos permite realizar el chequeo anterior para a lo sumo 2 estados. Otra alternativa es codificar los elementos de la signatura *Relation* como signaturas "one" dentro del modelo. Alloy Analyzer

superar un máximo de 480 signaturas, con lo cual seguimos sin poder superar las relaciones sobre 2 estados. Recientemente, con la última versión de Alloy Analyzer (4.1.9, de 27/10/2008), la cantidad máxima de signaturas “one” que se pueden definir se incrementó a 1000, con lo cual podemos chequear relaciones de hasta 3 estados.

Además, no sólo es un problema el número de relaciones, sino también lo costoso de su generación, en términos computacionales. La siguiente tabla resume el costo de análisis de la propiedad, y muestra una limitación importante en la escalabilidad del análisis.

| Estados | Tiempos |
|---|---|
| 1 | 3ms |
| 2 | 44ms |
| 3 | 1s45ms |

Si bien la diferencia entre 2 y 3 estados puede parecer irrelevante, debe tenerse en cuenta que estos tiempos corresponden sólo al chequeo de si un par de MTS cumple con que uno es un refinamiento de otro. Este chequeo se realizará para un gran número de MTS (todas las combinaciones de a pares de MTS que podamos generar). Veremos mas adelante como estos tiempos influyen notoriamente en los tiempos de análisis para la propiedad completa.

## 4.2 Mínimo conjunto de Relaciones

Como hemos visto, las limitaciones de Alloy Analyzer nos impiden chequear la propiedad en la cual estamos interesados. El problema está dado por el número de relaciones posibles entre estados, y la relación entre este número y la cantidad de elementos de la signatura *Relation*.

En nuestro caso, las relaciones entre estados tienen una finalidad específica: las queremos para estudiar potenciales relaciones de refinamiento entre MTS. Podemos preguntarnos entonces si todas las relaciones binarias entre estados son realmente necesarias. Si un número significativo de relaciones son innecesarias, podemos eliminar los mismos y así incrementar las cotas para el análisis de la propiedad.

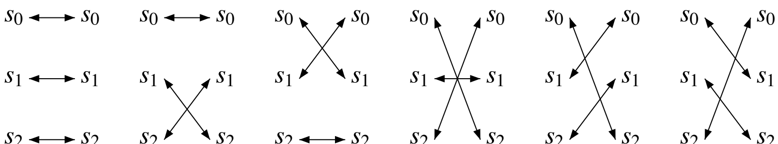
La primera simplicación que realizaremos consiste en jar el estado inicial de los MTS/LTS. El estado inicial para éstos será $s_0$. Esto fuerza que las relaciones, candidatos a ser renamientos, deben contener el par $(s_0, s_0)$. Esta simplificación, que parece trivial, reduce el número de relaciones posibles significativamente, como resumiremos en una tabla mas adelante en esta sección.

La segunda simplicación es mucho mas sofisticada; la misma consiste en obtener un conjunto (mínimo) de relaciones “canonicas” en algún sentido, tal que a partir de estas podamos obtener las restantes mediante biyecciones entre los estados. Describiremos esta simplificación a continuación.

### 4.2.1 En busca de relaciones representantes

Como describimos anteriormente, la idea de la segunda de nuestras optimizaciones es encontrar el mínimo conjunto de representantes tal que por medio de biyecciones poder reconstruir todo el universo de las relaciones. Para ello, en primer instancia debemos analizar las posibles biyecciones sobre estos estados.

Consideremos, por ejemplo, las biyecciones sobre 3 estados. Existen 6 posibles biyecciones, que enumeramos a continuación:




De estas, sólo las dos primeras mapean $s_0$ en $s_0$; por lo tanto, éstas son las únicas biyecciones que nos sirven, ya que preservan el mapeo de $s_0$ en $s_0$ (como requerimos para ser un refinamiento).

El conjunto mínimo de relaciones tales que mediante estas biyecciones nos permiten reconstruir el conjunto de relaciones original no tiene por que generarse a mano. En lugar de esto, utilizaremos Alloy Analyzer, y aprovecharemos su versatilidad, para *encontrar* este conjunto mínimo.

Una vez obtenido el conjunto mínimo de relaciones, lo utilizamos para reconstruir el conjunto de relaciones original mediante un simple script, cuya salida son especificaciones Alloy.

### 4.2.2 Minimizando relaciones con Alloy Analyzer

Continuemos con el caso en el cual queremos generar todas las relaciones para tres estados. En principio ya tenemos el conjunto mínimo de relaciones y sabemos cuales son las biyecciones a partir de las cuales podemos generar todas las relaciones originales. Veamos ahora con qué debemos contar para poder analizar nuestra propiedad mediante Alloy Analyzer, ya que debemos quedarnos con un representante de cada una de las relaciones. Para hacerlo debemos modelar lo siguiente:

- las instancias de los estados sobre los cuales vamos a trabajar y definir todo el conjunto de estados

```
abstract sig State {}
one sig S0, S1, S2 extends State {}

one sig SetState {
    cjto: set State
}{
    cjto = S0 + S1 + S2
}
```

- el universo de relaciones, y el dominio e imagen de su relación binaria

```
abstract sig Relation {
    rel: State -> State
}{
    rel in SetState.cjto -> SetState.cjto
}
```

- el universo con todas las relaciones entre estados en lenguaje Alloy, es decir, poblar el universo.

  La idea es obviar el axioma de generación, por lo que le pasaremos el universo de todas las relaciones que al menos contienen el par $(s_0,s_0)$. Estas instancias las generamos mediante un script simple que podemos observar en Apéndice E y en el Apéndice C el lector puede observar todas las instancias generadas.

- las biyecciones que analizamos anteriormente

```
abstract sig Biyeccion {
    rel: State -> State
}{
    rel in SetState.cjto -> SetState.cjto and
    rel[State] = SetState.cjto and
    rel.State = SetState.cjto
}

one sig B0 extends Biyeccion {} {
    rel = S0 -> S0 + S1 -> S1 + S2 -> S2
}

one sig B1 extends Biyeccion {} {
    rel = S0 -> S0 + S1 -> S2 + S2 -> S1
}
```

- por último, el predicado de mínimo conjunto de relaciones, a partir del cual necesitamos poder generar todo el universo de relaciones que codificamos previamente como instancias concretas.

```
pred conjuntosDeRs (SetR: set Relation) {
    (all r,r' : Relation | some f,f': Biyeccion|
        (r in SetR and r' in SetR and
        r.rel = (f.rel).(r'.rel).(f'.rel) implies r = r')) and
    (all r: Relation | S0 -> S0 in r.rel implies (some r': Relation|
        r' in SetR and
        some f,f': Biyeccion | r.rel = (f.rel).(r'.rel).(f'.rel)))
}
```

Notemos que sólo podemos hacer esto para a lo sumo 3 debido a la capacidad limita de instancias que soporta Alloy Analyzer.

Ya estamos en condiciones de realizar el analisis. Alloy Analyzer nos dará como resultado el mínimo conjunto de relaciones que nos permita generar el resto (en algún sentido, relaciones representantes). Trabajaremos con este conjunto en lugar de trabajar con el conjunto de relaciones original.

Para poder apreciar el impacto de las simplificaciones propuestas, observemos la siguiente tabla comparativa.

| Cant. Estados | (1) | (2) | (3) |
|---|---|---|---|
| 1 | 2 | 1 | 1 |
| 2 | 16 | 8 | 8 |
| 3 | 512 | 256 | 84 |

En esta tabla (1) corresponde a la cantidad original de relaciones entre estados, (2) corresponde a la cantidad de relaciones eliminando las que no tienen el par $(s_0, s_0)$ y (3) corresponde a la cantidad mínima de relaciones que no contienen $(s_0, s_0)$ y que permiten generar las demas a través de biyecciones.

Podemos apreciar que, eliminando las relaciones que no contienen el par $(s_0, s_0)$, y reduciendo las relaciones a un conjunto de representantes mínimo, el número de instancias de relaciones a chequear se reduce considerablemente.

Hemos logrado reducir el problema, para el caso de relaciones sobre tres estados, con un universo de 84 relaciones, cuando partimos con un total de 512. Más adelante aclararemos por que podemos prescindir de las demás relaciones.

Cabe aclarar que Alloy Analyzer cuenta con un mecanismo incorporado en la herramienta para eliminación de relaciones isomorformas [JJD98] similar al mecanismo

que acabamos de describir, sin embargo en nuestro caso nos hemos visto forzados a incorporar un modelo Alloy de relaciones (caracterizado por la signatura Relation). Esto trae como consecuencia que a nuestras relaciones no se les aplique tal optimización. Por esta razón la desición de incorporar este mecanismo.

## 4.3 Generación de LTS/MTS

Resumamos lo que hemos hecho hasta el momento. Las cuantificaciones universales sobre MTS que teníamos en la fórmula de nuestra propiedad de interés las redujimos mediante la generación de pares de MTS tales que el primero no es refinamiento del segundo. En este proceso aplicamos optimizaciones en el número de relaciones a chequear para comprobar la existencia de un refinamiento. Una vez que tenemos este par de MTS, intentamos generar un LTS que sea implementación del segundo pero no del primero, y asi haga falsa la propiedad en la cual estamos interesados.

Un problema que surge al aplicar este mecanismo es que Alloy utiliza una forma determinista de realizar las búsquedas de instancias, que tiene como resultado la generación de siempre el mismo par de MTS. Si bien Alloy Analyzer ofrece la opcion de pedir otras instancias, esto nos demanda mucho tiempo, y en principio no se puede automatizar. Estas razones nos llevan a que nosotros generemos (automáticamente pero no a través de SAT solving) los MTS, los tomemos de a pares y se lo pasemos a Alloy para que éste analice la propiedad.

Algo que conviene preguntarse en este momento es: ¿cómo deben ser nuestros MTS candidatos? No tenemos una respuesta precisa para esta pregunta, aunque podemos pensar en algunas condiciones que estos MTS deben satisfacer:

- Tener como estado inicial al estado $s_0$.
- No ser bisimilar al MTS "vacío", esto es, al menos contener una transición probable que tenga por origen al estado $s_0$.
- Las transiciones requeridas deben estar incluidas en las probables.
- Deben ser conexos. Esto se debe a que podemos prescindir de los MTS no conexos, ya que analizar estos MTS es equivalente a analizar sólo la parte conexa de los mismos en la cual se encuentre el estado $s_0$ ; pero éste es otro MTS de menor tamaño, y por lo tanto ya debería haber sido contemplado (la visita a los MTS se realiza de manera creciente de acuerdo a su tamaño). La generación de los MTS candidatos, teniendo en cuenta las consideraciones anteriores, fue automatizada a través de un script, que se detalla en el Apéndice E.

Para poder apreciar el número de MTS a considerar, es importante notar que el mismo esta dado por la siguiente fórmula (notemos que ésta no contiene la reducción de MTS no conexos):

$$\sum_{i=0}^{|S|^2 \cdot L} \left[ \binom{|S|^2 \cdot |L|}{i} - \left\{ \begin{matrix} |S|^2 \cdot |L| - |S| \cdot |L| \\ i \end{matrix} \right\} \right] \cdot 2^i$$

donde,

$$\left\{ \begin{matrix} k \\ l \end{matrix} \right\} = \begin{cases} \binom{k}{l} & \text{si } l \leq k \\ 0 & \text{caso contrario} \end{cases}$$

y $S$ representa el número de estados y $L$ representa el número de acciones.

Donde $\binom{|S|^2 \cdot |L|}{i}$ representa la cantidad de transiciones probables con $i$ tuplas (tuplas de la forma, (Action,State,State)), $\left\{ {|S|^2 \cdot |L| - |S| \cdot |L| \atop i} \right\}$ representa la cantidad de transiciones probables de tamao $i$ que no contienen ninguna tupla que parta del estado $s_0$, y $2^i$ corresponde a la cantidad transiciones requeridas para cada conjunto de transiciones probables de tamao $i$.

La fórmula anterior presenta una cota superior, dado que no contempla la eliminación de los grafos no conexos.

Para eliminar los grafos no conexos utilizamos Alloy Analyzer. Modelamos la propiedad como sigue:

```
module Is_Conexo
open library

abstract sig Transition {
    trans: Action -> State -> State
}

pred is_conexo (t: Transition) {
    S0.*(t.trans[Action]) =
                    (t.trans[Action][State]+(t.trans[Action]).State)
}
```

También, podríamos eliminar algunas de las simetrías que se introducen por las acciones. Consideremos, por ejemplo, los siguientes MTS:

módulo el nombre de la acción estos MTS podrían representar el mismo contraejemplo para la propiedad mencionada. De todas maneras se debería tener particular cuidado en hacer el mismo tipo de reducción de simetría en todos los MTS involucrados. Esta técnica no la hemos implementado ya que el lenguaje que consideramos sólo contiene una acción.

Resumiendo,

- Trabajamos con 3 estados y 1 acción.
- La cantidad de MTS es de 18.954, mientras que los útiles son 14.690.
- Tenemos sólo 215.796.100 pares de MTS vs 387.420.489 que tendríamos sin las optimizaciones.

## 4.4 Metodología de Análisis

Hemos generado todos los candidatos de MTS, como asi también el mínimo conjunto de relaciones. Ahora, sólo nos resta ver cómo realizamos nuestro chequeo.

La idea es tomar dos MTS N y M, ver si se satisface el refinamiento entre estos dos MTS N y M, y si esto se cumple chequear si existe una implementación (i.e. un LTS L) de N, que no lo sea de M. En resumen, el proceso nos queda de la siguiente manera:

- tomar 2 MTS del conjunto de MTS generados, como por ejemplo los que se describen en el punto 2.

```
one sig mts1 extends MTS {}{
    init = S0
    req = A->S0->S0 + A->S0->S1
    trans = A->S0->S0 + A->S0->S1
}

one sig mts2 extends MTS {}{
    init = S0
    req = A->S1->S0
    trans = A->S0->S0 + A->S1->S0 + A->S1->S2
}
```

  Podemos almacenar esta especificaicón en módulo Alloy, digamos intanciasDeMTS

- analizar con Alloy si existe un refinamiento entre estos dos MTS; esto es, analizamos si el antecente de la propiedad de interés verdadera antes de continuar.

```
module AlloyInstanceA
open library
open RelationMinimal3State
open InstancesMts

pred a {
  some b1, b2: Biyeccion, r: Relation |
    mts1.init -> mts2.init in (b1.rel).(r.rel).(b2.rel) and
    StrongRefinement[(b1.rel).(r.rel).(b2.rel),
                        mts1.trans, mts1.req, mts2.trans, mts2.req]
}
run a for 1
```

  Notemos que debemos utilizar las biyecciones, para analizar todo el universo de relaciones.

- en caso en que Alloy Analyzer no genere una instancia, el predicado anterior no es satisfactible. Por lo tanto, no existe un refinamiento entre estos dos MTS. Si existe una accón de refinamiento descartamos los MTS, ya que no constituyen candidatos adecuados para ser contraejemplos de la propiedad.

- Si tenemos un par de MTS que hacen verdadero el antecedente de la propiedad de interés, proseguimos a analizar el consecuente de la propiedad de la siguiente manera:

```
module AlloyInstanceB
open RelationMinimal3State
open InstancesMts

pred a (lts1: LTS) {
  (some f1, f2: Biyeccion, r: Relation |
    mts2.init -> lts1.init in (f1.rel).(r.rel).(f2.rel) and
    StrongImplementation[(f1.rel).(r.rel).(f2.rel),
                            mts2.trans, mts2.req, lts1.trans]) and
  !(some f1, f2: Biyeccion, r: Relation |
    mts1.init -> lts1.init in (f1.rel).(r.rel).(f2.rel) and
    StrongImplementation[(f1.rel).(r.rel).(f2.rel),
                            mts1.trans, mts1.req, lts1.trans])
}
run a for 1
```

- Si este predicado es satisfactible nuevamente estos MTS no hacen falsa a la propiedad. Deberemos entonces descartarlos y comenzar con otros candidatos. Si en cambio este predicado resulta insatisfactible estamos en presencia de un candidato a contraejemplo de la propiedad que podría utilizarse para verificar que la propiedad es falsa.

  Este candidato a contraejemplo deberá ser chequeado a mano para corroborar si es un contraejemplo real o un contraejemplo espúreo.

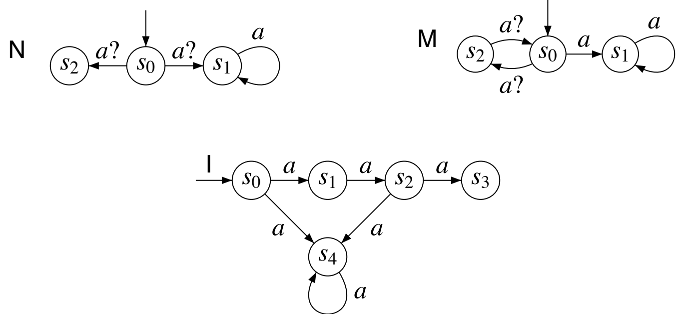
## 4.5 Algunos Ejemplos de contraejemplos

Alloy analyzer no ha encontrado contraejemplos de dicha propiedad. A continuación analizaremos algunos:

- Este ejemplo, al tener a lo sumo 2 acciones es muy simple para analizarlo. Notemos que el conjunto de implementaciones no sólo es finito, sino que sólo tiene 5 implementaciones. Con lo cual, su análisis es completo.

- Por su simplicidad, vemos que el MTS M acepta todas las implementaciones. Por lo tanto, podemos ver que basta con encontrar un N que no sea refinamiento de M, más aún, esto se reduce a encontrar un M que contenga una acción requerida.

- Otro contraejemplo que nos encuentra la herramienta es el siguiente. Pero veamos que éste, es un contraejemplo espúre ya que existe una implementación. Alloy no es capaz de encontrar este contraejemplo debido a que contiene 5 estados, y Alloy sólo tiene capacidad para analizar las implementaciones de hasta 3 estados. Veamos la implementación que hace a este contraejemplo, un contraejemplo espúreo.



## 4.6 Estimación de tiempo de análisis

Como habíamos visto, tenemos 18.954 MTS para 3 estados y 1 acción. Si consideramos todos los posibles pares de estos MTS, tenemos 359.254.116. Si bien este número llama la atención, debemos tener en cuenta que con las optimizaciones que hemos descripto el chequeo de un par de MTS se puede realizar en unos pocos segundos, a lo sumo un segundo. Luego, el tiempo aproximado del chequeo de la propiedad en el peor caso y con un segundo de tiempo de chequeo por cada MTS nos conduce a los siguientes tiempos de análisis en término de números de CPU.

| CPU | segundos |
|---|---|
| 1 | $3{,}59\ 10^{8}$ |
| 10 | $3{,}59\ 10^{7}$ |
| 50 | $7{,}18\ 10^{6}$ |
| 100 | $3{,}59\ 10^{6}$ |

Esto es, necesitamos casi 2 meses para analizar esta propiedad si contásemos con 100 CPU. La característica del análisis nos lleva a que el chequeo de cada par de MTS sea independiente, lo cual nos provee un entorno de trabajo altamente paralelizable. Además cada chequeo es barato en cuanto a memoria requerida.

## 4.7 Mejora al tiempo de análisis

Podemos optimizar el tiempo de análisis evitando el acceso de lectura y escritura a disco. Si observamos detenidamente, una gran parte de nuestro proceso de análisis requiere lectura y escritura de archivos. Podemos optimizar el tiempo de análisis, si en lugar de escribir en disco, escribimos en memoria. Un truco útil para conseguir esto es mediante la creación de un disco ram que es accedido como un disco común. En el apéndice E, detallamos los pasos para realizarlo. Para más detalles el lector puede ver [Dis].

Con nuestra optimización, hemos podido reducir el tiempo de análisis a un mes, si contásemos con 100 CPU.

## 4.8 Una alternativa utilizando DynAlloy

Un inconveniente con el proceso de análisis descripto está dado por la limitación en el número de instancias que Alloy Analyzer soporta. Una alternativa para intentar resolver este problema consiste en la utilización de una extensión de Alloy llamada DynAlloy [FPB+05]. DynAlloy permite describir propiedades de sistemas usando acciones. Las acciones nos permiten especificar propiedades dinámicas apropiadamente, en particular, propiedades en lo que respecta a trazas de ejecuciones, en el estilo de lógica dinámica. DynAlloy extiende la sintaxis de Alloy con acciones y programas y aserciones de corrección parcial para esto.

Esencialmente la alternativa que describimos consiste en generar a través de un programa DynAlloy las relaciones candidatas a refinamiento entre MTS. Esta alternativa nos permite trabajar con relaciones sobre conjuntos de estados más grandes, pero con un costo de generación más alto.

### 4.8.1 Análisis de la propiedad mediante DynAlloy

La idea descripta anteriormente requiere la especificación de un programa DynAlloy para la generación de relaciones entre MTS. El modelo DynAlloy que contiene dicho programa se detalla a continuación. El lector puede familiarizarse con DynAlloy viendo el Apéndice B.

Notemos cómo definimos las diferentes propiedades con DynAlloy,

```
/* ASERCIONES */

pred PreAssert[i: LTS, r: State -> State] { no i.trans and no r}

pred postAssert[i: LTS, r: State -> State] {
    StrongRefinement[r, MTS1.trans, MTS1.req, i.trans]
}

pred existeAux[i: LTS] {
    some r: State -> State |
        StrongImplementation[r, MTS2.trans, MTS2.req, i.trans]
}

assertCorrectness BInvA[i: LTS, r: State -> State] {
    pre = { PreAssert[i,r] }
    program = { (AgregarArc[i])*;[existeAux[i]]?;(AgregarPar[r])* }
    post = { postAssert[i',r'] }
}

/* ASERCIONES */

pred PreAssert[i: LTS, r: State -> State] { no i.trans and no r}

pred postAssert[i: LTS, r: State -> State] {
    StrongRefinement[r, MTS1.trans, MTS1.req, MTS2.trans, MTS2.req]
}

assertCorrectness BInvA[i: LTS, r: State -> State] {
    pre = { PreAssert[i,r] }
    program = { (AgregarArc[i])*;[existeAux[i]]?;(AgregarPar[r])* }
    post = { !postAssert[i',r'] }
}
```

# 5 Conclusiones y trabajo futuro

## 5.1 Conclusiones

La finalidad de este trabajo ha sido el estudio de la viabilidad del uso de SAT solving para el análisis de diferentes tipos de propiedades sobre sistemas de transiciones etiquetadas, y una variante con soporte de parcialidad, los sistemas de transiciones modales. Hemos elegido como lenguaje para la caracterización de estos sistemas a Alloy, un lenguaje formal fuertemente basado en la noción de relación. Alloy nos brindó interesantes ventajas como lenguaje de especificaciones de LTS y MTS. En primer lugar, Alloy ofrece una sintaxis que está a un nivel de abstracción adecuado para la caracterización de LTS y MTS, a diferencia del lenguaje aceptado comúnmente por los SAT solvers (fórmulas proposicionales, o fórmulas proposicionales en forma normal conjuntiva). En particular, Alloy soporta clausura transitiva como un operador provisto para la descripción de expresiones relacionales, que es indispensable para la definición de varios tipos de relaciones de refinamiento e implementación entre MTS y LTS. En segundo lugar, la herramienta de análisis asociada a Alloy, Alloy Analyzer, implementa una codificación de relaciones en fórmulas proposicionales muy eficiente, además de proveer mecanismos de aceleración en el análisis externos a los SAT solvers, como la eliminación de instancias isomorfas y el aprovechamiento de cotas de relaciones en la traducción.

El resultado de nuestro estudio muestra que SAT solving es una herramienta viable para el análisis de refinamientos entre MTS y LTS, a pesar de ser menos eficiente que los algoritmos específicos para este tipo de análisis, como era previsible. Esto se debe a la naturaleza combinatoria del análisis basado en SAT solving, un mecanismos de constraint solving, esencialmente de fuerza bruta en la búsqueda de instancias que satisfagan las fórmulas en las especificaciones. No sólo ha demostrado ser viable, sino que mediante la utilización directa del motor relacional subyacente a Alloy, denominado KodKod, los tiempos de análisis resultan comparables a los de la aplicación de algoritmos específicos para chequeo de refinamientos, como los implementados en las herramientas MTSA y MTSChecker.

La caracterización de MTS, LTS y relaciones de refinamiento en Alloy ofrece algunas ventajas que MTSA y MTSChecker no brindan. En primer lugar, las especificaciones pueden consultarse de maneras más variadas que sólo preguntar por la existencia de una relación de refinamiento. Hemos mostrado algunos ejemplos de esto en este trabajo. En segundo lugar, la declaratividad natural de nuestra especificación Alloy, que nos permite expresar directamente las definiciones de las relaciones de refinamiento

(en lugar de implementar algoritmos para su chequeo) permite utilizar nuestra especificación para contrastar resultados con implementaciones de algoritmos de chequeo de refinamiento; es decir, utilizar la herramienta como un accesorio para la depuración de implementaciones de algoritmos de análisis de MTS y LTS.

Buena parte de nuestro trabajo se concentró en el análisis de propiedades de alto orden de MTS y LTS. La motivación principal de esto ha sido el estudio de una propiedad, la (supuesta) completitud de refinamiento fuerte con respecto a implementaciones en MTS. Esta propiedad se creyó verdadera durante un tiempo, e incluso se publicó una demostración de la misma. Sin embargo, la propiedad es falsa, y los contraejemplos de la misma son relativamente pequeños. Esto sugiere utilizar Alloy para analizar la verdad de esta propiedad en modelos acotados, o en otras palabras, intentar encontrar contraejemplos de la propiedad en dominios acotados. Este tipo de análisis resultó ser muy costoso. Simplificamos el problema a la búsqueda de *potenciales* contraejemplos, ya que comprobar si un contraejemplo no es espúreo requiere en algunos casos la exploración de un número infinito de LTS. Luego de simplificar el problema, y realizar una variedad de optimizaciones, incluyendo la generación de un conjunto mínimo de relaciones de representación a partir de las posibles relaciones de refinamiento entre MTS y el ahorro de accesos a disco para el almacenamiento y lectura de resultados parciales, logramos reducir los tiempos de análisis significativamente.

## 5.2 Trabajo Futuro

Debido a limitaciones de tiempo, han quedado varias tareas importantes por concluir, que nos ofrecen líneas para continuar trabajando. En primer lugar, debemos completar el estudio experimental de los mecanismos de análisis desarrollados. Esta tarea requerirá considerable tiempo de cómputo, pero no resulta ser complicada de llevar adelante. Queremos además estudiar algunas propiedades ligadas a MTS, de particular interés para la búsqueda de instancias acotadas de especificaciones. Una de estas propiedades, que nos permitiría saber cuál es el tamaño máximo de LTS que es necesario explorar para el chequeo de completitud de relaciones de refinamiento respecto de implementaciones de MTS es la siguiente: Si la propiedad es violada. ¿Existe una cota $k$ para el tamaño de la implementación que hacer violar la propiedad? Es decir, ¿Existe $k$ tal que $k \geq min\{\#i | i \in I(M) \setminus I(N)\}$?

Una de las mejoras al análisis que manejamos en este trabajo consiste en la eliminación (o poda en la búsqueda) de MTS que resultan de permutaciones en las etiquetas de las transiciones. Nuevamente debido a limitaciones de tiempo no hemos podido explotar al máximo esta alternativa de mejora. Continuar explotándola forma parte de nuestras tareas de trabajo futuro.

# A La herramienta MTSA

La herramienta MTSA permite verificar propiedades y analizar modelos parciales (MTS) utilizando model checking. La herramienta Modal Transition System Analyzer (MTSA) que soporta la construcción, elaboración y análisis de MTS se desarrolló como una extensión del ya existente Labeled Transition System Analyzer (LTSA).

## A.1 LTSA - Labeled Transition System Analyzer

LTSA es una herramienta para verificación y validación de propiedades para sistemas concurrentes. En LTSA un sistema se modela como un conjunto de máquinas de estados finitos que interactúan entre sí. Las propiedades que se desean estudiar sobre el sistema, también se modelan en máquinas de estados.

LTSA realiza la composición del modelo del sistema con la propiedad deseada y sobre esta composición realiza una búsqueda exahustiva de posibles estados erróneos. Más formalmente, cada uno de los componentes de una especificación se describe como un Sistema de Transición Etiquetado (LTS), que contiene todos los estados a los que el componente puede llegar y todas las transiciones que puede realizar. Sin embargo, la descripción explícita de un LTS en términos de sus estados, conjunto de etiquetas de acción y relación de transición es complicada para cualquier sistema que no sea muy pequeño. En consecuencia, LTSA una notación al estilo de las álgebras de procesos para describir el comportamiento de las componentes. Tal notación se denomina FSP (por "finite state process", proceso de estudio finito).

Un proceso se define mediante un listado de definiciones de procesos locales separados por coma y finalizados por punto. Por ejemplo:

```
SWITCH = OFF,
OFF = (on -> ON),
ON = (off -> OFF).
```

describe el comportamiento de un interruptor. ERROR y STOP son procesos primitivos.

- Prefijos de acciones ->: si x es una acción y P un proceso, entonces (x -> P) describe un proceso que inicialmente realiza la acción x y, a continuación, se comporta exactamente como se describe por P.

- Elección |: Si x e y son acciones entonces (x -> P | y -> Q) describe un proceso que inicialmente responde a cualquiera de las acciones x o y. Después de haber realizado esta primera acción, el comportamiento continúa según según se describe en P o Q de acuerdo a si la primera acción realizada fue x o y respectivamente.
- Elección con guarda: La elección con guarda es posible que las distintas alternativas en una elección sean habilitadas o deshabilitadas según el valor de verdad de una guarda. Por ejemplo, en el proceso (when x B -> P | y -> Q) representaría dos opciones en la elección si la guarda B es verdadera. En caso de que sea falso, sólo la elección por y puede ser realizada.
- Extensión **+**: El alfabeto de un proceso es el conjunto de acciones en las que éste puede participar. P + S extiende el alfabeto del proceso P con las acciones en el conjunto S.
- Recursión: El comportamiento de un proceso puede ser definido recursivamente. La recursividad puede ser directamente en términos del proceso que se define, o indirectamente, en términos de otro proceso.Un ejemplo de esto, es el interruptor presentado recientemente.

También existen procesos compuestos, éstos se definen mediante la composiciń paralela de uno o más procesos. La definición de un proceso compuesto es precedida por ||. Veamos algunos operados útiles para la buena modelización de estos procesos compuestos:

- Composición paralela ||: Si P y Q son procesos, entonces (P || Q) representa la ejecución de manera concurrente entre P y Q, donde las acciones comunes son realizadas conjuntamente por ambos procesos en una única transición, y las no comunes de manera independiente por cada uno de ellos.
- Replicador **forall**: forall[i:1..N] P(i) corresponde a la composición paralela de las n replicaciones del proceso P de la siguiente manera: (P(1) || ... || P(N))

Existen además otros operadores que ayudan a modelar los procesos, entre ellos los que tienen que ver con el renombrado de las acciones. De particular interés para nosotros, son los operadores de ocultamiento e interfaz:

- Ocultamiento \ El proceso P\\$\{a_1, \ldots, a_x\}$ se comporta como el proceso P sólo que las acciones $a_1 \ldots a_x$ son renombradas por la acción "silenciosa" tau. La acción tau no es una acción que se comparte entre procesos, por lo tanto no pueden sincronizarse mediante ella.
- Interfaz **@**: Es el dual del operador anterior. P@$\{a_1, \ldots, a_x\}$ se comporta como el proceso P donde sólo las acciones $a_1$ .. $a_x$ son visibles y el resto son renombradas a tau.

El lector interesado puede encontrar más información en [KM06].

## A.2 MTSA - Modal Transition System Analyzer

MTSA es una extensión de LTSA que además permite verificar automáticamente diversas relaciones de refinamiento entre MTS. Para poder especificar MTS el lenguaje de FSP se extiende de manera de poder especificar acciones probables. Las acciones probables se describen como cualquier acción sólo que se agrega ? al final del nombre.

## A.3 MTSChecker

MTSChecker es una una herramienta stand alone que permite verificar relaciones de refinamiento entre MTS. MTSChecker es el motor de MTSA.

La sintáxis de MTSChecker utiliza una sintaxis semejante a la de FSP. Como MT-SA, introduce las acciones probables agregando al final del nombre el signo ?. Además la acción "silenciosa" tau, puede representarse explícitamente.

# B Las herramientas Alloy y DynAlloy

## B.1 Alloy

Alloy Analyzer es un SAT solver que provee un lenguaje relacional. Su motor es un nuevo SAT basado en el model finder KodKod.

Un modelo en Alloy está compuesto por: las signaturas, utilizadas para la construcción de nuevos tipos, y una variedad de fórmulas.

### B.1.1 Gramática y semántica de Alloy

problem ::= decl form
decl ::= var : typexpr
typexpr ::= type
| type → type
| type ⇒ typexpr

form ::= expr in expr (subset)
| !form (neg)
| form && form (conj)
| form || form (disj)
| all v : type/form (univ)
| some v : type/form (exist)

expr ::= expr + expr (union)
| expr & expr (intersection)
| expr  expr (diference)
| ~ expr (transpose)
| expr.expr (navigation)
| +expr (transitive closure)
| v : t/form (set former)
| Var

Var ::= var (variable)
| Var[var] (application)

M : form → env → Boolean
X : expr → env → value
env = (var + type) → value
value = (atom × · · · × atom)+
(atom → value)

M [a in b]e = X[a]e ⊆ X[b]e
M [!F ]e = ¬ M [F ]e
M [F && G]e = M [F ]e ∧ M [G]e
M [F ||G]e = M [F]e ∨ M [G]e
M [all v : t/F] =
⋀M [F](e ⊕ v→{x})/x ∈ e(t)
M[some v : t/F] =
⋁M [F](e ⊕ v→{x})/x ∈ e(t)

X[a + b]e = X[a]e ∪ X[b]e
X[a & b]e = X[a]e ∩ X[b]e
X[a  b]e = X[a]e \ X[b]e
X[~ a]e = {$\langle x, y\rangle$ : $\langle y, x\rangle$  X[a]e}
X[a.b]e = X[a]e;X[b]e
X[+a]e = the smallest r such that
r;r ⊆ r and X[a]e ⊆ r
X[{v : t/F}]e =
{x  e(t)/M [F](e ⊕ v→x)}
X[v]e = e(v)
X[a[v]]e = {$\langle y_1, ..., y_n\rangle$ / $\exists x\langle x, y_1, ..., y_n\rangle$ ∈ e(a) ∧$\langle x\rangle$ ∈ e(v)}

### B.1.2 Signaturas

Una signatura introduce un tipo básico y una colección de relaciones (que son llamados campos), junto con los tipos de los campos y las restricciones sobre estos valores. Por ejemplo,

```
sig Objeto {}
```

Objeto se introduce como un tipo no interpretado (o conjunto de átomos indivisibles). Además una signatura puede heredar campos y condiciones de otra signatura. Por ejemplo,

```
sig Predicado extends Objeto {
    def: set Objeto
}
```

declara a Predicado como un subconjunto de Objeto, mientras que el campo def declara una relación de Predicado a Objeto.

La palabra abstract, permite definir una signatura abstracta por lo que no va a tener ninguna instancia del conjunto de Objeto. En general lo utilizamos cuando queremos declarar signaturas concretas a partir de signaturas abstractas. Por ejemplo, las siguientes signaturas se extienden de Objeto:

```
abstract sig Objeto {}
sig ElemA extends Objeto {}
sig ElemB extends Objeto {}
```

así, obtenemos instancias de Objeto a partir de las instancias de ElemA y ElemB. Por otro lado, podemos especificar si queremos sólo una instancia; a ésto lo realizamos de la siguiente manera:

```
abstract sig Objeto {}
one sig ElemA extends Objeto {}
```

### B.1.3 Fórmulas y declaraciones

Una fórmula está formada por expresiones de Alloy. El valor de cualquier expresión en Alloy es siempre una relación -que es una colección de tuplas-. Cada elemento de esa tupla es atómica y pertenece a algún tipo básico. Una relación puede tener aridad uno o más. Las relaciones son tipadas. Los conjuntos son vistos como relaciones unarias. Las relaciones se puede combinar con una variedad de operadores para formar expresiones. Las expresiones quantificadas transforman una expresión en una fórmula. La fórmula *no exp* es verdadera cuando *exp* denota una relación que no contiene tuplas. Del mismo modo, *some exp*, *lone exp*, y *one exp* son ciertas cuando tiene algunas, al menos una, y exactamente una tupla respectivamente. Las fórmulas también pueden construirse mediante comparaciones de operadores relacionales. Alloy provee operadores de lógica estandar:

- && (conjunción)
- || (disyunción)
- => (implicación)
- ! (negación)

Una declaración es una fórmula *v op exp* que consta de una variable *v*, un operador de comparación *op*, y una expresión arbitraria *exp*. Las fórmulas cuantificadas consisten de un cuantificador, una lista separada por comas de las declaraciones, y una fórmula. Además del cuantificador universal y existencial (quantificadores *all* y *some*), y *one* (exactamente uno).

Alloy, también provee los operadores relacionales básicos; subset (in), igualdad (: o =) y sus negaciones (!:, !in, !=). Además provee otros operadores relacionales los cuales explicaremos con más detalle. Sea p que contiene una relación con m-tuplas de la forma $(p_1,...,p_m)$ y sea q una relación que contenga n-tuplas de la forma $(q_1,...,q_n)$.

- **p + q, p & q** y **p - q**: la unión, la intersección y la diferencia combinan dos relaciones del mismo tipo, vistas como conjuntos de tuplas.(i.e. m=n)

- **p -> q**: el producto relacional de p y q resulta en una nueva relación r = $(p_1, ..., p_m, q_1,..., q_n)$ para cada combinación de tupla de p y tupla q

- **p.q**: el join de p y q, es la relación que contiene tuplas de la forma $(p_1,...,p_{m-1},q_2,...,q_n)$ para cada par de tuplas donde el primero es una tupla de p, y la segunda es una tupla de q y el último átomo de la primera tupla coincide con el primer átomo de la segunda tupla.

- **p[q]**: la relación unión de p y q, es la relación que contiene tuplas de la forma $(q_1,...,q_{n-1},p_2,...,p_m)$ para cada par de tuplas donde el primero es una tupla de p, y la segunda es una tupla de q y el primer átomo de la segunda tupla coincide con el último átomo de la primer tupla. Alternativamente, p[q] puede ser escrito como q.p

- **ˆp**: la clausura transitiva de p es la menor relación que contiene a p y es transitiva. Transitivo significa que si p contiene (a, b) y (b, c) entonces, también contiene (a, c). Notemos que p es una relación binaria y que la relación resultante es también una relación binaria.

- ***p**: la clausura reflexiva transitiva de p es la menor relación que contiene p y es a la vez transitiva y reflexiva, lo que significa que todas las tuplas de la forma (a, a) están presentes. Una vez más, p es una relación binaria y la relación resultante es también una relación binaria.

- **~p**: la transpuesta de una relación r forma una nueva relación que tiene el orden de los átomos en sus tuplas invertido. Por lo tanto, si p contiene (a, b), entonces p contendrá (b, a)

### B.1.4 Funciones, hechos, aserciones y predicados

Una función (fun) es una fórmula parametrizada que vincula sus parámetros a las expresiones cuyos tipos coinciden con el tipo declarado del parámetro. De forma predeterminada, una función devuelve un valor booleano, el valor de la fórmula en su cuerpo, aunque una función puede devolver un valor relacional (no booleano) para que esto ocurra hay que declararlo explícitamente como se puede observar en el siguiente ejemplo.

```
fun isObject (x: Object, p: Predicado) {
    x in p.def
}
```

Un hecho (fact) es una fórmula que no tiene argumentos, y no necesita ser invocado explícitamente. Los hechos son propiedades globales de la especificación que deben ser siempre verdaderas.

```
fact AllObjectoInPredicado {
    all x:Objecto | some p: Predicado | x in Predicado
}
```

Una aserción (assert) es una fórmula cuya corrección necesita ser comprobada, asumiendo los hechos en el modelo.

```
assert ObjectIn2Predicado {
    some x:Objecto | some p1, p2: Predicado |
            p1!=p2 and x in p1.def and x in p2.def
}
```

Un predicado (pred) es una fórmula que necesita ser corroborado al igual que la aserción, asumiendo los hechos en el modelo.

```
pred PredicadoTotal (p:Predicado) {
    all x: Objeto | x in p.def
}
```

### B.1.5 Corridas y chequeos

Tanto las aserciones como los predicados, deben ser corroborados. Los predicados se corroboran mediante corridas (run), mientras que las aserciones se corroborar mediante chequeos (check). Por lo que a la hora de querer corroborar, debemos tener en cuenta las siguientes cuatro posibilidades:

- Si run(x) produce un ejemplo, eso significa que x es cierto en virtud de “algunas” circunstancias. Pero no necesariamente en todas las circunstancias.
- Si run(x) no produce un ejemplo, x significa que es falso en virtud de todas las posibles circunstancias.
- Si check(x) produce una contraejemplo, x significa que es falsa en virtud de “algunas” circunstancias. Pero no necesariamente en todas las circunstancias.
- Si check(x) no produce un contraejemplo, eso significa que x es cierta en virtud de todas las posibles circunstancias.

### B.1.6 Skolemización de Relaciones

Muchas veces, las fórmulas pueden reducirse a fórmulas equivalentes sin el uso de quantificadores. Esta reducción se denomina skolemización y se basa en la introducción de una o más constantes o funciones que capturan la fórmula cuantificada mediante sus valores. Consideremos el siguiente ejemplo,

```
sig A { r: lone B }
sig B {}

fact {
    some x: A | no x.
}
```

El “some” de la fórmula puede ser expresado equivalentemente por:

```
x' in A && no x'.r
```

x' es la relación skolemizada en este caso. El cuantificador existencial "some" no es necesario porque el análisis se basa en la búsqueda de la existencia de la relación skolemizada.

## B.2 DynAlloy

DynAlloy es una extensión del lenguaje de especificación de Alloy para describir propiedades dinámicas de los sistemas usando acciones. Las acciones nos permiten especificar adecuadamente propiedades dinámicas, en particular, propiedades en relación con trazas de ejecución.

### B.2.1 Gramática y semántica de DynAlloy

La sintáxis de fórmulas de DynAlloy extiende a la sintáxis de Alloy mediante las siguientes reglas para construcción de sentencias de correción parcial (partial correctness statements)

formula ::= ... |{formula} program {formula} (partial correctness)

program ::= formula, formula (x) (atomic action)
| formula? (test)
| program + program (non-deterministic choice)
| program;program (sequential composition)
| program (iteration)

### B.2.2 Predicados

DynAlloy permite describir ejecuciones, vistas desde un punto de vista de trazas. Por lo tanto, el funcionamiento de DynAlloy consiste en describir el estado inicial del sistema (supongamos que lo denominamos $\alpha$). Luego, describir las operaciones que deben ir ocurriendo, esto lo realizamos especificando los pares de estados que se van relacionando mediante la ejecución de una traza. Podemos suponer que los operadores son Oper 1 y Oper 2. Finalmente debemos especificar el estado final del sistema (llamemosle $\beta$). Lo cual nos lleva a que la especificación pueda ser escrita tan simple y elegante como sigue:

$$\{\alpha\}$$
$$(\text{Oper 1} + \text{Oper 2})^*$$
$$\{\beta\}$$

Para más detalles, el funcionamiento de DynAlloy es introducido en [FGPA05].

## B.3 KodKod

Alloy es muy utilizado debido al buen soporte de lógica relacional y la plena automaticación de sus análisis. Muchas veces se ha querido integrar Alloy con otras herramientas y esto ha fallado por diversas razones. Algunas de éstas las detallamos a continuación,

- una API limpia,

- apoyo a los casos parciales, y
- un mecanismo para compartir subformulas y subexpresiones.

Es por esto que KodKod está diseado expresamente como un componente plugin, el motor relacional KodKod supera estas limitaciones. A diferencia de los analizadores, KodKod proporciona una interfaz simple para la construcción y el análisis de fórmulas de Alloy y que emplea un sólido régimen compartido para la explotación de fórmulas y expresiones.

### B.3.1 Sintáxis abstracta de KodKod

```
problem := univDecl relDecl formula

univDecl := {atom[, atom]}
relDecl := relVar :arity [constant, constant]
varDecl := quantVar : expr

constant := {tuple}
tuple := atom[, atom]

arity := 1 | 2 | 3 | 4 | ...
atom := identier
relVar := identier
quantVar := identier

formula := expr in expr (subset)
| some expr (non-empty)
| one expr (singleton)
| no expr (empty)
| lone expr (empty or singleton)
| not formula (negation)
| formula and formula (conjunction)
| formula or formula (disjunction)
| all varDecl | formula (universal)
| some varDecl | formula (existential)

expr := expr + expr (union)
| expr & expr (intersection)
| expr - expr (diference)
| expr.expr (join)
| expr -> expr (product)
| ~expr (transpose)
| ^ expr (closure) | {varDecl | formula} (comprehension)
| relVar (relation)
| quantVar (quantied variable)
```

## B.4 Cotas

Lo bueno de KodKod, es que permite el chequeo de propiedades parciales. Para ésto utiliza un sistema de cotas. Este sistema de cotas permite definir los límites de sus universos, es decir definir qué elementos hay en cada uno de los conjuntos.

Primero generamos todos los átomos que tendrá nuestro universo,

```
final List<String> atoms = new LinkedList<String>();

for (int i = 0; i < states; i++) {
 atoms.add("S" + i);
}

for (int i = 0; i < actions; i++) {
 atoms.add("act" + i);
}

atoms.add("tau");
```

Luego, generamos el universo sobre el cuál vamos a trabajar,

```
final TupleFactory factory = universe.factory();
final Bounds b = new Bounds(universe);
final String stateMax = "S" + (states - 1);
```

mediante, la sentencia *boundExactly* definimos el conjunto de exactamente todos los átomos que se encuentran en este rango, mientras que si ponemos la sentencia *bound*, definimos el conjunto que a lo sumo tiene todos los átomos dentro del rango.

```
b.boundExactly(this.state,
        factory.range(factory.tuple("S0"), factory.tuple(stateMax)));
b.bound(this.action,
        factory.range(factory.tuple("act0"), factory.tuple("tau")));
b.boundExactly(this.tau,
        factory.range(factory.tuple("tau"), factory.tuple("tau")));
```

además podemos crear tuplas, para definir universos de signatuas, como por ejemplo:

```
// Relations
final List<Tuple> relUpperBound = new LinkedList<Tuple>();

String stateI = "";
String stateF = "";

for (int i = 0; i < states; i++) {
 for (int j = 0; j < states; j++) {
  stateI = "S"+j;
  stateF = "S"+i;
    relUpperBound.add(factory.tuple(stateI, stateF));
 }
}

b.bound(this.rel.rel, factory.setOf(relUpperBound));
```

En ésta definimos el conjunto de átomos para las relaciones binarias sobre estados. Luego, la relación *rel* será como mínimo la relación vacía, y a lo sumo, contendrá todos los pares de estados que definimos en relUpperBound.

# C Ejemplos

## C.1 MTSA

A continuación se listan los códigos de los ejemplos representados en la Tabla 3.6 en MTSA.

```
P1 = (a -> b -> P1 | a -> b -> P1 ).

Q1 = Q10,
Q10 = (a -> Q12),
Q12 = (b -> Q13),
Q13 = (a -> b -> Q10 | a -> Q12).

P2 = P20,
P20 = (a -> P21),
P21 = (b -> c -> STOP | b -> d -> STOP).

Q2 = Q20,
Q20 = (a -> b -> c -> STOP | a -> b -> d -> STOP).

P3 = P30,
P30 = (a -> P31),
P31 = (b -> STOP | _tau -> c -> STOP)\{_tau}.

Q3 = Q30,
Q30 = (a -> Q31 | a -> Q33),
Q31 = (b -> STOP | _tau -> Q33)\{_tau},
Q33 = (c -> STOP).

P4 = (a -> b -> P4 | a -> b -> P4).

Q4 = Q40,
Q40 = (_tau -> Q42),
Q42 = (a -> Q43),
Q43 = (b -> Q42 | _tau -> a -> Q40)\{_tau}.

P5 = (a -> b -> P5).

Q5 = (_tau -> a -> _tau -> b -> Q5)\{_tau}.

P6 = P60,
P60 = (b -> a -> P61),
P61 = (c -> P61 | _tau -> P60)\{_tau}.

Q6 = Q60,
```

```
Q60 = (b -> Q61),
Q61 = (a -> Q60 | a -> Q62),
Q62 = (c -> Q62 | _tau -> Q60)\{tau}.

P7 = P70,
P70 = (a -> P71),
P71 = (b -> P72),
P72 = (b? -> P71 | c? -> P70).

Q7 = Q70,
Q70 = (a -> Q71),
Q71 = (b -> Q72),
Q72 = (b? -> Q71 | c -> Q70).

P8 = (a -> b -> STOP | _tau? -> b? -> STOP)\{_tau?}.

Q8 = (a -> _tau -> b -> STOP | b? -> STOP)\{_tau?}.

P9 = P90,
P90 = (a -> tau? -> P92)\{_tau?},
P92 = (b -> P90 | c? -> P92).

Q9 = Q90,
Q90 = (a -> _tau -> Q92)\{_tau},
Q92 = (b? -> Q90 | c? -> Q92).

P10 = P100,
P100 = (a -> P101),
P101 = (b -> P102),
P102 = (b? -> P101 | c? -> P100).

Q10 = (a -> b -> STOP).

P11 = (a -> _tau? -> b? -> P11)\{_tau?}.

Q11 = (a -> b -> Q11).

P12 = P120,
P120 = (_tau? -> a? -> STOP | a -> P122)\{_tau?},
P122 = (b -> STOP | _tau -> STOP)\{_tau}

Q12 = Q120,
Q120 = (a -> Q121 | a -> STOP),
Q121 = (b -> STOP | _tau -> STOP)\{_tau}.

P13 = P130,
P130 = (_tau? -> a? -> STOP | a -> P132)\{_tau?,a?},
P132 = (b -> STOP | c -> STOP).

Q13 = (a -> b -> STOP | a -> c -> STOP).

P14 = P140,
P140 = (a -> b -> P143),
P143 = (c -> P140 | _tau? -> c? -> P140)\{_tau?}.

Q14 = (a -> _tau -> b -> c -> Q14)\{_tau}.

P20 = P200,
P200 = (a -> P201 | b-> P202),
P201 = (c -> P202 | b -> P204),
P202 = (b -> P201 | c -> P203 | a -> P204),
P203 = (d -> P205 | d -> P203 | e -> P205),
```

```
P204 = (a -> P203 | d -> P202 | c -> P204),
P205 = (f -> P201 | d -> P204 | d -> P203).

Q20 = Q200,
Q200 = (a -> Q201 | b-> Q202),
Q201 = (c -> Q202 | b -> Q204),
Q202 = (b -> Q201 | c -> Q203 | a -> Q203),
Q203 = (d -> Q205 | d -> Q203 | e -> Q204),
Q204 = (a -> Q203 | d -> Q202 | c -> Q204),
Q205 = (f -> Q201 | d -> Q204 | d -> Q203).
```

## C.2 MTSChecker

A continuación se listan los códigos de los ejemplos representados en la Tabla 3.6 en MTShecker.

```
P1 = (a -> b -> P1 | a -> b -> P1 ).

Q1 = Q10,
Q10 = (a -> Q12),
Q12 = (b -> Q13),
Q13 = (a -> b -> Q10 | a -> Q12).

P2 = P20,
P20 = (a -> P21),
P21 = (b -> c -> STOP | b -> d -> STOP).

Q2 = Q20,
Q20 = (a -> b -> c -> STOP | a -> b -> d -> STOP).

P3 = P30,
P30 = (a -> P31),
P31 = (b -> STOP | _tau -> c -> STOP).

Q3 = Q30,
Q30 = (a -> Q31 | a -> Q33),
Q31 = (b -> STOP | _tau -> Q33),
Q33 = (c -> STOP).

P4 = (a -> b -> P4 | a -> b -> P4).

Q4 = Q40,
Q40 = (_tau -> Q42),
Q42 = (a -> Q43),
Q43 = (b -> Q42 | _tau -> a -> Q40).

P5 = (a -> b -> P5).

Q5 = (_tau -> a -> _tau -> b -> Q5).

P6 = P60,
P60 = (b -> a -> P61),
P61 = (c -> P61 | _tau -> P60).

Q6 = Q60,
Q60 = (b -> Q61),
Q61 = (a -> Q60 | a -> Q62),
Q62 = (c -> Q62 | _tau -> Q60).

P7 = P70,
```

```
P70 = (a -> P71),
P71 = (b -> P72),
P72 = (b? -> P71 | c? -> P70).

Q7 = Q70,
Q70 = (a -> Q71),
Q71 = (b -> Q72),
Q72 = (b? -> Q71 | c -> Q70).

P8 = (a -> b -> STOP | _tau? -> b? -> STOP).

Q8 = (a -> _tau -> b -> STOP | b? -> STOP).

P9 = P90,
P90 = (a -> _tau? -> P92),
P92 = (b -> P90 | c? -> P92).

Q9 = Q90,
Q90 = (a -> _tau -> Q92),
Q92 = (b? -> Q90 | c? -> Q92).

P10 = P100,
P100 = (a -> P101),
P101 = (b -> P102),
P102 = (b? -> P101 | c? -> P100).

Q10 = (a -> b -> STOP).

P11 = (a -> _tau? -> b? -> P11).

Q11 = (a -> b -> Q11).

P12 = P120,
P120 = (_tau? -> a? -> STOP | a -> P122),
P122 = (b -> STOP | _tau -> STOP).

Q12 = Q120,
Q120 = (a -> Q121 | a -> STOP),
Q121 = (b -> STOP | _tau -> STOP).

P13 = P130,
P130 = (_tau? -> a? -> STOP | a -> P132),
P132 = (b -> STOP | c -> STOP).

Q13 = (a -> b -> STOP | a -> c -> STOP).

P14 = P140,
P140 = (a -> b -> P143),
P143 = (c -> P140 | _tau? -> c? -> P140).

Q14 = (a -> _tau -> b -> c -> Q14).

P20 = P200,
P200 = (a -> P201 | b-> P202),
P201 = (c -> P202 | b -> P204),
P202 = (b -> P201 | c -> P203 | a -> P204),
P203 = (d -> P205 | d -> P203 | e -> P205),
P204 = (a -> P203 | d -> P202 | c -> P204),
P205 = (f -> P201 | d -> P204 | d -> P203).

Q20 = Q200,
Q200 = (a -> Q201 | b-> Q202),
```

```
Q201 = (c -> Q202 | b -> Q204),
Q202 = (b -> Q201 | c -> Q203 | a -> Q203),
Q203 = (d -> Q205 | d -> Q203 | e -> Q204),
Q204 = (a -> Q203 | d -> Q202 | c -> Q204),
Q205 = (f -> Q201 | d -> Q204 | d -> Q203).
```

## C.3 Alloy

A continuación se listan los códigos de los ejemplos representados en la Tabla 3.6 en Alloy.

```
one sig P1 extends LTS {} {
    init = S0
    trans = A -> S0 -> S1 + B -> S1 -> S0 + A -> S0 -> S2
            + B -> S2 -> S0
}

one sig Q1 extends LTS {} {
    init = S0
    trans = A -> S0 -> S2 + B -> S2 -> S3 + A -> S3 -> S2
            + A -> S3 -> S1 + B -> S1 -> S0
}

one sig P2 extends LTS {} {
    init = S0
    trans = A -> S0 -> S1 + B -> S1 -> S2 + C -> S2 -> S4
            + B -> S1 -> S3 + D -> S3 -> S5
}

one sig Q2 extends LTS {} {
    init = S0
    trans = A -> S0 -> S1 + B -> S1 -> S3 + C -> S3 -> S5
            + A -> S0 -> S2 + B -> S2 -> S4 + D -> S4 -> S6
}

one sig P3 extends LTS {} {
    init = S0
    trans = A -> S0 -> S1 + B -> S1 -> S2 + Tau -> S1 -> S3
            + C -> S3 -> S4
}

one sig Q3 extends LTS {} {
    init = S0
    trans = A -> S0 -> S1 + A -> S0 -> S3 + B -> S1 -> S2
            + Tau -> S1 -> S3 + C -> S3 -> S4
}

one sig P4 extends LTS {} {
    init = S0
    trans = A -> S0 -> S1 + B -> S1 -> S0 + A -> S0 -> S2
            + B -> S2 -> S0
}

one sig Q4 extends LTS {} {
    init = S0
    trans = Tau -> S0 -> S2 + A -> S2 -> S3 + B -> S3 -> S2
            + Tau -> S3 -> S1 + A -> S1 -> S0
}
```

```
one sig P5 extends LTS {} {
    init = S0
    trans = A -> S0 -> S1 + B -> S1 -> S0
}

one sig Q5 extends LTS {} {
    init = S0
    trans = Tau -> S0 -> S2 + A -> S2 -> S3 + Tau -> S3 -> S1
            + B -> S1 -> S0
}

one sig P6 extends LTS {} {
    init = S0
    trans = B -> S0 -> S1 + A -> S1 -> S2 + C -> S2 -> S2
            + Tau -> S2 -> S0
}

one sig Q6 extends LTS {} {
    init = S0
    trans = B -> S0 -> S1 + A -> S1 -> S0 + A -> S1 -> S2
            + C -> S2 -> S2 + Tau -> S2 -> S0
}

one sig P7 extends MTS {} {
    init = S0
    req = A -> S0 -> S1 + B -> S1 -> S2
    trans = req + B -> S2 -> S1 + C -> S2 -> S3
}

one sig Q7 extends MTS {} {
    init = S0
    req = A -> S0 -> S1 + B -> S1 -> S2  + C -> S2 -> S3
    trans = req + B -> S2 -> S1
}

one sig P8 extends MTS {} {
    init = S0
    req = A -> S0 -> S1 + B -> S1 -> S3
    trans = req + Tau -> S0 -> S2 + B -> S2 -> S4
}

one sig Q8 extends MTS {} {
    init = S0
    req = A -> S0 -> S1 + Tau -> S1 -> S3 + B -> S3 -> S4
    trans = req + B -> S0 -> S2
}

one sig P9 extends MTS {} {
    init = S0
    req = A -> S0 -> S1 + Tau -> S1 -> S2 + B -> S2  -> S0
    trans = req + C -> S1 -> S2
}

one sig Q9 extends MTS {} {
    init = S0
    req = A -> S0 -> S1 + Tau -> S1 -> S2
    trans = req + C -> S1 -> S2  + B -> S2  -> S0
}

one sig P10 extends MTS {} {
    init = S0
    req = A -> S0 -> S1 + B -> S1 -> S2
```

```
    trans = req + B -> S2 -> S1 + C -> S2 -> S0
}

one sig Q10 extends LTS {} {
    init = S0
    req = A -> S0 -> S1 + B -> S1 -> S2
}

one sig P11 extends MTS {} {
    init = S0
    req = A -> S0 -> S1
    trans = req + Tau -> S1 -> S2 + B -> S2 -> S3
}

one sig Q11 extends LTS {} {
    init = S0
    trans = A -> S0 -> S1 + B -> S1 -> S0
}

one sig P12 extends MTS {} {
    init = S0
    req = A -> S0 -> S2 + B -> S2 -> S4 + Tau -> S2 -> S5
    trans = req + Tau -> S0 -> S1 + A -> S1 -> S3
}

one sig Q12 extends LTS {} {
    init = S0
    trans = A -> S0 -> S1 + A -> S0 -> S3 + B -> S1 -> S2
            + Tau -> S1 -> S3
}

one sig P13 extends MTS {} {
    init = S0
    req = A -> S0 -> S2 + B -> S2 -> S4 + C -> S2 -> S5
    trans = req + Tau -> S0 -> S1 + B -> S1 -> S3
}

one sig Q13 extends LTS {} {
    init = S0
    trans = A -> S0 -> S1 + B -> S1 -> S3 + A -> S0 -> S2
            + C -> S2 -> S4
}

one sig P14 extends MTS {} {
    init = S0
    req = A -> S0 -> S1 + B -> S1 -> S3 + C -> S3 -> S0
    trans = req + Tau -> S3 -> S2 + C -> S2 -> S0
}

one sig Q14 extends LTS {} {
    init = S0
    trans = A -> S0 -> S1 + Tau -> S1 -> S3 + B -> S3 -> S2
            + C -> S2 -> S0
}

one sig P15 extends MTS {} {
    init = S0
    req = A -> S0 -> S1 + B -> S1 -> S3 + C -> S3 -> S0
    trans = req + Tau -> S3 -> S2 + C -> S2 -> S0
}

one sig Q15 extends LTS {} {
```

```
    init = S0
    trans = A -> S0 -> S1 + Tau -> S1 -> S3 + B -> S3 -> S2
            + C -> S2 -> S0
}
```

## C.4 KodKod

A continuación se listan los códigos de los ejemplos representados en la Tabla 3.6 en KodKod.

```
// P1 y Q1
pTrans.add(factory.tuple("act1", "S1", "S0"));
pTrans.add(factory.tuple("act0", "S0", "S1"));
pTrans.add(factory.tuple("act0", "S0", "S2"));
pTrans.add(factory.tuple("act1", "S2", "S0"));
qTrans.add(factory.tuple("act0", "S0", "S2"));
qTrans.add(factory.tuple("act1", "S2", "S3"));
qTrans.add(factory.tuple("act0", "S3", "S2"));
qTrans.add(factory.tuple("act0", "S3", "S1"));
qTrans.add(factory.tuple("act1", "S1", "S0"));

// P2 y Q2
pTrans.add(factory.tuple("act0", "S0", "S1"));
pTrans.add(factory.tuple("act1", "S1", "S2"));
pTrans.add(factory.tuple("act1", "S1", "S3"));
pTrans.add(factory.tuple("act2", "S2", "S4"));
pTrans.add(factory.tuple("act3", "S3", "S5"));
qTrans.add(factory.tuple("act0", "S0", "S1"));
qTrans.add(factory.tuple("act0", "S0", "S2"));
qTrans.add(factory.tuple("act1", "S1", "S3"));
qTrans.add(factory.tuple("act1", "S2", "S4"));
qTrans.add(factory.tuple("act2", "S3", "S5"));
qTrans.add(factory.tuple("act3", "S4", "S6"));

// P3 y Q3
pTrans.add(factory.tuple("act0", "S0", "S1"));
pTrans.add(factory.tuple("act1", "S1", "S2"));
pTrans.add(factory.tuple("tau", "S1", "S3"));
pTrans.add(factory.tuple("act2", "S3", "S4"));
qTrans.add(factory.tuple("act0", "S0", "S1"));
qTrans.add(factory.tuple("act0", "S0", "S3"));
qTrans.add(factory.tuple("act1", "S1", "S2"));
qTrans.add(factory.tuple("tau", "S1", "S3"));
qTrans.add(factory.tuple("act2", "S3", "S4"));

// P4 y Q4
pTrans.add(factory.tuple("act0", "S0", "S1"));
pTrans.add(factory.tuple("act1", "S1", "S0"));
pTrans.add(factory.tuple("act0", "S0", "S2"));
pTrans.add(factory.tuple("act1", "S2", "S0"));
qTrans.add(factory.tuple("tau", "S0", "S2"));
qTrans.add(factory.tuple("act0", "S2", "S3"));
qTrans.add(factory.tuple("act1", "S3", "S2"));
qTrans.add(factory.tuple("tau", "S3", "S1"));
qTrans.add(factory.tuple("act0", "S1", "S0"));

// P5 y Q5
pTrans.add(factory.tuple("act0", "S0", "S1"));
pTrans.add(factory.tuple("act1", "S1", "S0"));
qTrans.add(factory.tuple("tau", "S0", "S2"));
```

```
qTrans.add(factory.tuple("act0", "S2", "S3"));
qTrans.add(factory.tuple("tau", "S3", "S1"));
qTrans.add(factory.tuple("act1", "S1", "S0"));

// P6 y Q6
pTrans.add(factory.tuple("act1", "S0", "S1"));
pTrans.add(factory.tuple("act0", "S1", "S2"));
pTrans.add(factory.tuple("act2", "S2", "S2"));
pTrans.add(factory.tuple("tau", "S2", "S0"));
qTrans.add(factory.tuple("act1", "S0", "S1"));
qTrans.add(factory.tuple("act1", "S1", "S0"));
qTrans.add(factory.tuple("act0", "S1", "S2"));
qTrans.add(factory.tuple("act2", "S2", "S2"));
qTrans.add(factory.tuple("tau", "S2", "S0"));

// P7 y Q7
pTrans.add(factory.tuple("act0", "S0", "S1"));
pTrans.add(factory.tuple("act1", "S1", "S2"));
pTrans.add(factory.tuple("act1", "S2", "S1"));
pTrans.add(factory.tuple("act2", "S2", "S0"));
pReq.add(factory.tuple("act0", "S0", "S1"));
pReq.add(factory.tuple("act1", "S1", "S2"));
qTrans.add(factory.tuple("act0", "S0", "S1"));
qTrans.add(factory.tuple("act1", "S1", "S2"));
qTrans.add(factory.tuple("act1", "S2", "S1"));
qTrans.add(factory.tuple("act2", "S2", "S0"));
qReq.add(factory.tuple("act0", "S0", "S1"));
qReq.add(factory.tuple("act1", "S1", "S2"));
qReq.add(factory.tuple("act2", "S2", "S0"));

// P8 y Q8
pTrans.add(factory.tuple("act0", "S0", "S1"));
pTrans.add(factory.tuple("tau", "S0", "S2"));
pTrans.add(factory.tuple("act1", "S1", "S3"));
pTrans.add(factory.tuple("act1", "S2", "S4"));
pReq.add(factory.tuple("act0", "S0", "S1"));
pReq.add(factory.tuple("act1", "S1", "S3"));
qTrans.add(factory.tuple("act0", "S0", "S1"));
qTrans.add(factory.tuple("act1", "S0", "S2"));
qTrans.add(factory.tuple("tau", "S1", "S3"));
qTrans.add(factory.tuple("act1", "S3", "S4"));
qReq.add(factory.tuple("act0", "S0", "S1"));
qReq.add(factory.tuple("tau", "S1", "S3"));
qReq.add(factory.tuple("act1", "S3", "S4"));

// P9 y Q9
pTrans.add(factory.tuple("act0", "S0", "S1"));
pTrans.add(factory.tuple("tau", "S1", "S2"));
pTrans.add(factory.tuple("act2", "S2", "S2"));
pTrans.add(factory.tuple("act1", "S2", "S0"));
pReq.add(factory.tuple("act0", "S0", "S1"));
pReq.add(factory.tuple("act1", "S2", "S0"));
qTrans.add(factory.tuple("act0", "S0", "S1"));
qTrans.add(factory.tuple("tau", "S1", "S2"));
qTrans.add(factory.tuple("act2", "S2", "S2"));
qTrans.add(factory.tuple("act1", "S2", "S0"));
qReq.add(factory.tuple("act0", "S0", "S1"));
qReq.add(factory.tuple("tau", "S1", "S2"));

// P10 y Q10
pTrans.add(factory.tuple("act0", "S0", "S1"));
pTrans.add(factory.tuple("act1", "S1", "S2"));
```

```
pTrans.add(factory.tuple("act1", "S2", "S1"));
pTrans.add(factory.tuple("act2", "S2", "S0"));
pReq.add(factory.tuple("act0", "S0", "S1"));
pReq.add(factory.tuple("act1", "S1", "S2"));
qTrans.add(factory.tuple("act0", "S0", "S1"));
qTrans.add(factory.tuple("act1", "S1", "S2"));

// P11 y Q11
pTrans.add(factory.tuple("act0", "S0", "S1"));
pTrans.add(factory.tuple("tau", "S1", "S2"));
pTrans.add(factory.tuple("act1", "S2", "S0"));
pReq.add(factory.tuple("act0", "S0", "S1"));
qTrans.add(factory.tuple("act0", "S0", "S1"));
qTrans.add(factory.tuple("act1", "S1", "S0"));

// P12 y Q12
pTrans.add(factory.tuple("tau", "S0", "S1"));
pTrans.add(factory.tuple("act0", "S0", "S2"));
pTrans.add(factory.tuple("act0", "S1", "S3"));
pTrans.add(factory.tuple("act1", "S2", "S4"));
pTrans.add(factory.tuple("tau", "S2", "S5"));
pReq.add(factory.tuple("act0", "S0", "S2"));
pReq.add(factory.tuple("act1", "S2", "S4"));
pReq.add(factory.tuple("tau", "S2", "S5"));
qTrans.add(factory.tuple("act0", "S0", "S1"));
qTrans.add(factory.tuple("act0", "S0", "S3"));
qTrans.add(factory.tuple("act1", "S1", "S2"));
qTrans.add(factory.tuple("tau", "S1", "S3"));

// P13 y Q13
pTrans.add(factory.tuple("tau", "S0", "S1"));
pTrans.add(factory.tuple("act0", "S0", "S2"));
pTrans.add(factory.tuple("act0", "S1", "S3"));
pTrans.add(factory.tuple("act1", "S2", "S4"));
pTrans.add(factory.tuple("act2", "S2", "S5"));
pReq.add(factory.tuple("act0", "S1", "S3"));
pReq.add(factory.tuple("act1", "S2", "S4"));
pReq.add(factory.tuple("act2", "S2", "S5"));
qTrans.add(factory.tuple("act0", "S0", "S1"));
qTrans.add(factory.tuple("act0", "S1", "S3"));
qTrans.add(factory.tuple("act1", "S2", "S4"));
qTrans.add(factory.tuple("act2", "S2", "S5"));

// P14 y Q14
pTrans.add(factory.tuple("act0", "S0", "S1"));
pTrans.add(factory.tuple("act1", "S1", "S3"));
pTrans.add(factory.tuple("act2", "S3", "S0"));
pTrans.add(factory.tuple("tau", "S3", "S2"));
pTrans.add(factory.tuple("act2", "S2", "S0"));
pReq.add(factory.tuple("act0", "S0", "S1"));
pReq.add(factory.tuple("act1", "S1", "S3"));
pReq.add(factory.tuple("act2", "S3", "S0"));
qTrans.add(factory.tuple("act0", "S0", "S1"));
qTrans.add(factory.tuple("tau", "S1", "S3"));
qTrans.add(factory.tuple("act1", "S3", "S2"));
qTrans.add(factory.tuple("act2", "S2", "S0"));


// P1 y Q1
final Formula show = model.p.BB(model.rel.rel, model.q);
final Solution sol = solver.solve(show, model.boundsSystem(4, 2));
// P2 y Q2
```

```
final Formula show = model.p.SB(model.rel.rel , model.q);
final Solution sol = solver.solve(show, model.boundsSystem(7, 4));
// P3 y Q3
final Formula show = model.p.WB(model.rel.rel , model.q);
final Solution sol = solver.solve(show, model.boundsSystem(5, 3));
// P4 y Q4
final Formula show = model.p.WB(model.rel.rel , model.q);
final Solution sol = solver.solve(show, model.boundsSystem(4, 2));
// P5 y Q5
final Formula show = model.p.BB(model.rel.rel , model.q);
final Solution sol = solver.solve(show, model.boundsSystem(4, 2));
// P6 y Q6
final Formula show = model.p.BB(model.rel.rel , model.q);
final Solution sol = solver.solve(show, model.boundsSystem(3, 3));
// Q7 y P7
final Formula show = model.q.SR(model.rel.rel , model.p);
final Solution sol = solver.solve(show, model.boundsSystemM(4, 4));
// P8 y Q8
final Formula show = model.p.WR(model.rel.rel , model.q);
final Solution sol = solver.solve(show, model.boundsSystemM(5, 2));
// P9 y Q9
final Formula show = model.p.WR(model.rel.rel , model.q);
final Solution sol = solver.solve(show, model.boundsSystemM(3, 3));
// P10 y Q10
final Formula show = model.p.WI(model.rel.rel , model.q);
final Solution sol = solver.solve(show, model.boundsSystemM(3, 3));
// P11 y Q11
final Formula show = model.p.SI(model.rel.rel , model.q);
final Solution sol = solver.solve(show, model.boundsSystemM(3, 2));
// P12 y Q12
final Formula show = model.p.WI(model.rel.rel , model.q);
final Solution sol = solver.solve(show, model.boundsSystemM(6, 2));
// P13 y Q13
final Formula show = model.p.WI(model.rel.rel , model.q);
final Solution sol = solver.solve(show, model.boundsSystemM(6, 3));
// P14 y Q14
final Formula show = model.p.WI(model.rel.rel , model.q);
final Solution sol = solver.solve(show, model.boundsSystemM(4, 3));
```

# D Mínimo conjunto de Relaciones en Alloy

El siguiente es el conjunto mínimo de relaciones obtenido a partir de en primer instancia un conjunto de 256 relaciones, las cuales las redujimos a 84 mediante el análisis de Alloy Analyzer. A partir de éstas es posible generar todo el universo de las relaciones.

```
one sig R0 extends Relation {}{
    rel = S0 -> S0
}

one sig R1 extends Relation {}{
    rel = S0 -> S0 + S0 -> S1
}

one sig R3 extends Relation {}{
    rel = S0 -> S0 + S1 -> S0
}

one sig R4 extends Relation {}{
    rel = S0 -> S0 + S1 -> S1
}

one sig R9 extends Relation {}{
    rel = S0 -> S0 + S0 -> S1 + S0 -> S2
}

one sig R10 extends Relation {}{
    rel = S0 -> S0 + S0 -> S1 + S1 -> S0
}

one sig R12 extends Relation {}{
    rel = S0 -> S0 + S0 -> S1 + S1 -> S1
}

one sig R13 extends Relation {}{
    rel = S0 -> S0 + S0 -> S2 + S1 -> S1
}

one sig R14 extends Relation {}{
    rel = S0 -> S0 + S1 -> S0 + S1 -> S1
}

one sig R18 extends Relation {}{
    rel = S0 -> S0 + S1 -> S1 + S1 -> S2
```

```
}

one sig R21 extends Relation {}{
    rel = S0 -> S0 + S1 -> S0 + S2 -> S0
}

one sig R22 extends Relation {}{
    rel = S0 -> S0 + S1 -> S1 + S2 -> S0
}

one sig R27 extends Relation {}{
    rel = S0 -> S0 + S1 -> S1 + S2 -> S1
}

one sig R28 extends Relation {}{
    rel = S0 -> S0 + S1 -> S2 + S2 -> S1
}

one sig R37 extends Relation {}{
    rel = S0 -> S0 + S0 -> S1 + S0 -> S2 + S1 -> S0
}

one sig R38 extends Relation {}{
    rel = S0 -> S0 + S0 -> S1 + S0 -> S2 + S1 -> S1
}

one sig R39 extends Relation {}{
    rel = S0 -> S0 + S0 -> S1 + S1 -> S0 + S1 -> S1
}

one sig R40 extends Relation {}{
    rel = S0 -> S0 + S0 -> S2 + S1 -> S0 + S1 -> S1
}

one sig R44 extends Relation {}{
    rel = S0 -> S0 + S0 -> S1 + S1 -> S1 + S1 -> S2
}

one sig R46 extends Relation {}{
    rel = S0 -> S0 + S1 -> S0 + S1 -> S1 + S1 -> S2
}

one sig R48 extends Relation {}{
    rel = S0 -> S0 + S0 -> S1 + S1 -> S0 + S2 -> S0
}

one sig R50 extends Relation {}{
    rel = S0 -> S0 + S0 -> S1 + S1 -> S1 + S2 -> S0
}

one sig R51 extends Relation {}{
    rel = S0 -> S0 + S0 -> S2 + S1 -> S1 + S2 -> S0
}

one sig R52 extends Relation {}{
    rel = S0 -> S0 + S1 -> S0 + S1 -> S1 + S2 -> S0
}

one sig R56 extends Relation {}{
    rel = S0 -> S0 + S1 -> S1 + S1 -> S2 + S2 -> S0
}
```

```
one sig R60 extends Relation {}{
    rel = S0 -> S0 + S0 -> S1 + S1 -> S1 + S2 -> S1
}

one sig R61 extends Relation {}{
    rel = S0 -> S0 + S0 -> S2 + S1 -> S1 + S2 -> S1
}

one sig R62 extends Relation {}{
    rel = S0 -> S0 + S1 -> S0 + S1 -> S1 + S2 -> S1
}

one sig R63 extends Relation {}{
    rel = S0 -> S0 + S0 -> S1 + S1 -> S2 + S2 -> S1
}

one sig R65 extends Relation {}{
    rel = S0 -> S0 + S1 -> S0 + S1 -> S2 + S2 -> S1
}

one sig R66 extends Relation {}{
    rel = S0 -> S0 + S1 -> S1 + S1 -> S2 + S2 -> S1
}

one sig R93 extends Relation {}{
    rel = S0 -> S0 + S0 -> S1 + S0 -> S2 + S1 -> S0 + S1 -> S1
}

one sig R95 extends Relation {}{
    rel = S0 -> S0 + S0 -> S1 + S0 -> S2 + S1 -> S1 + S1 -> S2
}

one sig R96 extends Relation {}{
    rel = S0 -> S0 + S0 -> S1 + S1 -> S0 + S1 -> S1 + S1 -> S2
}

one sig R98 extends Relation {}{
    rel = S0 -> S0 + S0 -> S1 + S0 -> S2 + S1 -> S0 + S2 -> S0
}

one sig R99 extends Relation {}{
    rel = S0 -> S0 + S0 -> S1 + S0 -> S2 + S1 -> S1 + S2 -> S0
}

one sig R100 extends Relation {}{
    rel = S0 -> S0 + S0 -> S1 + S1 -> S0 + S1 -> S1 + S2 -> S0
}

one sig R101 extends Relation {}{
    rel = S0 -> S0 + S0 -> S2 + S1 -> S0 + S1 -> S1 + S2 -> S0
}

one sig R105 extends Relation {}{
    rel = S0 -> S0 + S0 -> S1 + S1 -> S1 + S1 -> S2 + S2 -> S0
}

one sig R107 extends Relation {}{
    rel = S0 -> S0 + S1 -> S0 + S1 -> S1 + S1 -> S2 + S2 -> S0
}

one sig R109 extends Relation {}{
    rel = S0 -> S0 + S0 -> S1 + S0 -> S2 + S1 -> S1 + S2 -> S1
```

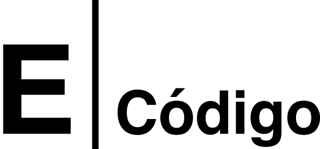
```
}

one sig R110 extends Relation {}{
    rel = S0 -> S0 + S0 -> S1 + S1 -> S0 + S1 -> S1 + S2 -> S1
}

one sig R111 extends Relation {}{
    rel = S0 -> S0 + S0 -> S2 + S1 -> S0 + S1 -> S1 + S2 -> S1
}

one sig R112 extends Relation {}{
    rel = S0 -> S0 + S0 -> S1 + S0 -> S2 + S1 -> S2 + S2 -> S1
}

one sig R113 extends Relation {}{
    rel = S0 -> S0 + S0 -> S1 + S1 -> S0 + S1 -> S2 + S2 -> S1
}

one sig R114 extends Relation {}{
    rel = S0 -> S0 + S0 -> S2 + S1 -> S0 + S1 -> S2 + S2 -> S1
}

one sig R115 extends Relation {}{
    rel = S0 -> S0 + S0 -> S1 + S1 -> S1 + S1 -> S2 + S2 -> S1
}

one sig R116 extends Relation {}{
    rel = S0 -> S0 + S0 -> S2 + S1 -> S1 + S1 -> S2 + S2 -> S1
}

one sig R117 extends Relation {}{
    rel = S0 -> S0 + S1 -> S0 + S1 -> S1 + S1 -> S2 + S2 -> S1
}

one sig R123 extends Relation {}{
    rel = S0 -> S0 + S1 -> S0 + S1 -> S1 + S2 -> S0 + S2 -> S1
}

one sig R126 extends Relation {}{
    rel = S0 -> S0 + S1 -> S0 + S1 -> S2 + S2 -> S0 + S2 -> S1
}

one sig R127 extends Relation {}{
    rel = S0 -> S0 + S1 -> S1 + S1 -> S2 + S2 -> S0 + S2 -> S1
}

one sig R157 extends Relation {}{
    rel = S0 -> S0 + S1 -> S1 + S1 -> S2 + S2 -> S1 + S2 -> S2
}

one sig R163 extends Relation {}{
    rel = S0 -> S0 + S0 -> S1 + S0 -> S2 + S1 -> S0 + S1 -> S1
          + S1 -> S2
}

one sig R164 extends Relation {}{
    rel = S0 -> S0 + S0 -> S1 + S0 -> S2 + S1 -> S0 + S1 -> S1
          + S2 -> S0
}

one sig R166 extends Relation {}{
    rel = S0 -> S0 + S0 -> S1 + S0 -> S2 + S1 -> S1 + S1 -> S2
```

```
          + S2 -> S0
}

one sig R167 extends Relation {}{
    rel = S0 -> S0 + S0 -> S1 + S1 -> S0 + S1 -> S1 + S1 -> S2
          + S2 -> S0
}

one sig R169 extends Relation {}{
    rel = S0 -> S0 + S0 -> S1 + S0 -> S2 + S1 -> S0 + S1 -> S1
          + S2 -> S1
}

one sig R170 extends Relation {}{
    rel = S0 -> S0 + S0 -> S1 + S0 -> S2 + S1 -> S0 + S1 -> S2
          + S2 -> S1
}

one sig R171 extends Relation {}{
    rel = S0 -> S0 + S0 -> S1 + S0 -> S2 + S1 -> S1 + S1 -> S2
          + S2 -> S1
}

one sig R172 extends Relation {}{
    rel = S0 -> S0 + S0 -> S1 + S1 -> S0 + S1 -> S1 + S1 -> S2
          + S2 -> S1
}

one sig R173 extends Relation {}{
    rel = S0 -> S0 + S0 -> S2 + S1 -> S0 + S1 -> S1 + S1 -> S2
          + S2 -> S1
}

one sig R176 extends Relation {}{
    rel = S0 -> S0 + S0 -> S1 + S1 -> S0 + S1 -> S1 + S2 -> S0
          + S2 -> S1
}

one sig R177 extends Relation {}{
    rel = S0 -> S0 + S0 -> S2 + S1 -> S0 + S1 -> S1 + S2 -> S0
          + S2 -> S1
}

one sig R179 extends Relation {}{
    rel = S0 -> S0 + S0 -> S1 + S1 -> S0 + S1 -> S2 + S2 -> S0
          + S2 -> S1
}

one sig R181 extends Relation {}{
    rel = S0 -> S0 + S0 -> S1 + S1 -> S1 + S1 -> S2 + S2 -> S0
          + S2 -> S1
}

one sig R182 extends Relation {}{
    rel = S0 -> S0 + S0 -> S2 + S1 -> S1 + S1 -> S2 + S2 -> S0
          + S2 -> S1
}

one sig R183 extends Relation {}{
    rel = S0 -> S0 + S1 -> S0 + S1 -> S1 + S1 -> S2 + S2 -> S0
          + S2 -> S1
}
```

```
one sig R206 extends Relation {}{
    rel = S0 -> S0 + S0 -> S1 + S1 -> S1 + S1 -> S2 + S2 -> S1
         + S2 -> S2
}

one sig R208 extends Relation {}{
    rel = S0 -> S0 + S1 -> S0 + S1 -> S1 + S1 -> S2 + S2 -> S1
         + S2 -> S2
}

one sig R219 extends Relation {}{
    rel = S0 -> S0 + S0 -> S1 + S0 -> S2 + S1 -> S0 + S1 -> S1
         + S1 -> S2 + S2 -> S0
}

one sig R220 extends Relation {}{
    rel = S0 -> S0 + S0 -> S1 + S0 -> S2 + S1 -> S0 + S1 -> S1
         + S1 -> S2 + S2 -> S1
}

one sig R221 extends Relation {}{
    rel = S0 -> S0 + S0 -> S1 + S0 -> S2 + S1 -> S0 + S1 -> S1
         + S2 -> S0 + S2 -> S1
}

one sig R222 extends Relation {}{
    rel = S0 -> S0 + S0 -> S1 + S0 -> S2 + S1 -> S0 + S1 -> S2
         + S2 -> S0 + S2 -> S1
}

one sig R223 extends Relation {}{
    rel = S0 -> S0 + S0 -> S1 + S0 -> S2 + S1 -> S1 + S1 -> S2
         + S2 -> S0 + S2 -> S1
}

one sig R224 extends Relation {}{
    rel = S0 -> S0 + S0 -> S1 + S1 -> S0 + S1 -> S1 + S1 -> S2
         + S2 -> S0 + S2 -> S1
}

one sig R225 extends Relation {}{
    rel = S0 -> S0 + S0 -> S2 + S1 -> S0 + S1 -> S1 + S1 -> S2
         + S2 -> S0 + S2 -> S1
}

one sig R234 extends Relation {}{
    rel = S0 -> S0 + S0 -> S1 + S0 -> S2 + S1 -> S1 + S1 -> S2
          + S2 -> S1 + S2 -> S2
}

one sig R235 extends Relation {}{
    rel = S0 -> S0 + S0 -> S1 + S1 -> S0 + S1 -> S1 + S1 -> S2
          + S2 -> S1 + S2 -> S2
}

one sig R246 extends Relation {}{
    rel = S0 -> S0 + S1 -> S0 + S1 -> S1 + S1 -> S2 + S2 -> S0
          + S2 -> S1 + S2 -> S2
}

one sig R247 extends Relation {}{
```

```
    rel = S0 -> S0 + S0 -> S1 + S0 -> S2 + S1 -> S0 + S1 -> S1
          + S1 -> S2 + S2 -> S0 + S2 -> S1
}

one sig R249 extends Relation {}{
    rel = S0 -> S0 + S0 -> S1 + S0 -> S2 + S1 -> S0 + S1 -> S1
          + S1 -> S2 + S2 -> S1 + S2 -> S2
}

one sig R253 extends Relation {}{
    rel = S0 -> S0 + S0 -> S1 + S1 -> S0 + S1 -> S1 + S1 -> S2
          + S2 -> S0 + S2 -> S1 + S2 -> S2
}

one sig R255 extends Relation {}{
    rel = S0 -> S0 + S0 -> S1 + S0 -> S2 + S1 -> S0 + S1 -> S1
          + S1 -> S2 + S2 -> S0 + S2 -> S1 + S2 -> S2
}
```

2



# E Código

## E.1 Disco Ram

Como vimos el el último capítulo, creamos una partición ram para optimizar nuestros tiempos de lectura y escritura. Este se puede realizar en linux de la siguiente manera en S.O. Linux:

En el grub modificar lo siguiente:

```
title Red Hat Linux (2.4.20-20.9)
        root (hd0,0)
        kernel /vmlinuz-2.4.20-20.9 ro
                root=LABEL=/ hdc=ide-scsi \textbf{ramdisk_size=16000}
        initrd /initrd-2.4.20-20.9.img
```

Luego, reiciamos y ejecutamos este script:

```
# Formats, mounts, and sets permissions on my 16MB ramdisk
/sbin/mke2fs -q -m 0 /dev/ram0
/bin/mount /dev/ram0 /mnt/rd
/bin/chown van:root /mnt/rd
/bin/chmod 0750 /mnt/rd
```

## E.2 Script de generación de MTS y análisis de nuestra propiedad

```
#!/usr/bin/perl
use Data::PowerSet 'powerset';
use Algorithm::Permute;
use Math::BigRat;
use Text::ParseWords;
use warnings;

# Datos para completar
my $StateMax = 3;
my $ActionMax = 1;
my @actions = ("A");
my $stateInit="S0";

my @ternas = (0..$StateMax*$StateMax*$ActionMax-1);
my @relationsReq=(0..19682);
my @relationsTrans=(0..19682);

my $i=0;
my $j=0;
my $k=0;
my $resul=0;
my $count=0;

# Genero los estados y los guardo en @states
while ($i < $StateMax){
 $states[$i] = "S".$i;
 $i = $i + 1;
}

$i = 0;

# Genero pares de estados con acciones y los guardo en @ternas
while ($i < $StateMax){
```

```
 $j = 0;
 while ($j < $StateMax){
  $k = 0;
  while ($k < $ActionMax){
   $ternas[$count] = $actions[$k]."->".$states[$i]."->".$states[$j];
   $k = $k +1;
   $count=$count+1;
  }
  $j = $j +1;
 }
 $i = $i + 1;
}

# Genero el universo de relaciones. Quedan guardadas en un array de arrays. Sin formato adecuado.
@ternasE=@ternas;
my $powerset = powerset(@ternasE);

# Guardo en dos arrays las relaciones generadas, las cuales est n en un array de arrays.
# Descarto los mts que no sean conexos o que desde el estado s0 no salga ninguna arista
$numRelation = 0;
my $reqA = "";
my $transA = "";
$i = 0;
for my $p (@$powerset) {
        $transA = "@$p";
        $transA =~ s/ / + /g;
 if ($transA =~m/->S0->/ && is_conexo($transA)){
         my $powersetAc = powerset($p);
         for my $m (@$powersetAc) {
                 $reqA = "@$m";
                 $reqA =~ s/ / + /g;
                 $relationsReq[$numRelation] = $reqA;
#    print $relationsReq[$numRelation]."\n";
                 $relationsTrans[$numRelation] = $transA;
                 $numRelation = $numRelation + 1;
         }
```

```
  }
}
print "Cantidad de MTS ".$numRelation."\n";
@timeData = localtime(time);
print join(' ', @timeData);
print "\n";


# fijo dos MTS, y chequeo las prop que deseo, mediante las funciones checkImp, y checkRef.
$k=14627;
$i=14611;
open(EJBKP, ">ejemplosbkp");
$count = 0;
while (0 <= $i){
 $mts1 = gen_mts($i);
 $mts2 = gen_mts($k);
 print EJBKP "Ejemplo de MTS i = ".$i." k = ".$k."\n";
 if (checkRef($mts1, $mts2)){
  if (checkImp($mts1, $mts2)){
   print EJBKP $mts1;
   print EJBKP $mts2;
   print EJBKP "Fin de Ejemplo\n";
  } else {
   print EJBKP "Imp false\n";
   }
 } else {
  print EJBKP "No ref\n";
 }
 $k = $k-1;
 if ($k == -1){
  $k = $numRelation-1;
  $i = $i -1;
 }
}
close EJBKP;
```

```
# genero el mts
sub gen_mts {
 $a = shift;
 my $req = "";
 my $trans = "";
 $req = $relationsReq[$a];
 $trans = $relationsTrans[$a];
 if ($req =~ m/[A-Z->0-9]/){
  $mts = "    req = ".$req."\n    trans = ".$trans."\n";
 } else {
  $mts = "    no req\n    trans = ".$trans."\n";
 }
 $mts = "one sig mts extends MTS {}{\n    init = S0\n".$mts."\}\n\n";
 return ($mts);
}

# chequeo la prop que no son refinamientos
sub checkRef {
 my $m1 = shift;
 my $m2 = shift;
 $m1 =~ s/mts/mts1/g;
 $m2 =~ s/mts/mts2/g;
  open (FILE, ">InstanciesMts.als");
 print FILE "module InstanciesMts\n";
 print FILE "open RelationMinimal3State\n\n";
 print FILE $m1."\n";
 print FILE $m2."\n\n";
 close FILE;

 my $r = `time java -Xmx10000m -cp alloy4.jar edu.mit.csail.sdg.alloy4whole.ExampleUsingTheCompiler alloyInstanciaA.als`;

 if ($r =~ /Unsatisfiable/){
  return (1);
 }else{
  return (0);
 }
```

```
}

# chequeo propiedad en la cual existe una imp tal que...
sub checkImp {
 my $r = `java -Xmx10000m -cp alloy4.jar edu.mit.csail.sdg.alloy4whole.ExampleUsingTheCompiler alloyInstanciaB.als`;
 if ($r =~ /Unsatisfiable/){
  return (1);
 } else {
  return (0);
 }
}

# chequeo si el grafo es conexo.
sub is_conexo {
 my $t = shift;
 open (FILE, ">InstancieConexo.als");
 print FILE "module InstancieConexo\n";
 print FILE "open libraryAux\n\n";
 print FILE "one sig T extends Transition {} {\n";
 print FILE "    trans = ".$t."\n}\n\n";
 close FILE;
 my $r = `java -Xmx1500m -cp alloy4.jar edu.mit.csail.sdg.alloy4whole.ExampleUsingTheCompiler IsConexo.als`;
 if ($r =~ /Unsatisfiable/){
  return (0);
 } else {
  return (1);
 }

}
```

### E.2.1 Es conexo

```
module Is_Conexo
open InstancieConexo

pred is_con (T: Transition) {
```

```
    is_conexo[T]
}

run is_con for 1
```

### E.2.2 Antecendente de la propiedad

```
module AlloyInstanceA
open library
open RelationMinimal3State
open InstanciesMts

pred a {
  some f1, f2: Bijection, r: Relation |
    mts1.init -> mts2.init in (f1.rel).(r.rel).(f2.rel) and
    StrongRefinement[(f1.rel).(r.rel).(f2.rel), mts1.trans, mts1.req, mts2.trans, mts2.req]
}
run a for 5
```

### E.2.3 Consecuente de la propiedad

```
module AlloyInstanceB
open RelationMinimal3State
open InstanciesMts

one sig lts1 extends LTS{}{
 init = S0
}

pred a (lts1: LTS) {
  (some f1, f2: Bijection, r: Relation |
    mts2.init -> lts1.init in (f1.rel).(r.rel).(f2.rel) and
    StrongImplementation[(f1.rel).(r.rel).(f2.rel), mts2.trans, mts2.req, lts1.trans]) and
          !(some f1, f2: Bijection, r: Relation |
    mts1.init -> lts1.init in (f1.rel).(r.rel).(f2.rel) and
```

```
    StrongImplementation[(f1.rel).(r.rel).(f2.rel), mts1.trans, mts1.req, lts1.trans])
}
run a for 1
```

## E.3 KodKod

### E.3.1 Clase Transition

```
import kodkod.ast.Expression;
import kodkod.ast.Relation;
import kodkod.ast.Formula;
import kodkod.ast.Variable;

/**
 * @author Carolina Dania & Nazareno Aguirre
 *
 */

public class Transition {

 public final Relation trans;
 public final Relation state, action, tau;

 public Transition() {
  state = Relation.unary("State");
  action = Relation.unary("Action");
  tau = Relation.unary("Tau");
  trans = Relation.nary("Trans", 3);
 }

 public Formula arrow(Variable s1, Variable s2, Variable a) {
  final Formula arrow = (a.product(s1.product(s2))).in(trans);
  return arrow;
 }
```

```
public Formula arrowT(Variable s1, Variable s2, Variable a) {
 final Formula arrowT = (arrow(s1, s2, a)).or((a.eq(tau)).and(s1.eq(s2)));
 return arrowT;
}

public Formula clTArrowClT(Variable s1, Variable s2, Variable a) {
 final Expression rel = s1.product(s2);
 final Expression clTArrow = (tau.join(trans)).reflexiveClosure();
 final Formula clTArrowClt = rel.in((clTArrow.join(a.join(trans))).join(clTArrow));
 return clTArrowClt;
}

public Formula clTArrowTClT(Variable s1, Variable s2, Variable a) {
 final Formula clTArrowTClT = clTArrowClT(s1,s2,a).or((a.eq(tau)).and(s1.eq(s2)));
 return clTArrowTClT;
}

public Expression augTauTransition() {
 final Expression RelState = state.product(state);
 final Expression augTauTrans = this.trans.union(tau.product((Relation.IDEN).intersection(RelState)));
 return augTauTrans;
}

public Expression preImageClTau(Expression c) {
 final Expression preImageClTau = ((tau.join(trans)).intersection(state.product(c))).reflexiveClosure();
 return preImageClTau;
}

public Formula altStrongSimulation(Expression r, Transition other) {
 final Variable a = Variable.unary("a");
       final Variable s1 = Variable.unary("s1");
    final Variable s2 = Variable.unary("s2");
    final Variable s3 = Variable.unary("s3");
    final Variable s4 = Variable.unary("s4");
    final Formula strongSimConsequentBody = (s2.product(s4)).in(a.join(other.trans)).and((s3.product(s4)).in(r));
```

```
    final Formula strongSimConsequent = strongSimConsequentBody.forSome(s4.oneOf(state));
    final Formula strongSimBody = ((s1.product(s2)).in(r)).and((s1.product(s3)).in(a.join(this.trans)));
    final Formula strongSimAll = strongSimBody.implies(strongSimConsequent);
    final Formula strongSimA = strongSimAll.forAll(a.oneOf(action));
    final Formula strongSimS = strongSimA.forAll(s3.oneOf(state)).forAll(s2.oneOf(state)).forAll(s1.oneOf(state));
    return strongSimS;
   }

public Formula altStrongBisimulation(Expression r, Transition other) {
 final Formula altStrongBisI = this.altStrongSimulation(r, other);
 final Formula altStrongBisD = other.altStrongSimulation(r.transpose(), this);
 final Formula altStrongBis = altStrongBisI.and(altStrongBisD);
 return altStrongBis;
}

public Formula strongSimulation(Expression r, Transition other) {
 final Variable a = Variable.unary("a");
 final Formula strongSimBody = ((r.transpose()).join(a.join(this.trans))).in(a.join(other.trans).join(r.transpose()));
 final Formula strongSim = strongSimBody.forAll(a.oneOf(action));
 return strongSim;
}

public Formula strongBisimulation(Expression r, Transition other) {
 final Formula strongBis = this.strongSimulation(r, other).and(other.strongSimulation(r.transpose(), this));
 return strongBis;
}

public Formula branchingSimulation(Expression r, Transition other) {
 final Variable a = Variable.unary("a");
 final Variable s1 = Variable.unary("s1");
 final Variable s2 = Variable.unary("s2");
 final Variable s3 = Variable.unary("s3");
 final Variable s4 = Variable.unary("s4");
 final Variable s5 = Variable.unary("s5");
 final Formula branchSimBodyI = (this.arrow(s1, s2, a)).and((s1.product(s3)).in(r));
 final Formula branchSimBody1 = (s3.product(s4)).in(other.preImageClTau(s1.join(r)));
```

```
 final Formula branchSimBody2 = other.arrowT(s4, s5, a);
 final Formula branchSimBody3 = ((s1.product(s4)).union(s2.product(s5))).in(r);
 final Formula branchSimBodyP = (branchSimBody1.and(branchSimBody2)).and(branchSimBody3);
 final Formula branchSimBody = (branchSimBodyP.forSome(s5.oneOf(state))).forSome(s4.oneOf(state));
 final Formula branchSim = (branchSimBodyI.implies(branchSimBody)).forAll(a.oneOf(action));
 final Formula branchingSim = ((branchSim.forAll(s3.oneOf(state))).forAll(s2.oneOf(state))).forAll(s1.oneOf(state));
 return branchingSim;
}

public Formula branchingBisimulation(Expression r, Transition other) {
 final Formula branchBis = this.branchingSimulation(r, other).and(other.branchingSimulation(r.transpose(), this));
 return branchBis;
}

public Formula weakSimulation(Expression r, Transition other) {
 final Variable a = Variable.unary("a");
 final Expression setPart = (tau.join(other.trans)).reflexiveClosure();
 final Expression set = ((setPart.join(a.join(other.augTauTransition()))).join(setPart)).join(r.transpose());
 final Formula weakSimBody = ((r.transpose()).join(a.join(this.trans))).in(set);
 final Formula weakSim = weakSimBody.forAll(a.oneOf(action));
 return weakSim;
}

public Formula weakBisimulation(Expression r, Transition other){
 final Formula WeakBis = this.weakSimulation(r, other).and(other.weakSimulation(r.transpose(), this));
 return WeakBis;
}

public Formula strongRefinement(Expression r, Transition thisr, Transition other, Transition otherr) {
 final Formula strongRef = thisr.strongSimulation(r, otherr).and(other.strongSimulation(r.transpose(), this));
 return strongRef;
}

public Formula weakRefinement(Expression r, Transition thisr, Transition other, Transition otherr) {
 final Formula weakRef = thisr.weakSimulation(r, otherr).and(other.weakSimulation(r.transpose(), this));
 return weakRef;
```

```java
}

public Formula strongImplementation(Expression r, Transition thisr, Transition other) {
 final Formula strongImp = this.strongRefinement(r, thisr, other, other);
 return strongImp;
}

public Formula weakImplementation(Expression r, Transition thisr, Transition other) {
 final Formula weakImp = this.weakRefinement(r, thisr, other, other);
 return weakImp;
}

public Formula branchingImplementation(Expression r, Transition thisr, Transition other) {
 final Variable a = Variable.unary("a");
 final Variable s1 = Variable.unary("s1");
 final Variable s2 = Variable.unary("s2");
 final Variable s3 = Variable.unary("s3");
 final Variable s4 = Variable.unary("s4");
 final Variable s5 = Variable.unary("s5");
 final Formula bodyR1 = (thisr.arrow(s1, s2, a)).and((s1.product(s3)).in(r));
 final Formula bodyR2 = ((s3.product(s4)).in(other.preImageClTau(s1.join(r)))).and((s2.product(s5)).in(r));
 final Formula bodyR3 = ((bodyR2.and(other.arrowT(s4, s5, a))).forSome(s5.oneOf(state))).forSome(s4.oneOf(state));
 final Formula bodyR = ((bodyR1.implies(bodyR3)).forAll(s3.oneOf(state))).forAll(a.oneOf(action));
 final Formula branchImpR = (bodyR.forAll(s2.oneOf(state))).forAll(s1.oneOf(state));
 final Formula bodyT1 = (other.arrow(s1, s2, a)).and((s1.product(s3)).in(r.transpose()));
 final Formula bodyT2a = ((s3.product(s4)).in(this.preImageClTau(s1.join(r.transpose()))));
 final Formula bodyT2b = bodyT2a.and((s2.product(s5)).in(r.transpose()));
 final Formula bodyT3 = ((bodyT2b.and(this.arrowT(s4, s5, a))).forSome(s5.oneOf(state))).forSome(s4.oneOf(state));
 final Formula bodyT = ((bodyT1.implies(bodyT3)).forAll(s3.oneOf(state))).forAll(a.oneOf(action));
 final Formula branchImpT = (bodyT.forAll(s2.oneOf(state))).forAll(s1.oneOf(state));
 final Formula branchingImp = branchImpR.and(branchImpT);
 return branchingImp;
}

}
```

### E.3.2 Clase LTS

```
import kodkod.ast.Relation;
import kodkod.ast.Formula;
import kodkod.ast.Expression;

/**
 * @author Carolina Dania & Nazareno Aguirre
 *
 */

public class Lts {

 public final Relation init;
 public final Transition trans;

 public Lts() {
  init = Relation.nary("Init", 1);
  trans = new Transition();
 }

 public Formula SB(Expression r, Lts other) {
  final Formula Prop = this.trans.strongBisimulation(r, other.trans);
  final Formula Pre = (this.init.product(other.init)).in(r);
  final Formula SB = Prop.and(Pre);
  return SB;
 }

 public Formula AltSB(Expression r, Lts other) {
  final Formula Prop = this.trans.altStrongBisimulation(r, other.trans);
  final Formula Pre = (this.init.product(other.init)).in(r);
  final Formula SB = Prop.and(Pre);
  return SB;
 }
```

```
public Formula BS(Expression r, Lts other) {
 final Formula Prop = this.trans.branchingSimulation(r, other.trans);
 final Formula Pre = (this.init.product(other.init)).in(r);
 final Formula BB = Prop.and(Pre);
 return BB;
}

public Formula BB(Expression r, Lts other) {
 final Formula Prop = this.trans.branchingBisimulation(r, other.trans);
 final Formula Pre = (this.init.product(other.init)).in(r);
 final Formula BB = Prop.and(Pre);
 return BB;
}

public Formula WS(Expression r, Lts other) {
 final Formula Prop = this.trans.weakSimulation(r, other.trans);
 final Formula Pre = (this.init.product(other.init)).in(r);
 final Formula WB = Prop.and(Pre);
 return WB;
}

public Formula WB(Expression r, Lts other) {
 final Formula Prop = this.trans.weakBisimulation(r, other.trans);
 final Formula Pre = (this.init.product(other.init)).in(r);
 final Formula WB = Prop.and(Pre);
 return WB;
}

}
```

### E.3.3 Clase MTS

```
import kodkod.ast.Formula;
import kodkod.ast.Expression;

/**
 * @author Carolina Dania & Nazareno Aguirre
 *
 */

public class Mts extends Lts {

 public final Transition req;

 public Mts() {
  req = new Transition();
 }

 public Formula SR(Rel r, Mts other) {
  final Formula Prop = this.trans.strongRefinement(r.rel, this.req, other.trans, other.req);
  final Formula IsRel = (this.init.product(other.init)).in(r.rel);
  final Formula SR = Prop.and(IsRel);
  return SR;
 }

 public Formula WR(Expression r, Mts other) {
  final Formula Prop = this.trans.weakRefinement(r, this.req, other.trans, other.req);
  final Formula IsRel = (this.init.product(other.init)).in(r);
  final Formula WR = Prop.and(IsRel);
  return WR;
 }

 public Formula SI(Expression r, Lts other) {
  final Formula Impl = this.trans.strongImplementation(r, this.req, other.trans);
  final Formula IsRel = (this.init.product(other.init)).in(r);
```

```
    final Formula SI = Impl.and(IsRel);
    return SI;
  }

  public Formula WI(Expression r, Lts other) {
    final Formula Impl = this.trans.weakImplementation(r, this.req, other.trans);
    final Formula IsRel = (this.init.product(other.init)).in(r);
    final Formula WI = Impl.and(IsRel);
    return WI;
  }

  public Formula BI(Expression r, Lts other) {
    final Formula Impl = this.trans.branchingImplementation(r, this.req, other.trans);
    final Formula IsRel = (this.init.product(other.init)).in(r);
    final Formula BI = Impl.and(IsRel);
    return BI;
  }

}
```

### E.3.4 Clase Rel

```java
import kodkod.ast.Relation;

/**
 * @author Carolina Dania & Nazareno Aguirre
 *
 */

public class Rel {

 public final Relation rel;

 public Rel() {
  rel = Relation.nary("Rel", 2);
 }

}
```

### E.3.5 Ejemplos de (bi)simulaciones

```
import java.util.List;
import java.util.LinkedList;

import kodkod.ast.Formula;
import kodkod.ast.Relation;
import kodkod.instance.Bounds;
import kodkod.instance.Tuple;
import kodkod.instance.TupleFactory;
import kodkod.instance.Universe;
import kodkod.engine.Solution;
import kodkod.engine.Solver;
import kodkod.engine.satlab.SATFactory;

/**
 * @author Carolina Dania & Nazareno Aguirre
 *
 */

public class ExamplesLtsToLts {

 public final Lts p,q;
 public final Rel rel;
 public final Relation state, action, tau;

 public ExamplesLtsToLts() {
  state = Relation.unary("State");
  action = Relation.unary("Action");
        tau = Relation.unary("Tau");
  p = new Lts();
  q = new Lts();
  rel = new Rel();
 }
```

```java
public Bounds boundsSystem(int states, int actions) {

 final List<String> atoms = new LinkedList<String>();

 for (int i = 0; i < states; i++) {
  atoms.add("S" + i);
 }

 for (int i = 0; i < actions; i++) {
  atoms.add("act" + i);
 }
 atoms.add("tau");

     final Universe universe = new Universe(atoms);
 final TupleFactory factory = universe.factory();
 final Bounds b = new Bounds(universe);
 final String stateMax = "S" + (states - 1);

 b.boundExactly(this.state, factory.range(factory.tuple("S0"), factory.tuple(stateMax)));
 b.boundExactly(this.action, factory.range(factory.tuple("act0"), factory.tuple("tau")));
 b.boundExactly(this.tau, factory.range(factory.tuple("tau"), factory.tuple("tau")));

 final List<Tuple> pTrans = new LinkedList<Tuple>();
 final List<Tuple> qTrans = new LinkedList<Tuple>();

 // Mts

 // P1 y Q1
 pTrans.add(factory.tuple("act1", "S1", "S0"));
 pTrans.add(factory.tuple("act0", "S0", "S1"));
 pTrans.add(factory.tuple("act0", "S0", "S2"));
 pTrans.add(factory.tuple("act1", "S2", "S0"));
 qTrans.add(factory.tuple("act0", "S0", "S2"));
 qTrans.add(factory.tuple("act1", "S2", "S3"));
 qTrans.add(factory.tuple("act0", "S3", "S2"));
 qTrans.add(factory.tuple("act0", "S3", "S1"));
```

```
  qTrans.add(factory.tuple("act1", "S1", "S0"));

  b.boundExactly(this.p.trans.state, factory.range(factory.tuple("S0"), factory.tuple(stateMax)));
  b.boundExactly(this.p.trans.action, factory.range(factory.tuple("act0"), factory.tuple("tau")));
  b.boundExactly(this.p.trans.tau, factory.range(factory.tuple("tau"), factory.tuple("tau")));
  b.boundExactly(this.p.trans.trans, factory.setOf(pTrans));
//  b.bound(this.p.trans.trans, factory.range(factory.tuple("act0","S0","S0"), factory.tuple("tau", stateMax, stateMax)));
  b.boundExactly(this.p.init, factory.range(factory.tuple("S0"), factory.tuple("S0")));

  b.boundExactly(this.q.trans.state, factory.range(factory.tuple("S0"), factory.tuple(stateMax)));
  b.boundExactly(this.q.trans.action, factory.range(factory.tuple("act0"), factory.tuple("tau")));
  b.boundExactly(this.q.trans.tau, factory.range(factory.tuple("tau"), factory.tuple("tau")));
  b.boundExactly(this.q.trans.trans, factory.setOf(qTrans));
//  b.bound(this.q.trans.trans, factory.range(factory.tuple("act0","S0","S0"), factory.tuple("tau", stateMax, stateMax)));
  b.boundExactly(this.q.init, factory.range(factory.tuple("S0"), factory.tuple("S0")));

  // Relations
  final List<Tuple> relUpperBound = new LinkedList<Tuple>();

  String stateI = "";
  String stateF = "";

  for (int i = 0; i < states; i++) {
   for (int j = 0; j < states; j++) {
    stateI = "S"+j;
    stateF = "S"+i;
      relUpperBound.add(factory.tuple(stateI, stateF));
   }
  }

  b.bound(this.rel.rel, factory.setOf(relUpperBound));

  //  b.bound(this.rel.rel, factory.range(factory.tuple("S0","S0"), factory.tuple(stateMax, stateMax)));

  return b;
 }
```

```java
  public static void main(String[] args) {
   System.out.println("Hola mundo");
   // TODO Auto-generated method stub
          final ExamplesLtsToLts model = new ExamplesLtsToLts();
          final Solver solver = new Solver();
          solver.options().setSolver(SATFactory.MiniSat);
          // P1 y Q1
          final Formula show = model.p.BB(model.rel.rel, model.q);
          final Solution sol = solver.solve(show, model.boundsSystem(4, 2));
          System.out.println(show);
          System.out.println(sol);
  }
}
```

### E.3.6 Ejemplos de refinamientos

```
import java.util.List;
import java.util.LinkedList;

import kodkod.ast.Formula;
import kodkod.ast.Relation;
import kodkod.instance.Bounds;
import kodkod.instance.Tuple;
import kodkod.instance.TupleFactory;
import kodkod.instance.Universe;
import kodkod.engine.Solution;
import kodkod.engine.Solver;
import kodkod.engine.satlab.SATFactory;

/**
 * @author Carolina Dania & Nazareno Aguirre
 *
 */

public class ExamplesMtsToLts {

 public final Mts p;
 public final Lts q;
 public final Rel rel;
 public final Relation state, action, tau;

 public ExamplesMtsToLts() {
  state = Relation.unary("State");
  action = Relation.unary("Action");
        tau = Relation.unary("Tau");
  p = new Mts();
  q = new Lts();
  rel = new Rel();
 }
```

```
public Bounds boundsSystemM(int states, int actions) {

 final List<String> atoms = new LinkedList<String>();

 for (int i = 0; i < states; i++) {
  atoms.add("S" + i);
 }

 for (int i = 0; i < actions; i++) {
  atoms.add("act" + i);
 }
 atoms.add("tau");

     final Universe universe = new Universe(atoms);
 final TupleFactory factory = universe.factory();
 final Bounds b = new Bounds(universe);
 final String stateMax = "S" + (states - 1);//, actionMax = "act" + (actions - 1);

 b.boundExactly(this.state, factory.range(factory.tuple("S0"), factory.tuple(stateMax)));
 b.boundExactly(this.action, factory.range(factory.tuple("act0"), factory.tuple("tau")));
 b.boundExactly(this.tau, factory.range(factory.tuple("tau"), factory.tuple("tau")));

 final List<Tuple> pTrans = new LinkedList<Tuple>();
 final List<Tuple> pReq = new LinkedList<Tuple>();
 final List<Tuple> qTrans = new LinkedList<Tuple>();

 // P12 y Q12
 pTrans.add(factory.tuple("tau", "S0", "S1"));
 pTrans.add(factory.tuple("act0", "S0", "S2"));
 pTrans.add(factory.tuple("act0", "S1", "S3"));
 pTrans.add(factory.tuple("act1", "S2", "S4"));
 pTrans.add(factory.tuple("tau", "S2", "S5"));
 pReq.add(factory.tuple("act0", "S0", "S2"));
 pReq.add(factory.tuple("act1", "S2", "S4"));
 pReq.add(factory.tuple("tau", "S2", "S5"));
```

```
qTrans.add(factory.tuple("act0", "S0", "S1"));
qTrans.add(factory.tuple("act0", "S0", "S3"));
qTrans.add(factory.tuple("act1", "S1", "S2"));
qTrans.add(factory.tuple("tau", "S1", "S3"));

b.boundExactly(this.p.trans.state, factory.range(factory.tuple("S0"), factory.tuple(stateMax)));
b.boundExactly(this.p.trans.action, factory.range(factory.tuple("act0"), factory.tuple("tau")));
b.boundExactly(this.p.trans.tau, factory.range(factory.tuple("tau"), factory.tuple("tau")));
b.boundExactly(this.p.trans.trans, factory.setOf(pTrans));
b.boundExactly(this.p.req.state, factory.range(factory.tuple("S0"), factory.tuple(stateMax)));
b.boundExactly(this.p.req.action, factory.range(factory.tuple("act0"), factory.tuple("tau")));
b.boundExactly(this.p.req.tau, factory.range(factory.tuple("tau"), factory.tuple("tau")));
b.boundExactly(this.p.req.trans, factory.setOf(pReq));
b.boundExactly(this.p.init, factory.range(factory.tuple("S0"), factory.tuple("S0")));

b.boundExactly(this.q.trans.state, factory.range(factory.tuple("S0"), factory.tuple(stateMax)));
b.boundExactly(this.q.trans.action, factory.range(factory.tuple("act0"), factory.tuple("tau")));
b.boundExactly(this.q.trans.tau, factory.range(factory.tuple("tau"), factory.tuple("tau")));
b.boundExactly(this.q.trans.trans, factory.setOf(qTrans));
b.boundExactly(this.q.init, factory.range(factory.tuple("S0"), factory.tuple("S0")));

// Relations
final List<Tuple> relUpperBound = new LinkedList<Tuple>();

String stateI = "";
String stateF = "";

for (int i = 0; i < states; i++) {
 for (int j = 0; j < states; j++) {
  stateI = "S"+j;
  stateF = "S"+i;
    relUpperBound.add(factory.tuple(stateI, stateF));
 }
}

b.bound(this.rel.rel, factory.setOf(relUpperBound));
```

```
  return b;
 }

 public static void main(String[] args) {
  System.out.println("Hola mundo");
  // TODO Auto-generated method stub
        final ExamplesMtsToLts model = new ExamplesMtsToLts();
        final Solver solver = new Solver();
        solver.options().setSolver(SATFactory.MiniSat);
        // P12 y Q12
        final Formula show = model.p.BI(model.rel.rel, model.q);
        final Solution sol = solver.solve(show, model.boundsSystemM(6, 2));
        System.out.println(show);
        System.out.println(sol);
 }
}
```

### E.3.7 Ejemplos de implementaciones

```
import java.util.List;
import java.util.LinkedList;

import kodkod.ast.Formula;
import kodkod.ast.Relation;
import kodkod.instance.Bounds;
import kodkod.instance.Tuple;
import kodkod.instance.TupleFactory;
import kodkod.instance.Universe;
import kodkod.engine.Solution;
import kodkod.engine.Solver;
import kodkod.engine.satlab.SATFactory;

/**
 * @author Carolina Dania & Nazareno Aguirre
 *
 */

public class ExamplesMtsToMts {

 public final Mts p,q;
 public final Rel rel;
 public final Relation state, action, tau;

 public ExamplesMtsToMts() {
  state = Relation.unary("State");
  action = Relation.unary("Action");
        tau = Relation.unary("Tau");
  p = new Mts();
  q = new Mts();
  rel = new Rel();
 }
```

```
public Bounds boundsSystemM(int states, int actions) {

 final List<String> atoms = new LinkedList<String>();

 for (int i = 0; i < states; i++) {
  atoms.add("S" + i);
 }

 for (int i = 0; i < actions; i++) {
  atoms.add("act" + i);
 }
 atoms.add("tau");

     final Universe universe = new Universe(atoms);
 final TupleFactory factory = universe.factory();
 final Bounds b = new Bounds(universe);
 final String stateMax = "S" + (states - 1);//, actionMax = "act" + (actions - 1);

 b.boundExactly(this.state, factory.range(factory.tuple("S0"), factory.tuple(stateMax)));
 b.boundExactly(this.action, factory.range(factory.tuple("act0"), factory.tuple("tau")));
 b.boundExactly(this.tau, factory.range(factory.tuple("tau"), factory.tuple("tau")));

 final List<Tuple> pTrans = new LinkedList<Tuple>();
 final List<Tuple> pReq = new LinkedList<Tuple>();
 final List<Tuple> qTrans = new LinkedList<Tuple>();
 final List<Tuple> qReq = new LinkedList<Tuple>();

 // P7 y Q7
 pTrans.add(factory.tuple("act0", "S0", "S1"));
 pTrans.add(factory.tuple("act1", "S1", "S2"));
 pTrans.add(factory.tuple("act1", "S2", "S1"));
 pTrans.add(factory.tuple("act2", "S2", "S0"));
 pReq.add(factory.tuple("act0", "S0", "S1"));
 pReq.add(factory.tuple("act1", "S1", "S2"));
 qTrans.add(factory.tuple("act0", "S0", "S1"));
 qTrans.add(factory.tuple("act1", "S1", "S2"));
```

```
qTrans.add(factory.tuple("act1", "S2", "S1"));
qTrans.add(factory.tuple("act2", "S2", "S0"));
qReq.add(factory.tuple("act0", "S0", "S1"));
qReq.add(factory.tuple("act1", "S1", "S2"));
qReq.add(factory.tuple("act2", "S2", "S0"));


b.boundExactly(this.p.trans.state, factory.range(factory.tuple("S0"), factory.tuple(stateMax)));
b.boundExactly(this.p.trans.action, factory.range(factory.tuple("act0"), factory.tuple("tau")));
b.boundExactly(this.p.trans.tau, factory.range(factory.tuple("tau"), factory.tuple("tau")));
b.boundExactly(this.p.trans.trans, factory.setOf(pTrans));
b.boundExactly(this.p.req.state, factory.range(factory.tuple("S0"), factory.tuple(stateMax)));
b.boundExactly(this.p.req.action, factory.range(factory.tuple("act0"), factory.tuple("tau")));
b.boundExactly(this.p.req.tau, factory.range(factory.tuple("tau"), factory.tuple("tau")));
b.boundExactly(this.p.req.trans, factory.setOf(pReq));
b.boundExactly(this.p.init, factory.range(factory.tuple("S0"), factory.tuple("S0")));

b.boundExactly(this.q.trans.state, factory.range(factory.tuple("S0"), factory.tuple(stateMax)));
b.boundExactly(this.q.trans.action, factory.range(factory.tuple("act0"), factory.tuple("tau")));
b.boundExactly(this.q.trans.tau, factory.range(factory.tuple("tau"), factory.tuple("tau")));
b.boundExactly(this.q.trans.trans, factory.setOf(qTrans));
b.boundExactly(this.q.req.state, factory.range(factory.tuple("S0"), factory.tuple(stateMax)));
b.boundExactly(this.q.req.action, factory.range(factory.tuple("act0"), factory.tuple("tau")));
b.boundExactly(this.q.req.tau, factory.range(factory.tuple("tau"), factory.tuple("tau")));
b.boundExactly(this.q.req.trans, factory.setOf(qReq));
b.boundExactly(this.q.init, factory.range(factory.tuple("S0"), factory.tuple("S0")));

// Relations
final List<Tuple> relUpperBound = new LinkedList<Tuple>();

String stateI = "";
String stateF = "";

for (int i = 0; i < states; i++) {
 for (int j = 0; j < states; j++) {
  stateI = "S"+j;
```

```
      stateF = "S"+i;
        relUpperBound.add(factory.tuple(stateI, stateF));
    }
  }

  b.bound(this.rel.rel, factory.setOf(relUpperBound));
  return b;
 }

 public static void main(String[] args) {
  System.out.println("Hola mundo");
  // TODO Auto-generated method stub
        final ExamplesMtsToMts model = new ExamplesMtsToMts();
        final Solver solver = new Solver();
        solver.options().setSolver(SATFactory.MiniSat);
        // P7 y Q7
        final Formula show = model.p.WR(model.rel.rel, model.q);
        final Solution sol = solver.solve(show, model.boundsSystemM(3, 3));
        System.out.println(show);
        System.out.println(sol);
 }
}
```

# Bibliografía

[BCU06] Greg Brunet, Marsha Chechik, and Sebastián Uchitel. Properties of behavioural model merging. In Jayadev Misra, Tobias Nipkow, and Emil Sekerinski, editors, *FM*, volume 4085 of *Lecture Notes in Computer Science*, pages 98–114. Springer, 2006.

[BRJ98] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. AddisonWesley, 1998.

[Cla03] Koen Claessen. New techniques that improve mace-style finite model finding. In *Proceedings of the CADE-19 Workshop: Model Computation - Principles, Algorithms, Applications*, 2003.

[DFCU08] Nicolás D'Ippolito, Dario Fischbein, Marsha Chechik, and Sebastián Uchitel. Mtsa: The modal transition system analyser. In *ASE*, pages 475–476. IEEE, 2008.

[DFFU07] Nicolás D'Ippolito, Dario Fischbein, Howard Foster, and Sebastián Uchitel. Mtsa: Eclipse support for modal transition systems construction, analysis and elaboration. In Li-Te Cheng, Alessandro Orso, and Martin P. Robillard, editors, *ETX*, pages 6–10. ACM, 2007.

[Dil94] Antoni Diller. *Z: An Introduction to Formal Methods*. John Wiley & Sons, Inc., New York, NY, USA, 1994.

[Dis]

[DPP01] Agostino Dovier, Carla Piazza, and Alberto Policriti. A fast bisimulation algorithm. In Gérard Berry, Hubert Comon, and Alain Finkel, editors, *CAV*, volume 2102 of *Lecture Notes in Computer Science*, pages 79–90. Springer, 2001.

[ES03] Niklas Eén and Niklas Sörensson. An extensible sat-solver. In Enrico Giunchiglia and Armando Tacchella, editors, *SAT*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518. Springer, 2003.

[FGPA05] Marcelo F. Frias, Juan P. Galeotti, Carlos López Pombo, and Nazareno Aguirre. Dynalloy: upgrading alloy with actions. In Gruia-Catalin Roman, William G. Griswold, and Bashar Nuseibeh, editors, *ICSE*, pages 442–451. ACM, 2005.

[Fis] Dario Fischbein. On strong refinement outline or on mts strong semantics/refinement and merge. Notas no publicadas.

[Fis06] Dario Fischbein. Branching semantics for modal transition systems. Master's thesis, Universidad de Buenos Aires, 2006.

[FPB+05] Marcelo F. Frias, Carlos López Pombo, Gabriel A. Baum, Nazareno Aguirre, and T. S. E. Maibaum. Reasoning about static and dynamic properties in alloy: A purely relational approach. *ACM Trans. Softw. Eng. Methodol.*, 14(4):478–526, 2005.

[FUB06] Dario Fischbein, Sebastián Uchitel, and Víctor A. Braberman. A foundation for behavioural conformance in software product line architectures. In Robert M. Hierons and Henry Muccini, editors, *ROSATEA*, pages 39–48. ACM, 2006.

[GJM02] Carlo Ghezzi, Mehdi Jazayeri, and Dino Mandrioli. *Fundamentals of Software Engineering*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2002.

[GN07] Eugene Goldberg and Yakov Novikov. Berkmin: A fast and robust sat-solver. *Discrete Appl. Math.*, 155(12):1549–1561, 2007.

[GV90] Jan Friso Groote and Frits W. Vaandrager. An efficient algorithm for branching bisimulation and stuttering equivalence. In Mike Paterson, editor, *ICALP*, volume 443 of *Lecture Notes in Computer Science*, pages 626–638. Springer, 1990.

[Hut05] Michael Huth. Refinement is complete for implementations. *Formal Asp. Comput.*, 17(2):113–137, 2005.

[Ing87] Leif Ingevaldsson. *JSP - a Practical Method of Program Design*. Krieger Publishing Co., Inc., Melbourne, FL, USA, 1987.

[Jac03] Daniel Jackson. Alloy: A logical modelling language. In Didier Bert, Jonathan P. Bowen, Steve King, and Marina A. Waldén, editors, *ZB*, volume 2651 of *Lecture Notes in Computer Science*, page 1. Springer, 2003.

[Jac06] Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, 2006.

[JJD98] Daniel Jackson, Somesh Jha, and Craig Damon. Isomorph-free model enumeration: A new method for checking relational specifications. *ACM Trans. Program. Lang. Syst.*, 20(2):302–343, 1998.

[KM06] Jeff Kramer and Jeff Magee. *Concurrency: State Models Java Programs, 2nd Edition*. Worldwide Series in Computer Science. John Wiley Sons, March 2006.

[LT88] Kim Guldstrand Larsen and Bent Thomsen. A modal process logic. In *LICS*, pages 203–210. IEEE Computer Society, 1988.

[LX90] Kim Guldstrand Larsen and Liu Xinxin. Equation solving using modal transition systems. In *LICS*, pages 108–117. IEEE Computer Society, 1990.

[McC03] William McCune. Mace4 reference manual and guide. *CoRR*, cs.SC/0310055, 2003.

[MFM04] Yogesh S. Mahajan, Zhaohui Fu, and Sharad Malik. Zchaff2004: An efficient sat solver. In Holger H. Hoos and David G. Mitchell, editors, *SAT (Selected Papers*, volume 3542 of *Lecture Notes in Computer Science*, pages 360–375. Springer, 2004.

[Mil89] R. Milner. *Communication and concurrency*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989.

[Som00] I. Sommerville. *Software Engineering*. Addison Wesley, sexth edition, 2000.

[Sto96] Neil R. Storey. *Safety Critical Computer Systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1996.

[TJ07] Emina Torlak and Daniel Jackson. Kodkod: A relational model finder. pages 632–647. 2007.

[vGW96] Rob J. van Glabbeek and W. P. Weijland. Branching time and abstraction in bisimulation semantics. *J. ACM*, 43(3):555–600, 1996.

[Win90] Jeannette M. Wing. A specifier's introduction to formal methods. *Computer*, 23(9):8–23, 1990.