MTSA: Una herramienta de Model Checking de MTS

Tesista
Nicolás Roque D'Ippolito
ndippolito(at)dc.uba.ar
L.U. 244/99

Director
Sebastian Uchitel
suchitel(at)dc.uba.ar

Octubre 2007

Abstract

El modelado de sistemas es un problema difícil y costoso en la Ingeniería de Software. Este puede ser mitigado construyendo modelos parciales dando así feedback temprano del comportamiento de las componentes que se van conociendo del sistema. De este modo se va llegando iterativamente a un modelo en el cual ya no hay indefiniciones. El formalismo Modal Transition Systems (MTS) permite modelar el comportamiento requerido, prohibido y no definido de un sistema. Estos modelos son caracterizados por sus implementaciones, es decir, los modelos tales que el comportamiento desconocido en el MTS original ha sido definido. En esta tesis se avanzó en la teoría para poder verificar propiedades y analizar modelos parciales (MTS) utilizando model checking. Se implementó Modal Transition System Analizer (MTSA) una herramienta que soporta la construcción, elaboración y análisis de MTS. MTSA de desarrolló como una extensión del ya existente Labeled Transition System Analizer (LTSA).

Índice general

Capítulo 1

Introducción

Los requerimientos y el diseño de los sistemas de software son más simples de entender y analizar a través de la construcción de modelos de comportamiento, es decir, construyendo descripciones formales del comportamiento esperado de los sistemas. Este es, en general, el enfoque tradicional de la ingeniería del software para la construcción de sistemas complejos. La mayor ventaja de utilizar modelos es que éstos pueden ser analizados para aumentar la correctitud y la consistencia tanto de los requerimientos relevados como del diseño realizado. Existen varias técnicas probadas y útiles para realizar análisis de modelos de comportamiento (visualización, animación, model checking, etc), todas ellas intentan dar soporte a los ingenieros en la tarea de detectar errores en etapas tempranas del proceso de desarrollo, reduciendo así el costo y el impacto de los mismos.

A pesar de que se ha visto que el modelado de sistemas es muy útil para encontrar tempranamente inconsistencias tanto en el diseño [5] como en los requerimientos, la adopción de estas técnicas ha sido lenta. Esto se debe en parte a una incompatibilidad entre la esencia iterativa e incremental de la mayoría de los procesos para desarrollo de software utilizados hoy en día y ciertas características de los modelos de comportamiento tradicionales. Por un lado, debido a la esencia iterativa e incremental de los procesos de desarrollo de software, la descripción que se tiene del sistema, tiende a ser parcial, dejando algunos aspectos del comportamiento indefinidos hasta alcanzar etapas posteriores. Por otro lado, los modelos de comportamiento tradicionales como los *Labelled Transition Systems* (LTS) [20] y los diagramas de transición de estados [22], asumen que a algún nivel de abstracción se tiene la descripción completa del sistema a construir: la existencia de una transición modela el comportamiento que se espera que el sistema admita (i.e comportamiento requerido), mientras que la ausencia de transiciones modela el comportamiento que el sistema prohíbe. En resumen, los modelos de comportamiento tradicionales describen el comportamiento requerido y prohibido pero no aceptan definición de comportamiento parcial o indefinido, en otras palabras, cuando las ventajas de construir modelos son mayores, no se tiene una descripción completa del sistema.

Nuestra visión adopta a los *Modal Transition Systems* (MTS) [24] como las bases

para describir el comportamiento parcial (i.e comportamiento definido y no definido) de los sistemas. Los MTS son la extensión natural de los LTS, los cuales han dado muy buenos resultados como herramienta para modelar y analizar el comportamiento de sistemas. Los sistemas son modelados como un conjunto de componentes o sub-sistemas que interactúan comunicándose y sincronizándose para modelar el comportamiento total del sistema. Cada componente es descrito como un *sistema de transiciones* en el cual éstas últimas representan la interacción del componente con el ambiente.

En un MTS cada transición puede ser *requerida* o *maybe*. Éstas últimas representan comportamiento que en el modelo final puede ser requerido o prohibido, pero que aún no se tiene información al respecto. La unión del comportamiento maybe y requerido se lo conoce como comportamiento *posible*. Un MTS sin transiciones maybe es un modelo que esta totalmente definido y por lo tanto es un LTS.

La noción de refinamiento [24] entre MTS captura formalmente la intuición de ser "más definido que" o de "tener más información que" y provee una forma elegante de describir el proceso de elaboración de los modelos parciales como uno en el cual se va adquiriendo información sobre el comportamiento esperado y va siendo incorporada al modelo. A partir de un MTS con cierto grado de indefinición es posible obtener un modelo totalmente definido, refinando sucesivamente el modelo original. Un paso de refinamiento corresponde *intuitivamente* a remover una transición maybe o bien reemplazarla por una requerida. Se puede mostrar que la relación de refinamiento preserva las propiedades temporales de los modelos [3, 12], ésto significa que al refinar un MTS garantizamos que todas las propiedades que en el modelo original eran verdaderas (falsas) también lo serán en el modelo refinado. El modelo resultante del proceso de refinar iterativamente un modelo M hasta que no queden transiciones maybe en él, recibe el nombre de *implementación*. El refinamiento puede ser visto como una forma de ir acotando las posibles implementaciones de la descripción parcial provista por un MTS. Es simple ver que las implementaciones al ser modelos MTS sin transiciones maybe son en particular modelos LTS. Por último notemos que un MTS caracteriza un conjunto de implementaciones posibles.

Por lo mencionado hasta aquí, entendemos que utilizando MTS es posible, en etapas tempranas del proceso de desarrollo describir el comportamiento conocido de un sistema y analizarlo aún sin tener conocimiento completo de lo que se espera del mismo.

En nuestro trabajo implementamos *Modal Transition Systems Analyser* (MTSA), una herramienta que soporta la construcción, análisis, elaboración y verificación de MTS. Basándonos en la noción de que los MTS son la extensión natural de los LTS y en los teoremas fundamentales del model checking implementamos MTSA como una extensión de Labelled Transition Systems Analiser (LTSA) [25], aprovechando todo su desarrollo y experiencia en la manipulación y chequeo sobre LTS. Para construirla extendimos la teoría y los algoritmos [3, 29, 24] para minimizar, componer y determinizar MTS. MTSA soporta la generación de MTSs a partir de un álgebra de procesos (Finite State Process), así como también es posible obtener una representación gráfica de los modelos, animarlos y operar sobre ellos utilizando operadores de composición, determinización y minimización. Como parte de la funcionalidad de la herramienta incorporamos la posibilidad de verificar si existe alguna relación de refinamiento en-

tre dos modelos dados, así como también si ellos son equivalentes. Implementamos el análisis de deadlocks sobre MTS. Por último también es posible realizar model checking sobre MTS a partir de un modelo y propiedades escritas en Fluent Linear Temporal Logic (FLTL).

Capítulo 2

Fundamentos

2.1. Labelled Transition System

Un Labelled Transition System (LTS) es un sistema de transición de estados en el cual las transiciones están etiquetadas con acciones [20]. El conjunto de acciones de un LTS se llama *alfabeto de comunicación* y constituye las interacciones que el sistema modelado puede tener con el ambiente. Además un LTS puede tener transiciones etiquetadas con τ, representando transiciones que no son observables por el ambiente.

Definición 1 (Labelled Transition Systems (LTS)) *Sea States el conjunto universal de estados, Act el conjunto universal de etiquetas de acciones observables, y sea* $Act_\tau = Act \cup \{\tau\}$. *Un* sistema de transición etiquetado *(LTS, labelled transition system) es una tupla* $P = (S, L, \Delta, s_0)$, *donde* $S \subseteq States$ *es un conjunto finito de estados,* $L \subseteq Act_\tau$ *es un conjunto de etiquetas,* $\Delta \subseteq (S \times L \times S)$ *es una relación de transición entre estados, y* $s_0 \in S$ *es el estado inicial. Usamos* $\alpha P = L \setminus \{\tau\}$ *para notar el alfabeto de comunicación de* P.

En la figuras 2.1 y 2.2 mostramos LTSs cuyo alfabeto está formado por las acciones { getReadLock, relReadLock } . A lo largo de este trabajo tomaremos al estado etiquetado con el numero cero como el estado inicial del LTS salvo que se aclare lo contrario. Las semánticas existentes asumen que un LTS da una descripción completa del comportamiento en función de su alfabeto. En otras palabras la presencia de una transición describe comportamiento requerido y su ausencia modela el comportamiento prohibido del sistema.

Consideremos el LTS $\mathcal{A}$ que modela el locking de objetos. Comenzando por el estado 0, este modelo permite secuencias de las acciones getReadLock y relReadLock alternadamente. Debido a la naturaleza de los LTS, $\mathcal{A}$ no puede permite dos acciones getReadLock seguidas sin una relReadLock entre ellas. El LTS $\mathcal{A}$ describe un modelo de locking que admite únicamente un lector a la vez.

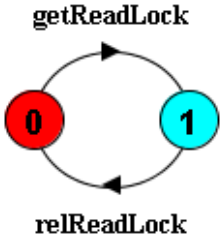

Figura 2.1: Modelo $\mathcal{A}$

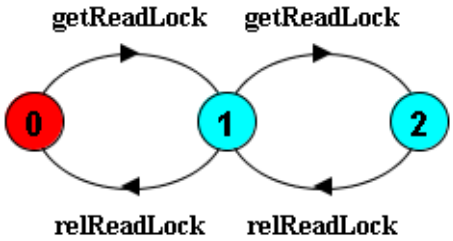

Figura 2.2: Modelo $\mathcal{B}$

Definición 2 (Simulación de LTS) *Sea $\wp$ el universo de todos los LTSs. Un LTS L simula a otro LTS M, escrito $M \sqsubseteq_{\mathrm{s}} L$, si (M, L) está contenido en alguna relación de simulación $R \subseteq \wp \times \wp$ para la cual vale que $\forall \ell \in Act$:*

$$1. \quad (M \xrightarrow{\ell} M') \Rightarrow (\exists L' \cdot L \xrightarrow{\ell} L' \ \wedge \ (M', L') \in R)$$

En el caso en que sea de interés cuál es la relación de simulación R entre dos modelos M y N, notaremos $M \sqsubseteq_{\mathrm{R}} N$ si N simula a M por medio de la relación R.

Básicamente un LTS L simula a otro M si cada vez que M puede moverse a un estado M' por una acción ℓ llegando a un estado M' y L puede moverse por la misma acción y llegando a un estado L' que a su vez simula a M'. El modelo $\mathcal{C}$ (Fig. 2.3) es capaz de simular cualquiera de los modelos anteriormente mencionados dado que admite cualquier combinación de acciones para el alfabeto { getReadLock, relReadLock } , en el caso del modelo $\mathcal{A}$, es cierta la relación $\mathcal{A} \sqsubseteq_{\mathrm{s}} \mathcal{C}$ y la relación de simulación entre ellos es $R =\{ (\mathcal{A}_0, \mathcal{C}_0), (\mathcal{A}_1, \mathcal{C}_0) \}$; pero ninguno de los anteriores ($\mathcal{A}$ y $\mathcal{B}$) es capaz de simular a $\mathcal{C}$, justamente por que éste no impone restricciones sobre la política de locking de objetos mas allá de establecer el nombre de las acciones.

Figura 2.3: Modelo $\mathcal{C}$

Definición 3 (Bisimulación Fuerte) *Sea $\wp$ el universo de todos los LTS, y $P, Q \in \wp$. P y Q son equivalentes por bisimilación fuerte, denotado $P \sim Q$, si $\alpha P = \alpha Q$ y*

(P, Q) esta contenido en alguna relación de bisimulación $R \subseteq \wp \times \wp$ para la cual vale lo siguiente $\ell \in Act_\tau$:

$$1. \quad (P \xrightarrow{\ell} P') \Rightarrow (\exists Q' \cdot Q \xrightarrow{\ell} Q' \ \wedge \ (P', Q') \in R)$$
$$2. \quad (Q \xrightarrow{\ell} Q') \Rightarrow (\exists P' \cdot P \xrightarrow{\ell} P' \ \wedge \ (P', Q') \in R)$$

Informalmente, dos modelos son equivalentes por una relación fuerte si sus estados iniciales son fuertemente equivalentes y dos estado son fuertemente equivalentes si para toda acción que pueda ser ejecutada en uno de ellos que conduce a un estado B, en el otro puede ser ejecutada la misma acción llegando a un estado B', donde B y B' son fuertemente equivalentes. Esta equivalencia no distingue las transiciones τ como transiciones especiales y no observables por el ambiente. En la figura 2.4 mostramos el modelo $\mathcal{D}$. $\mathcal{D}$ es equivalente a $\mathcal{C}$ ($\mathcal{C} \sim \mathcal{D}$) y su relación de bisimulación es trivialmente $R = \{ (\mathcal{C}_0, \mathcal{D}_0), (\mathcal{C}_0, \mathcal{D}_1), (\mathcal{C}_0, \mathcal{D}_2) \}$.

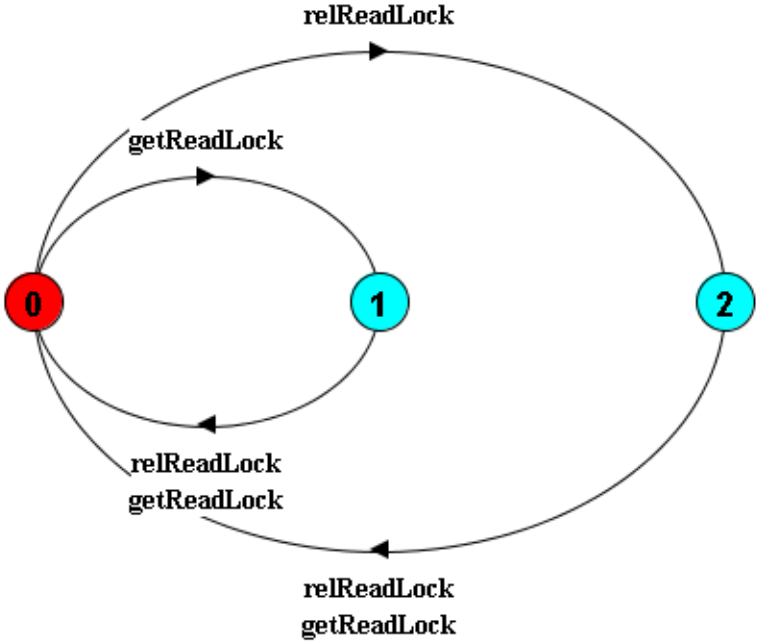

Figura 2.4: Modelo $\mathcal{D}$

Definición 4 (Bisimulación Débil) *Sea $\wp$ el universo de todos los LTS, y $P, Q \in \wp$. P y Q son equivalentes por bisimilación débil, denotado $P \approx Q$, si $\alpha P = \alpha Q$ y (P, Q) esta contenido en alguna relación de bisimulación $R \subseteq \wp \times \wp$ para la cual vale lo siguiente $\ell \in Act_\tau$:*

$$1. \quad (P \xrightarrow{\ell} P') \Rightarrow (\exists Q' \cdot Q \stackrel{\hat{\ell}}{\Longrightarrow} Q' \ \wedge \ (P', Q') \in R)$$
$$2. \quad (Q \xrightarrow{\ell} Q') \Rightarrow (\exists P' \cdot P \stackrel{\hat{\ell}}{\Longrightarrow} P' \ \wedge \ (P', Q') \in R)$$

Esta noción de equivalencia compara el comportamiento observacional de los modelos ignorando las acciones no visibles (transiciones-τ). Algunos autores se refieren a esta equivalencia como *equivalencia observacional*. Esta equivalencia es mas laxa que la anterior. En la figura 2.5 vemos un modelo que es equivalente por una bisimulación débil a $\mathcal{A}$ y una posible relación de bisimulación entre ellos es $R = \{ (\mathcal{A}_0, \mathcal{E}_0), (\mathcal{A}_1, \mathcal{E}_1), (\mathcal{A}_1, \mathcal{E}_2) \}$.

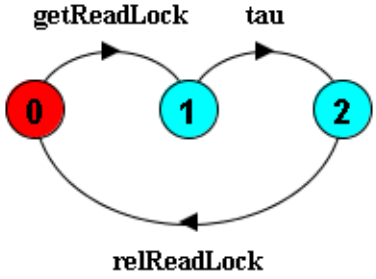

Figura 2.5: Modelo $\mathcal{E}$

La bisimulación, tanto fuerte como débil, requiere que exista una relación de simulación entre los modelos tal que su inversa también lo sea. Una forma mas laxa de entender la equivalencia de LTS es conocida como Equivalencia por Simulación. Esta relación simplemente requiere que los modelos puedan simularse mutuamente, independientemente de si lo hacen bajo una relación en un sentido y su inversa en el opuesto Es simple ver que la bisimulación es un caso particular de la simulación por equivalencia.

Definición 5 (Equivalencia por Simulación) *Sea $\wp$ el universo de todos los LTSs y M y N LTSs. Decimos que M y N equivalentes por simulación ($M \lesseqgtr N$) si $\alpha M = \alpha N$ y existen R y T relaciones de simulación $R, T \subseteq \wp \times \wp$ tales que:*

$$M \sqsubseteq_{\mathrm{R}} N \wedge \; M \sqsubseteq_{\mathrm{T}} N$$

En el caso de la equivalencia por simulación la única diferencia entre la versión fuerte y la débil radica en si se toman relaciones de simulación fuertes o débiles entre los modelos.

La figura 2.1 muestra un ejemplo de dos LTS que son equivalentes por simulación pero no bisimilares. En este ejemplo vemos que los modelos pueden simularse entre si pero no existe relación de bisimulación entre ellos. Es posible entender esta diferencia observando la rama $a \rightarrow b \rightarrow c$ del modelo $\mathcal{W}$. Esta puede ser simulada en $\mathcal{W}$ únicamente por la rama que luego de a admite tanto $b \rightarrow c$ como $b \rightarrow d$, pero obviamente esta ultima no puede ser simulada por la primera. Lo mismo pasa al tomar la rama $a \rightarrow b \rightarrow d$ del modelo $\mathcal{W}$.

El *ocultamiento* es una operación que hace que un conjunto de acciones de un modelo sean no observables por su ambiente reduciendo el alfabeto del modelo y reemplazando las transiciones etiquetadas con una acción en el conjunto de ocultamiento por τ.

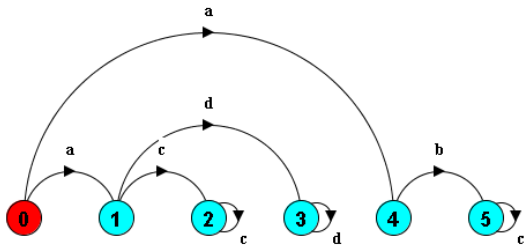

Figura 2.6: Modelo $\mathcal{W}$

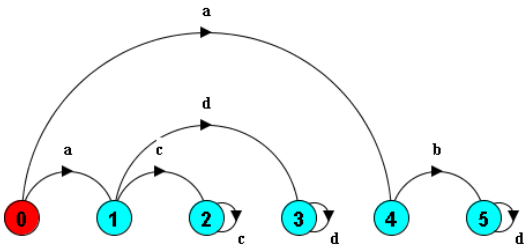

Figura 2.7: Modelo $\mathcal{V}$

Figura 2.8: Modelos equivalentes por simulación no bisimilares

$$\frac{M \xrightarrow{\ell} M'}{(M \backslash X) \xrightarrow{\ell} (M' \backslash X)} \; \ell \notin X, \; \gamma \in \{r, m\} \qquad \frac{M \xrightarrow{\ell} M'}{(M \backslash X) \xrightarrow{\tau} (M' \backslash X)} \; \ell \in X, \; \gamma \in \{r, m\}$$

Figura 2.9: Reglas para el operador de ocultamiento

Definición 6 (Ocultamiento) *Sea M un LTS (S, L, Δ, s_0) y $Z \subseteq Act$ un conjunto de acciones observables. M con las acciones de Z ocultas, notado $M \setminus X$ es un LTS $(S, L \setminus X, \Delta', s_0)$ donde Δ' son las relaciones más pequeñas que satisfacen las reglas en la figura 2.9 tal que $\gamma \in \{r, m\}$. Usamos $M@Z$ para decir $M \setminus (Act \setminus Z)$.*

2.2. Modal Transition Systems

Los Modal Transition Systems (MTS) [24] permiten modelar explícitamente aquello que no es conocido respecto del comportamiento de un sistema. Ellos extienden a los LTS con un conjunto adicional de transiciones que modelan las interacciones con el ambiente que el sistema no puede garantizar que vaya a proveer, así como tampoco puede garantizar que vayan a ser prohibidas, en otras palabras, los MTS incorporan transiciones para modelar las interacciones que todavía no están definidas.

Definición 7 (Modal Transition Systems) *Un Modal Transition System (MTS) M es una estructura $(S, L, \Delta^r, \Delta^p, s_0)$, en la cual $\Delta^r \subseteq \Delta^p$, (S, L, Δ^r, s_0) es un LTS que representa el comportamiento requerido del sistema y (S, L, Δ^r, s_0) es un LTS que representa el comportamiento posible (no necesariamente requerido) del sistema. Utilizaremos $\alpha M = L \setminus \{\tau\}$ para denotar el alfabeto de comunicación de M.*

En la figura 2.10 mostramos la representación gráfica de un MTS. El modelo $\mathcal{F}$ modela una política de locking de lectura en la cual el lock de lectura puede ser tomado por lo menos por un lector en cualquier momento pero no impone restricciones sobre lectores concurrentes. Las transiciones etiquetadas con el signo de pregunta "'?"' son aquellas que pertenecen al conjunto $\Delta^p \setminus \Delta^r$. Nos referimos a estas transiciones como transiciones "'maybe"', para distinguirlas de las transiciones posibles que son tambien requeridas (i.e. las transiciones en Δ^r). Los LTS son un tipo especial de MTSs que no posee transiciones maybe: por lo tanto los modelos $\mathcal{A}$ y $\mathcal{B}$ son también MTSs.

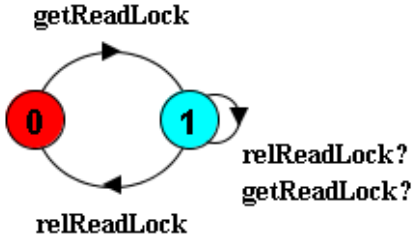

Figura 2.10: Modelo $\mathcal{F}$

Dado un MTS $M = (S, L, \Delta^r, \Delta^p, s_0)$, decimos que M avanza a M' sobre ℓ por una transición requerida (denotado $M \xrightarrow{\ell}_r M'$) si $M' = (S, L, \Delta^r, \Delta^p, s'_0)$ y $(s_0, \ell, s'_0) \in \Delta^r$. Además decimos que M avanza a M' sobre ℓ por una transición maybe (denotado $M \xrightarrow{\ell}_m M'$) si $(s_0, \ell, s'_0) \in \Delta^p \setminus \Delta^r$ y M avanza a M' sobre ℓ por una transición posible (denotado $M \xrightarrow{\ell}_p M'$) si $M \xrightarrow{\ell}_r M'$ o $M \xrightarrow{\ell}_m M'$ (i.e. $(s_0, \ell, s'_0) \in \Delta^p$). Escribimos $M \xrightarrow{\ell}_p M'$ abreviando $\exists M' \cdot M \xrightarrow{\ell}_p M'$. Decimos que M prohíbe ℓ (denotado $M \not\xrightarrow{\ell}$) si desde M no hay transiciones maybe o requeridas sobre ℓ.

Finalmente, para un MTS $M = (S, L, \Delta^r, \Delta^p, s_0)$ y un estado $n \in S$, denotamos el cambio de estado inicial de M de s_0 a n como M_n. Por ejemplo, algunas transiciones del MTS $\mathcal{C}$ mostradas en la figura 2.3 son $C_0 \xrightarrow{getReadLock}_r C_1$ (entre el estado 0 y el 1) y $C_1 \xrightarrow{getReadLock}_m C_1$ (un ciclo sobre el estado 1). Adicionalmente, llamamos $\alpha M \cap \alpha N$ al conjunto de las acciones compartidas M y N, y $(\alpha M \setminus \alpha N) \cup (\alpha N \setminus \alpha M)$ al conjunto de acciones *no compartidas* por M y N.

En este trabajo asociamos cada MTS M con su alfabeto de comunicación αM, extendiendo la representación de [24]. El alfabeto de comunicación es el conjunto de eventos que son relevantes para el modelo, i.e., el alcance de una descripción parcial. A menos que se aclare lo contrario asumiremos que el alfabeto de comunicación de un MTS coincide con el conjunto de acciones observables para las cuales el MTS tiene alguna transición. Sin embargo este no es necesariamente el caso: un MTS puede no incluir transiciones etiquetadas con una acción de su alfabeto, modelando que esa acción esta prohibida de ocurrir desde todos los estados del MTS.

El refinamiento de MTSs captura la noción de elaboración de una descripción parcial en una más completa, en la cual se ha ganado algún conocimiento respecto del comportamiento maybe. Intuitivamente un paso en el refinamiento de MTSs es convertir una transición maybe en requerida, o eliminarla del modelo. Un MTS N refina a un MTS M si N preserva todo el comportamiento requerido y prohibido de M, en otras palabras, N refina a M si N puede simular el comportamiento requerido de M y M puede simular el comportamiento posible de N.

Definición 8 (Refinamiento fuerte) *Sea $\wp$ el universo de todos los MTSs. Un MTS N es un refinamiento de un MTS M, escrito $M \preceq N$, si (M, N) está contenido en*

alguna relación de refinamiento $R \subseteq \wp \times \wp$ *para la cual vale que* $\forall \ell \in Act$*:*

1. $(M \xrightarrow{\ell}_r M') \Rightarrow (\exists N' \cdot N \xrightarrow{\ell}_r N' \ \wedge \ (M', N') \in R)$
2. $(N \xrightarrow{\ell}_p N') \Rightarrow (\exists M' \cdot M \xrightarrow{\ell}_p M' \ \wedge \ (M', N') \in R)$

Notemos que la segunda condición garantiza que si N tiene una transición requerida entonces M tiene una requerida o una maybe, mientras que si N tiene una transición maybe, entonces M tiene una transición maybe ya que de otra forma se violaría la primera condición. Otra cuestion interesante para observar es que esta definición es muy parecida a la definición de bisimulación fuerte 3. Por último, notaremos $M \preceq_R N$ en el caso en que la relación de refinamiento entre M y N sea R y nos interese mostrar que es esta última relación que estamos teniendo en cuenta. Si el diseñador decidiera a partir del modelo $\mathcal{F}$ tomar la decisión de eliminar la transición $1 \xrightarrow{relReadLock}_m 1$ y transformar la transición $1 \xrightarrow{getReadLock}_m 1$ en requerida ($1 \xrightarrow{relReadLock}_r 1$), obtendría el modelo $\mathcal{G}$ (Fig. 2.11), que es refinamiento de F por la relación $R = \{ (g_0, f_0), (g_1, f_1) \}$.

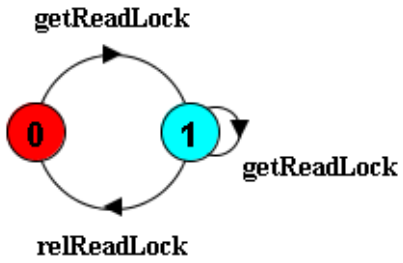

Figura 2.11: Modelo $\mathcal{G}$

A pesar de que el refinamiento captura la noción de la elaboración de un modelo, este requiere que los alfabetos de los modelos a ser comparados sean iguales. En la práctica, la elaboración de un modelo puede conducir a un aumento del alfabeto del modelo del sistema para describir aspectos del comportamiento que previamente no habían sido tenido en cuenta. Para capturar estos aspectos de la elaboración de modelos, se han introducido dos conceptos: Ocultamiento y Refinamiento Observacional (refinamiento débil).

Definición 9 (Ocultamiento) *Sea* M *un MTS* $(S, L, \Delta^r, \Delta^p, s_0)$ *y* $Z \subseteq Act$ *un conjunto de acciones observables.* M *con las acciones de* Z *ocultas, notado* $M \setminus X$ *es un MTS* $(S, L \setminus X, \Delta^{r'}, \Delta^{p'}, s_0)$ *donde* $\Delta^{r'}$ *y* $\Delta^{p'}$ *son las relaciones más chicas que satisfacen las reglas en la figura 2.12 tal que* $\gamma \in \{r, m\}$*. Usamos* $M@Z$ *para decir* $M \setminus (Act \setminus Z)$*.*

Sea $w = w_1, \ldots, w_k$ una palabra sobre Act_τ. $M \xrightarrow{w}_r M'$ significa que existe $M_0, \ldots, M_k$ tal que $M_0 = M$, $M_k = M'$, y $M_i \xrightarrow{w_{i+1}}_r M_{i+1}$ para $0 \leq i < k$. $M \xrightarrow{w}_m M'$ significa que existe $M_0, \ldots, M_k$ tal que $M_0 = M$, $M_k = M'$, $M_i \xrightarrow{w_{i+1}}_p M_{i+1}$ para

$$\frac{M \xrightarrow{\ell}_\gamma M'}{(M\backslash X) \xrightarrow{\ell}_\gamma (M'\backslash X)}\ \ell \notin X,\ \gamma \in \{r,m\} \qquad \frac{M \xrightarrow{\ell}_\gamma M'}{(M\backslash X) \xrightarrow{\tau}_\gamma (M'\backslash X)}\ \ell \in X,\ \gamma \in \{r,m\}$$

Figura 2.12: Reglas para el operador de ocultamiento

$0 \leq i < k$, y $\exists j \cdot 0 \leq j \leq k \cdot M_j \xrightarrow{w_{j+1}}_m M_{j+1}$, i.e., existe al menos una transición maybe sobre alguna letra de w. Para toda $\gamma \in \{r,p\}$, escribimos $M \overset{\epsilon}{\Longrightarrow}_\gamma M'$ para denotar $M(\xrightarrow{\tau}_\gamma)^* M'$, y $M \overset{\epsilon}{\Longrightarrow}_m M'$ para denotar $M(\overset{\epsilon}{\Longrightarrow}_p)(\xrightarrow{\tau}_m)(\overset{\epsilon}{\Longrightarrow}_p)M'$, i.e., existe al menos una transición maybe sobre τ. Para toda $\ell \neq \tau$ y $\gamma \in \{r,p\}$, escribimos $M \overset{\ell}{\Longrightarrow}_\gamma M'$ para denotar $M(\overset{\epsilon}{\Longrightarrow}_\gamma)(\xrightarrow{\ell}_\gamma)(\overset{\epsilon}{\Longrightarrow}_\gamma)M'$, y $M \overset{\ell}{\Longrightarrow}_m M'$ para $M(\overset{\epsilon}{\Longrightarrow}_m)(\xrightarrow{\ell}_p)(\overset{\epsilon}{\Longrightarrow}_p)M'$ o $M(\overset{\epsilon}{\Longrightarrow}_p)(\xrightarrow{\ell}_m)(\overset{\epsilon}{\Longrightarrow}_p)M'$, i.e., la transición maybe precede a una etiquetada o ella esta etiquetada con ℓ en el camino de M a M'. Para $\gamma \in \{r,m,p\}$, extendemos $\Longrightarrow_\gamma$ a palabras en el mismo sentido que para $\longrightarrow_\gamma$. Para toda $\ell \in Act_\tau$, sea $\hat{\ell} = \ell$ si $\ell \neq \tau$ y $\hat{\ell} = \epsilon$ si $\ell = \tau$. Para toda $\gamma \in \{r,m,p\}$ y $\ell \in Act_\tau$, normalmente escribimos $s \xrightarrow{\ell}_\gamma s'$ para decir $M_s \xrightarrow{\ell}_\gamma M_{s'}$ y análogamente para $\Longrightarrow_\gamma$. Las transiciones con flecha simple $\longrightarrow_\gamma$ son llamadas transiciones *simples*, y las transiciones con flecha "gruesa" $\Longrightarrow_\gamma$ son llamadas transiciones *observables*.

Para comparar un modelo con otro con un alfabeto mayor, debemos ocultar las acciones adicionales del segundo modelo y luego trabajar con *refinamiento (débil) observacional*, es decir, utilizaremos una relación de refinamiento que ignore las diferencias en la cantidad exacta de transiciones τ.

Definición 10 (Refinamiento Observacional) *N es un refinamiento observacional de M, denotado $M \preceq_o N$, si $\alpha M = \alpha N$ y (M,N) es contenido en alguna relación de refinamiento $R \subseteq \wp \times \wp$, para la cual vale para todo $\ell \in Act_\tau$ y todo $(M,N) \in R$:*

1. $(M \xrightarrow{\ell}_r M') \Rightarrow (\exists N' \cdot N \overset{\hat{\ell}}{\Longrightarrow}_r N' \ \wedge\ (M',N') \in R)$
2. $(N \xrightarrow{\ell}_p N') \Rightarrow (\exists M' \cdot M \overset{\hat{\ell}}{\Longrightarrow}_p M' \ \wedge\ (M',N') \in R)$

Tomando el modelo $\mathcal{F}$ (Fig. 2.10) ocultar las acciones `getWriteLock` y `relWriteLock` resulta en un MTS igual pero con las etiquetas `getWriteLock`? y `relWriteLock`? reemplazadas por τ?. Mas aún, el resultado es observacionalmente refinado por el MTS, $\mathcal{D}$:

$$\mathcal{A}' = \mathcal{F} \setminus \{\texttt{getWriteLock}, \texttt{relWriteLock}\} \preceq_{\mathrm{O}} \mathcal{A}$$

donde la relación de refinamiento es:

$$R = \{(\mathcal{F}'_0, \mathcal{A}_0), (\mathcal{F}'_1, \mathcal{A}_1)$$

De hecho, $\mathcal{F}'$ es también un refinamiento de $\mathcal{A}$ vía la inversa de R. Decimos que $\mathcal{F}'$ y $\mathcal{A}$ son *bisimilares observacionalmente*, denotado $\mathcal{F}' \sim_o \mathcal{A}$.

En el trabajo de Larsen and Thomsen [24] se definió operador de composición en paralelo sobre MTSs, en un intento por mostrar el comportamiento conjunto de dos modelos de diferentes sistemas.

$$\text{TD}\ \frac{M \xrightarrow{\ell}_r M'}{M\|N \xrightarrow{\ell}_r M'\|N}\ \ell \notin \alpha N \qquad \text{MT}\ \frac{M \xrightarrow{\ell}_m M',\ N \xrightarrow{\ell}_r N'}{M\|N \xrightarrow{\ell}_m M'\|N'}\ \ell \neq \tau \qquad \text{MD}\ \frac{M \xrightarrow{\ell}_m M'}{M\|N \xrightarrow{\ell}_m M'\|N}\ \ell \notin \alpha N$$

$$\text{TT}\ \frac{M \xrightarrow{\ell}_r M',\ N \xrightarrow{\ell}_r N'}{M\|N \xrightarrow{\ell}_r M'\|N'}\ \ell \neq \tau \qquad \text{MM}\ \frac{M \xrightarrow{\ell}_m M',\ N \xrightarrow{\ell}_m N'}{M\|N \xrightarrow{\ell}_m M'\|N'}\ \ell \neq \tau$$

Figura 2.13: Reglas para la composición en paralelo.

Definición 11 (Composición en paralelo) *Sea* $M = (S_M,\ L_M,\ \Delta^r_M,\ \Delta^p_M,\ s_{0M})$ *y* $N = (S_N,\ L_N,\ \Delta^r_N,\ \Delta^p_N,\ s_{0N})$ *dos MTSs. La* Composición en paralelo *(*$\|$*) es un operador simétrico tal que* $M\|P$ *es un MTS* $(S_M \times S_N,\ L_M \cup L_N,\ \Delta^r,\ \Delta^p,\ (s_{0M}, s_{0N}))$*, donde* Δ^r *y* Δ^p *son las relaciones mas pequeñas que satisfacen las reglas de la figura 2.13.*

Notemos que en las reglas en la figura 2.13, "T" significa "true", "M" significa "maybe", y "D" significa "don't care". En particular, la regla TT captura el caso en el cual ambos modelos tienen una transición true (requerida), la regla MT captura el caso en el cual un modelo tiene una transición maybe y el otro una requerida, y la regla TD captura el caso en el cual hay una transición requerida en un modelo sobre una acción no compartida por ellos (i.e. una acción que no esta en el alfabeto de uno de los modelos).

Proposition 1 (Propiedades de $\|$ [24]) *La composición en paralelo satisface las siguientes propiedades:*

1. *(Conmutatividad)* $M\|N = N\|M$.
2. *(Asociatividad)* $(M\|N)\|P = M\|(N\|P)$.
3. *(Monotonía)* $M \preceq_o N \Rightarrow M\|P \preceq_o N\|P$.

Definición 12 (Trazas) *Sea* $M = (S, L, \Delta^r_M, \Delta^p_M, s_0)$*. Decimos que una traza* $\pi = a_0, a_1, \ldots,$ *donde* $a_i \in Act$ *es una traza true en* M *si existe una secuencia infinita* M_i *tal que* $M = M_0$ *y* $M_i \xrightarrow{a_i}_r M_{i+1} \forall i \in \mathbb{N}$*. Una traza* π *es una traza maybe en* M *si* π *no es una traza true, pero existe una secuencia infinita* M_i *tal que* $M = M_0$ *y* $M_i \xrightarrow{a_i}_p M_{i+1} \forall i \in \mathbb{N}$*. Una traza* π *es una traza posible en* M *si es una traza maybe o true en* M*. Finalmente, una traza* π *es una traza false en* M *si no es una traza posible en* M*.*

Denotaremos al conjunto de trazas true, maybe, posibles y false sobre un MTS M por $ReqTr(M)$, $MaybeTr(M)$, $PosTr(M)$ y $FalseTr(M)$, respectivamente. Para un LTS $L = (S, Act, \Delta, s_0)$ denotamos con $Tr(M)$ el conjunto de las trazas true del MTS que lo contiene, i.e. $ReqTr(L = (S, Act, \Delta, \Delta, s_0))$

Definición 13 (Trazas infinitas) *Un MTS* $M = (S, L, \Delta^r_M, \Delta^p_M, s_0)$ *es un MTS de trazas infinitas si para todo* $s \in S$*, existe* $a \in L$ *y un* $s' \in S$ *tales que* $M_s \xrightarrow{a}_p M_{s'}$

En nuestro trabajo salvo que se aclare lo contrario siempre que prediquemos sobre trazas, estas serán trazas infinitas. Un MTS M es un MTS de trazas infinitas si todo es-

tado tiene al menos una transición saliente. Debido a que nosotros nos concentraremos en el chequeo de propiedades escritas en Fluent Linear Temporal Logic (LTL) sobre MTS nos concentraremos en los modelos de trazas infinitas, por lo tanto salvo que se aclare, al escribir MTS estaremos queriendo decir MTS de trazas infinitas y lo mismo para el caso de los LTS.

Dado un par de descripciones de comportamiento parcial, la operación de Merging de MTSs [30] se basa en combinar aquel comportamiento que es conocido de cada una, en otras palabras, es la construcción de un MTS que incluye todo el comportamiento que es requerido y prohibido de cada MTS y que es el menos refinado posible. Formalmente el merging de MTSs es el proceso de encontrar su refinamiento común mínimo (minimal common refinement).

Definición 14 (Refinamiento común mínimo) *Sean Q, M, y N MTSs. Q es un* refinamiento común (CR) *de M y N si $\alpha Q = (\alpha M \cup \alpha N)$, $M @ \alpha Q \preceq Q$ y $N @ \alpha Q \preceq Q$. Q es un* refinamiento común mínimo *(MCR) de M y N si Q es un CR de M y N y no hay un MTS $Q' \not\equiv Q$ tal que Q' es un CR de M y N, y $Q' \preceq Q$.*

Dados dos MTS M y N, un MCR para ellos puede no existir, en cuyo caso se dice que M y N son inconsistentes, o puede que no sea único [30]. En [7], se provee un algoritmo que construye un CR a partir de MTSs consistentes. El operador para dicho algoritmo es $+$. Se ha demostrado que este algoritmo produce un MCR en el caso que M y N tengan el mismo alfabeto, pero una prueba de que también lo hace cuando hay alfabetos distintos aún no ha sido provista.

Fluents y FLTL

Los fluents se utilizan para razonar acerca de los efectos de acciones sobre estados de un sistema.

Definición 15 *Un fluent* Fl *se define por un par de conjuntos I_{Fl}, el conjunto de acciones iniciadoras, y T_{Fl}, el conjunto de acciones finalizadoras:* $\mathrm{Fl} = \langle I_{\mathrm{Fl}}, T_{\mathrm{Fl}} \rangle$ *donde* $I_{\mathrm{Fl}}, T_{\mathrm{Fl}} \subseteq \mathrm{Act}$ *y* $I_{\mathrm{Fl}} \cap T_{\mathrm{Fl}} = \emptyset$.

Un fluent puede ser inicialmente verdaderao falsaindicado por el atributo $Initially_{Fl}$ y se asume falsasi no se incluye el atributo. Cada acción $a \in Act$ induce un fluent; a saber: $a = \langle a, Act \setminus \{a\} \rangle$.

FLTL [10] es una lógica para razonar sobre fluents.

Definición 16 *Dado el conjunto de fluents Φ,* $\mathrm{Fl} \in \Phi$ *es una fórmula FLTL, y otras fórmulas FLTL se definen inductivamente usando los conectivos booleanos conocidos y los operadores temporales* **X** *(next),* **U** *(strong until),* **W** *(weak until),* **F** *(eventually), and* **G** *(always).*

$$\pi \models Fl \triangleq \pi^0 \models Fl \qquad \pi \models \neg\phi \triangleq \neg(\pi \models \phi)$$
$$\pi \models \mathbf{X}\phi \triangleq \pi^1 \models \phi \qquad \pi \models \mathbf{G}\phi \triangleq \forall i \geq 0 \cdot \pi^i \models \phi$$
$$\pi \models \phi \wedge \psi \triangleq (\pi \models \phi) \wedge (\pi \models \psi)$$

Figura 2.14: Semántica del operador satisface

Sea Π el conjunto de trazas infinitas sobre *Act*. Para $\pi = a_0, a_1, \ldots \in \Pi$, se nota π^i al sufijo de π comenzando en a_i.

Definición 17 *$\pi = a_0, a_1, \ldots$ satisface un fluent* Fl*, notado $\pi \models$ Fl, si y solo si vale alguna de las siguientes condiciones:*

- $\text{Initially}_{\text{Fl}} \wedge (\forall j \in \mathbb{N} \cdot 0 \leq j \leq i \Rightarrow a_j \notin T_{\text{Fl}})$
- $\exists j \in \mathbb{N} \cdot (j \leq i \wedge a_j \in I_f) \wedge (\forall k \in \mathbb{N} \cdot j < k \leq i \Rightarrow a_k \notin T_{\text{Fl}})$

En otras palabras, un fluent vale en un cierto momento si y solo si vale inicialmente, u ocurrió alguna acción iniciadora, y en ambos casos, no ha sucedido una acción finalizadora. La Figura 2.14 muestra el operador de satisfacción $\models$ para algunos operadores de FLTL [10].

La semántica trivaluada de FLTL sobre un MTS M da el valor de cada fórmula $\phi \in$ FLTL en M. La semántica se da en función de las trazas de un MTS.

Definición 18 (Trazas de un MTS) *Sea M un MTS, una traza $\pi = a_0, a_1, \ldots$, donde $a_i \in Act$ es una traza verdaderaen M si hay una secuencia infinita $\{M_i\}$ tal que $M_0 = M$ y $M_i \xrightarrow{a_i}_r M_{i+1}$ para todo $i \in \mathbb{N}$.*
Una traza π es una traza maybeen M si no es una traza verdadera, pero existe una secuencia infinita $\{M_i\}$ tal que $M_0 = M$ y $M_i \xrightarrow{a_i}_p M_{i+1}$ para todo $i \in \mathbb{N}$.
Una traza π es una traza possible *en M si es una maybeo verdadera. Finalmente, una traza π es una traza falsaen M si no es una traza* possible*.*

Se nota el conjunto de trazas verdadera, maybe, *possible*, y falsasobre un MTS M como TrueTr(M), MaybeTr(M), PosTr(M) y FalseTr(M) respectivamente.

Entonces, en base a las trazas del MTS se define cuando una fórmula FLTL satisface al modelo.

Definición 19 *Sea ϕ una fórmula FLTL y M un MTS, ϕ es verdaderaen M (notado $M \models \phi$) si cada traza en* PosTr*(M) satisface ϕ, y falsaen M (notado $M \not\models \phi$) si hay una traza en* TrueTr*(M) que refuta ϕ, o si cada traza en* PosTr*(M) refuta ϕ. Sino, ϕ evalúa como maybe en M.*

La propiedad más importante es que el refinamiento preserva las propiedades verdaderay falsa:

Propiedad 1 (Preservación de FLTL) *Sea M un MTS. Entonces $\forall \phi \in FLTL$, $M \models \phi \Rightarrow \forall I \in \mathcal{I}[M] \cdot I \models \phi$ y $M \not\models \phi \Rightarrow \forall I \in \mathcal{I}[M] \cdot I \not\models \phi$*

Entonces, si una propiedad se evalúa como verdaderaen M, es *true* en todas las implementaciones de M, y si una propiedad se evalúa como *false* en M, es *false* en todas sus implementaciones.

En [29], se detalla un procedimiento $M(x)$ para construir automáticamente un MTS a partir de propiedades FLTL de *safety* (i.e. a formulas que pueden siempre ser refutadas de manera finita). Este procedimiento asegura que $M(\phi)$ caracteriza a todos los modelos LTS que satisfacen la propiedad ϕ.

Propiedad 2 (Caracterización de ϕ [29]) *Si ϕ es una propiedad de* safety *satisfacible, entonces para toda implementación LTS de trazas infinitas L, $L \models \phi \Leftrightarrow L \in \mathcal{I}[M(\phi)]$.*

Capítulo 3

Contribuciones

En esta sección analizaremos el desarrollo teórico en el cual trabajamos a lo largo de esta tesis, para cada tema detallaremos las alternativas que analizamos y las decisiones que tomamos en cada caso.

3.1. Equivalencia de MTS

En esta sección describiremos dos posibles alternativas para la noción de equivalencia entre MTS. En primer lugar analizaremos la bisimulación y en segundo lugar trabajaremos sobre la equivalencia por simulación, luego mostraremos porqué creemos que esta última es la relación más adecuada para ser la equivalencia entre MTS.

Definición 20 (Bisimulación) *Sea $\wp$ el universo de todos los MTSs y M y N MTSs. Decimos que M y N son bisimilares ($M \sim N$) si $\alpha M = \alpha N$ y existe R relación de refinamiento $R \subseteq \wp \times \wp$ tal que:*

$$M \preceq_R N \wedge \ M \preceq_{R^{-1}} N$$

Intuitivamente dos modelos M y N son bisimilares si existe una relación de refinamiento R tal que N refina a M por R y M refina a N por R^{-1}. La relacion de bisimulación de MTS es similar a la de bisimulación de LTS pero teniendo en cuenta que en MTS la relación de simulación está dada por la noción de refinamiento.

Definición 21 (Equivalencia por Simulación) *Sea $\wp$ el universo de todos los MTSs y M y N MTSs. Decimos que M y N equivalentes por simulación ($M \lesseqgtr N$) si $\alpha M = \alpha N$ y existen R y T relaciones de refinamiento $R, T \subseteq \wp \times \wp$ tales que:*

$$M \preceq_R N \wedge \ M \preceq_T N$$

Tomamos como equivalencia entre MTS a la *equivalencia por simulación* basándonos principalmente en dos motivos: En primer lugar, la simulación [26] en el mundo de los

MTS está representada por la relación de refinamiento, es decir, dados M y N MTS, M es simulado por N siempre y cuando exista una relación de refinamiento R entre ellos; entonces N simula a M si su conjunto de implementaciones está incluido o es igual al de M. En segundo lugar, esperamos que la equivalencia entre MTSs capture la noción de que dos MTS son equivalentes si tienen el mismo conjunto de implementaciones. A pesar de que estos puntos son satisfechos tanto por la bisimulación como por la *equivalencia por simulación* decidimos tomar la segunda por dos razones: por un lado la *equivalencia por simulación* es la más débil de ambas, es decir, para que dos modelos sean equivalentes por simulación alcanza con que puedan simularse mutuamente independientemente de si la inversa de alguna de esas relaciones también es una simulación entre ellos. Por otro lado, la *equivalencia por simulación* preserva Fluent Linear Temporal Logic (FLTL), en otras palabras si dos modelos son *equivalentes por simulación* entonces satisfacen las mismas propiedades LTS.

Si bien como mostramos hasta aquí la equivalencia por simulación es una relación de equivalencia menos restrictiva que la bisimulación para el caso de los LTS en el caso de los MTS no pudimos encontrar un par de modelos para los cuales fuera cierto que eran equivalentes por simulación pero no bisimilares. Luego de analizar la relación de refinamiento y viendo que tiene cierto "sabor" a bisimulación, creemos fuertemente que para el caso de los MTS vale que si existe una relación de equivalencia por simulación entre dos modelos entonces existe una relación de simulación entre ellos tal que su inversa también lo es (i.e son bisimilares). Mostramos esta noción más formalmente en la conjetura 1, no tenemos una demostración para ella, pero tampoco encontramos un contraejemplo.

Conjetura 1 *Sean M, N MTS, vale que, existen R, T relaciones de refinamiento entre M y N, y*

$$(M \preceq_R N \wedge M \preceq_T N) \Leftrightarrow (N \preceq_{R^{-1}} M \wedge N \preceq_{T^{-1}} M)$$

De aquí en adelante tomaremos como definición de equivalencia entre MTSs, la *equivalencia por simulación.*

3.2. Minimización

En esta sección definiremos la noción de minimización de MTS, plantearemos un algoritmo de minimización para MTS a partir de uno para LTS. Además mostraremos que algunas de las heurísicas aplicables a minimización de LTS no lo son en el contexto de MTS.

El objetivo de la minimización es el de generar a partir de un MTS $M = (S, L, \Delta^r, \Delta^p, s)$, otro $M' = (S', L', \Delta'^r, \Delta'^p, s')$ tal que M' es *equivalente* a M y su cantidad de estados es la mínima posible. Al construir modelos en forma composicional puede suceder que la combinación de estados entre los modelos compuestos, dé como resultado modelos con demasiados estados de forma tal que sea impracticable o como mínimo muy dificultoso cualquier análisis sobre ellos, a este problema se lo conoce como *explosión de*

estados. Para tratar de minimizar el impacto del crecimiento exponencial de estados pueden utilizarse diversas técnicas, entre ellas la reducción de ordenes parciales [31] y la minimización [15, 2].

En nuestro trabajo implementamos un algoritmo de minimización basándonos en una combinación de los propuestos en [15] y [2]. A alto nivel la idea del algoritmo es la de construir el modelo M' de forma tal que para cada estado s en S existe uno s' en S' que representa al conjunto de estados equivalentes a s ($\forall s \in S\ \exists s' \in S' \mid s' = [s]$). A este modelo lo llamaremos modelo *cociente*, este nombre proviene de forma en la que construimos sus estados como la partición de los estados de M en clases de equivalencia. La estructura a alto nivel del algoritmo que implementamos está descrita en el algoritmo 1.

El primer paso del algoritmo de minimización es la partición de los estados en clases de equivalencia, dado que la intención de dicho algoritmo es la de reducir los requerimientos de espacio. Es importante reducir el espacio ocupado por el propio algoritmo, por esto nuestra implementación no calcula a priori, la relación de equivalencia basada en los estados de la estructura original. Esto nos provee un mejor orden espacial pero empeora el orden de complejidad temporal.

A continuación detallaremos algunas variantes y problemáticas estudiadas respecto de las dos semánticas con las que trabajamos: la bisimulación y la equivalencia por simulación.

Al trabajar en el mundo de los LTS, la partición de los estados en clases de equivalencia tomando simulación puede no ser la más pequeña. Para resolver este problema en [2] se propone la eliminación de los *hermanos menores*. A continuación detallaremos esta técnica y luego veremos porque no es aplicable en el contexto de los MTS.

Definición 22 (Hermanos Menores) *Sea $M = (S, L, \Delta^r, \Delta^p, s)$ un MTS, H una relación de refinamiento maximal y $s_1, s_2 \in S$ decimos que s_1 es un* hermano menor *de s_2 si y sólo si vale simultáneamente lo siguiente:*

$$\exists s_3 \in S \text{ tal que } \forall \ell \in L\ (s_3 \xrightarrow{\ell}_r s_1 \Leftrightarrow s_3 \xrightarrow{\ell}_r s_2) \wedge\ (s_1 \preceq_H s_2 \wedge\ s_2 \not\preceq_H s_1)$$

También diremos que s_2 es un *hermano mayor* de s_1.

Intuitivamente, si s_1 es un hermano menor de s_2 significa que a partir de s_2 pueden realizarse todas las acciones posibles desde s_1, por lo tanto es posible eliminar la transición desde s_3 sin alterar el comportamiento aceptado por el modelo y dando la posibilidad de que el modelo final tenga menos cantidad de estados. En la figura 3.1 mostramos el modelo $\mathcal{G}$ en el cual, el estado $\mathcal{X}_5$ es hermano menor del estado $\mathcal{X}_1$ y por lo tanto puede eliminarse la transición $\mathcal{X}_0 \xrightarrow{a} \mathcal{X}_5$ y los estados no alcanzables obteniendo el modelo $\mathcal{H}$ (figura 3.2) equivalente por simulación al original.

En [2] se propone la eliminación de los *hermanos menores* para acercarse la cantidad mínima de estados.

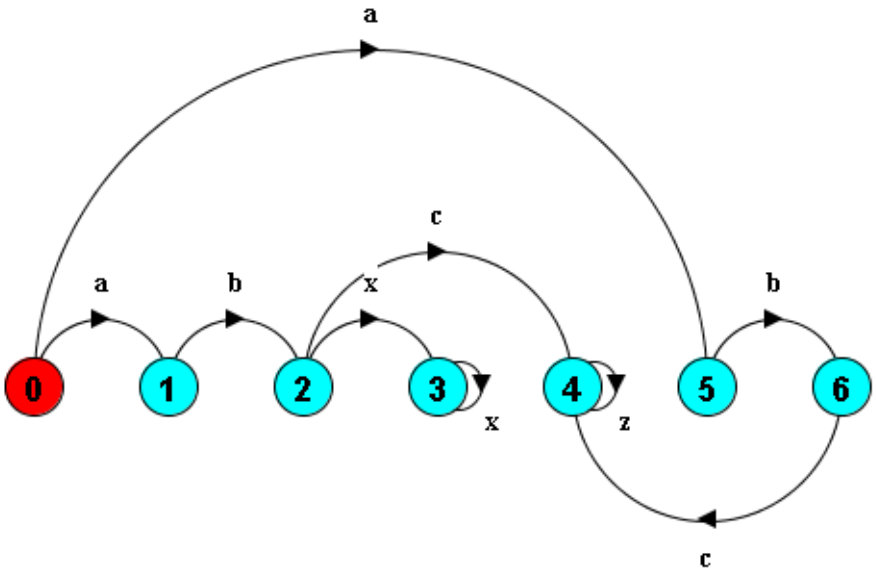

Figura 3.1: Modelo $\mathcal{X}$

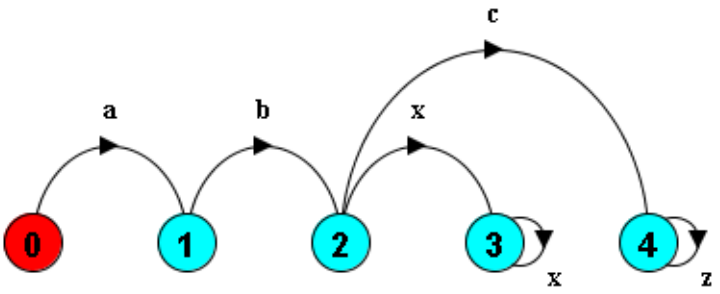

Figura 3.2: Modelo $\mathcal{H}$

En el dominio de los MTS, eliminar un *hermano menor* significa si se tienen dos estados s y s' con un mismo padre (antecesor) tal que se refinan en una única dirección (i.e $s \preceq s' \ \wedge \ s' \not\preceq s$) se elimina la transición al más refinado de ellos (i.e s). El modelo resultante en este caso admite un conjunto de implementaciones que es más pequeño que el del modelo original. En el modelo $\mathcal{J}$ (figura 3.3) el estado 1 es un *hermano menor* del estado 4, luego en la figura 3.4 observamos el modelo resultante de la eliminación de la transición desde 0 hacia 1 ($0 \xrightarrow{b}_r 1$), el *hermano menor* de 4. En esta transformación del modelo es simple ver que el conjunto de implementaciones del modelo de la figura 3.4 es más pequeño que el conjunto de implementaciones del modelo de la figura 3.3, en particular éste ultimo admite la implementación que muestra la figura 3.6 que el primero no. Por este motivo una modificación al algoritmo que en principio nos parecía interesante era la de eliminar las transiciones a los *hermanos mayores* y de esta forma, preservaríamos las implementaciones. Esta idea si bien parece razonable, plantea un problema similar al anterior. El modelo original admite comportamiento requerido que en el modelo sin el hermano mayor (figura 3.5) es descrito como posible (por definición

de hermanos menores) y por lo tanto éstos modelos no pueden ser equivalentes por simulación.

En función del análisis realizado y viendo que para MTSs no es cierto que eliminando los *hermanos menores/mayores* se obtiene un modelo equivalente por simulación, *no* implementamos en nuestro algoritmo la eliminación de *hermanos menores/mayores*.

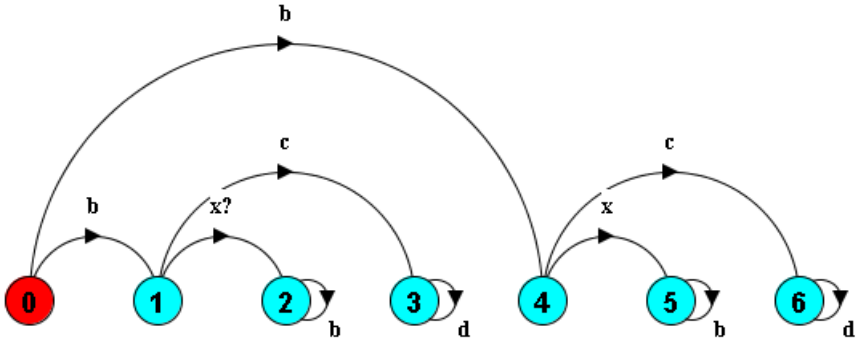

Figura 3.3: Modelo $\mathcal{J}$

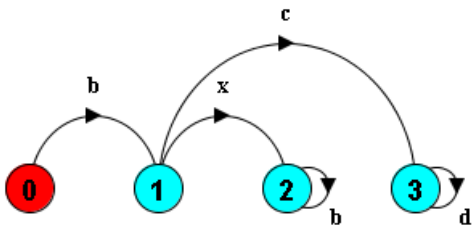

Figura 3.4: Modelo $\mathcal{J}$ sin Hermanos Menores

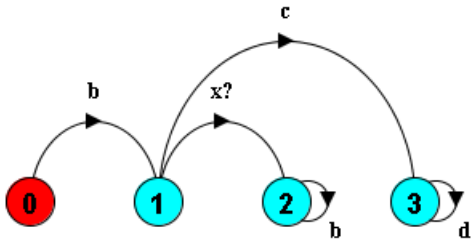

Figura 3.5: Modelo $\mathcal{J}$ sin Hermanos Mayores

A pesar de haber visto que la minimización basada en *equivalencia por simulación* tiene mayor orden de complejidad que la basada en *Bisimulación* [23], nuestro algoritmo está basado en la primera debido a que parece ser la mejor noción de equivalencia entre MTS, dado que dos MTS son equivalentes si tienen el mismo conjunto de implementaciones.

Para clarificar la legibilidad del algoritmo definimos $[s]^i$ para notar la clase de equivalencia del estado s en Σ_i. También definimos la función Π que asocia a cada

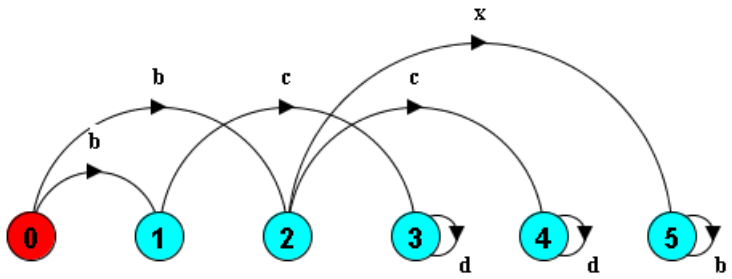

Figura 3.6: Implementación del modelo $\mathcal{J}$

clase $\alpha \in \Sigma_i$ con el conjunto de clases $\alpha' \in \Sigma_{i-1}$ que contienen al menos un sucesor de algún estado de α.

$$\Pi(\alpha) = \{\ [t]^{i-1} \mid \exists l \in L,\ \exists s \in \alpha,\ (s \xrightarrow{l}_p t)\ \in \Delta^p\}$$

A continuación daremos algunas definiciones preliminares:

Definición 23 (Refinamiento de orden i) *Sea $M = (S, L, \Delta^r, \Delta^p, s_0)$ un MTS y $s, t \in S$ se dice que $s \preceq_i t$ si y sólo si $i \geq 0$ y vale que:*

1. $(s \xrightarrow{\ell}_r s') \Rightarrow (\exists t' \cdot t \xrightarrow{\ell}_r t' \ \wedge\ (s', t') \in H_{i-1})$
2. $(t \xrightarrow{\ell}_p t') \Rightarrow (\exists s' \cdot s \xrightarrow{\ell}_p s' \ \wedge\ (s', t') \in H_{i-1})$

Definición 24 (i-equivalencia) *Sea $M = (S, L, \Delta^r, \Delta^p, s_0)$ un MTS y $s, t \in S$ se dice que $s \lesssim_i s'$ si y sólo si $i \geq 0 \wedge\ s \preceq_i t \wedge\ t \preceq_i s$*

Es importante notar que el algoritmo mantiene dos invariantes:

1. En cada paso i vale que para todo par de estados $s_1, s_2 \in S$, s_1 y s_2 están en la misma clase $\alpha \in \Sigma$ si y sólo si $s_1 \lesssim_i s_2$ (son i-equivalentes)
2. Para todo par de estados $s_1, s_2 \in S$, vale que $s_1 \preceq_i s_2 \Leftrightarrow ([s_1], [s_1]) \in H_{i-1}$

En el algoritmo 2 mostramos en pseudocódigo el algoritmo de partición.

El algoritmo esta dividido conceptualmente en dos procedimientos estrechamente relacionados: *refine* y *update*. En la primera se calculan las clases de equivalencia Σ_i (i.e los estados del modelo cociente) para cada iteración y en la segunda se actualiza la relación de simulación (i.e refinamiento) H_i.
El algoritmo toma como input un MTS M y comienza con una partición inicial Σ_0 (i.e. estados $0 - equivalentes$) que contiene un único elemento, el conjunto de todos los estados de M y además toma como relación de refinamiento inicial una H tal

que $H = \{\alpha \mid (\alpha, \alpha) \in \Sigma_0\}$. Luego, iterativamente, se van partiendo las Σ_i y se va construyendo la H final. Es decir, en la iteración $i+1$ se construye Σ_{i+1} generando nuevas particiones de Σ_i y H_{i+1} va siendo modificada en función de Σ_i, Σ_{i+1} y de H_i. Cuando el algoritmo termina (luego de k iteraciones) Σ_k es la partición en clases de equivalencia respecto de la simulación de los estados de M, estas forman los estados del modelo *cociente*. H_k representa la relación de simulación maximal entre los estados del modelo *cociente*.

Más detalladamente, Σ_i es el conjunto de clases de equivalencia según *i-equivalencia*. En la iteración i partimos las clases de Σ_{i-1} eligiendo un estado s_p, llamado *splitter* e identificando los estados *i-equivalentes* a s_p (línea11). Para armar las nuevas clases de equivalencia primero construimos dos conjuntos GT y LT que representan los conjuntos de estados cuyos elementos son refinamientos de s_p y los que son refinados por s_p respectivamente. La clase α' es entonces calculada como la intersección entre los conjuntos GT y LT ($\alpha' = LG \cap GT$, línea 14); y de esta forma α' está compuesta por los estados que son *i-equivalentes* a s_p.

La segunda parte del algoritmo de partición (update) se basa fuertemente en la definición 24. Sabemos que H_i representa la relación de simulación entre las clases de equivalencia y además sabemos que las relaciones de *i-refinamiento* $\preceq_1, \preceq_2, \ldots$ están incluidas de menor a mayor, es decir $\preceq_1 \supseteq \preceq_2 \supseteq \ldots$ [15], por lo tanto si $s \preceq_i t$ entonces $s \preceq_{i-1} t$, es decir si $([s],[t]) \in H_i$ implica que $([s],[t]) \in H_{i-1}$, basándonos en ésto es suficiente con chequear que $(\alpha'_1, \alpha'_2) \in H_i$ sólo si $\alpha_1 \supseteq \alpha'_1 \wedge \alpha_2 \supseteq \alpha'_2 \wedge (\alpha_1, \alpha_2) \in H_{i-1}$.

Para agregar el par (α'_1, α'_2) a H_i, primero calculamos el conjunto Φ de las clases que son menos refinadas que las clases de $\Pi(\alpha'_2)$, luego chequeamos las dos componentes de la guarda (línea 28), en primer lugar si $\Phi \supseteq \Pi(\alpha'_1)$ si toda clase en $\Pi(\alpha'_1)$ es menos refinada que alguna clase en $\Pi(\alpha'_2)$ y en segundo lugar si es cierto que $\alpha'_1 \preceq_{H_i} \alpha'_2$. En el caso en que estas dos condiciones sean verdaderas agregamos el par (α'_1, α'_2) a H_i. De esta forma cuando el algoritmo termina, H_i es la relación maximal sobre el modelo *cociente*.

El algoritmo tiene como condición de parada que no se hayan modificado ni Σ_i ni H_i en una misma iteración. Un cambio en Σ_i es el resultado de una partición de alguna clase $\alpha \in S$, por lo tanto el número de particiones posibles esta acotado por $|S|$. Así mismo un cambio en H_i implica que se encontró una $\preceq_{i+1}$ que estaba contenida $\preceq_i$ y es más pequeña en tamaño, por tanto el número de modificaciones de H está acotado por $|S^2|$ Podemos estar seguros que el algoritmo termina y que la cantidad de iteraciones esta acotada por $|S^2| + |S|$ iteraciones.

Algoritmo 1 (Minimización) *1: Calcular el conjunto de estados del* cociente
2: y la relación de simulación maximal H
3: Eliminar todos los estados inalcanzables

Algoritmo 2 *1:* ***procedure*** PARTITIÓN
2: *change = true*

```
3:    Σ_0 = S
4:    H_0 = { (α, α) | α ∈ Σ_0}
5:    while change == true do
6:        change = false
7:        refine :
8:        Σ_{i+1} = ∅
9:        for all α ∈ Σ_i do
10:           while α ≠ ∅ do
11:               Tomar s_p tal que s_p ∈ α
12:               GT = {s_g | s_g ∈ α ∧ [s_p] ⪯_{H_i} [s_g]}
13:               LT = {s_l | s_l ∈ α ∧ [s_l] ⪯_{H_i} [s_p]}
14:               α' = GT ∩ LT
15:               if α ≠ α' then
16:                   change = true
17:               end if
18:               α = α \ α'
19:               Σ_{i+1} = Σ_{i+1} ∪ {α'}
20:           end while
21:       end for
22:       Update :
23:       H_{i+1} = ∅
24:       for all (α_1, α_2) ∈ H_i do
25:           for all (α'_1, α'_2) ∈ Σ_{i+1} do
26:               if α_1 ⊇ α'_1 ∧ α_2 ⊇ α'_2 then
27:                   Φ = {φ | ∃ξ ∈ Π(α'_2), (φ, ξ) ∈ H_i}
28:                   if Φ ⊇ Π(α'_1) ∧ α'_1 ⪯_{H_i} α'_2 then
29:                       H_{i+1} = H_{i+1} ∪ {(α'_1, α'_2)}
30:                   else
31:                       change = true
32:                   end if
33:               end if
34:           end for
35:       end for
36:   end while
37: end procedure
```

3.3. Determinización

En esta sección describiremos la operación de determinización sobre LTS. Luego plantearemos algunas alternativas para la determinización sobre MTSs: preservación de trazas requeridas y posibles, preservación de implementaciones y finalmente preservación de propiedades. Luego daremos un algoritmo para determinizar MTS según la que consideramos la opción más adecuada para la determinización, la preservación de propiedades.

3.3.1. Determinización LTS

Definición 25 (Determinización) *Sean L y L' LTS, se dice que L' es la versión determinística de L si vale que*

$$isDet_{LTS}(L') = true \ \wedge \ \text{Tr}(L) \equiv \ \text{Tr}(L')$$

.

Definición 26 (Determinización LTS) *Sea $L = (S, L, \Delta, s_0)$ un LTS, diremos que L es determinístico ($isDet_{LTS}(L) = true$) si y sólo no existen dos estados diferentes a los que se pueda llegar desde un estado s por transiciones sobre la misma etiqueta del alfabeto.*

Intuitivamente un LTS es determinístico si no posee ningún estado desde el cual estén habilitadas dos o más transiciones sobre la misma acción del alfabeto. Las figuras 3.7 y 3.8 muestran los modelos $\mathcal{K}$ y $\mathcal{M}$ respectivamente, de ellos se desprende que $Tr(\mathcal{K}) = Tr(\mathcal{M}) \wedge \ isDet_{LTS}(\mathcal{M}) = true$ y por lo tanto, $\mathcal{M}$ es la versión determinística de $\mathcal{K}$. Es importante notar que la determinización es una operación que preserva trazas [15] pero no garantiza preservación de la bisimulación. En otras palabras, el LTS resultante de la determinización tiene el mismo conjunto de trazas que el modelo original pero no necesariamente son bisimilares. En el ejemplo anterior mostramos que a pesar de que $\mathcal{M}$ es la versión determinística de $\mathcal{K}$, éstos modelos no son bisimilares (e.g el estado 1 de $\mathcal{M}$ no puede ser simulado por ningún estado en $\mathcal{K}$).

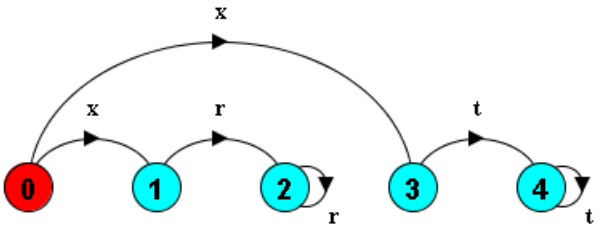

Figura 3.7: Modelo $\mathcal{K}$

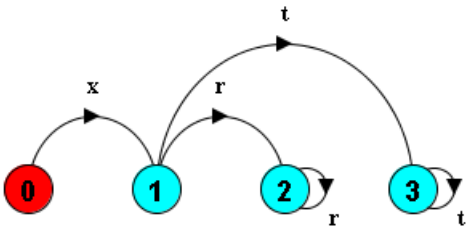

Figura 3.8: Modelo $\mathcal{M}$

3.3.2. Determinización MTS

En esta subsección describiremos las alternativas que analizamos para resolver la determinización de MTS. A lo largo de esta subsección al referirnos al conjunto de implementaciones de un MTS, salvo que se aclare lo contrario, nos estaremos refieriendo al conjunto minimal de implementaciones bajo equivalencia por bisimulación, es decir, el conjunto tal que no hay dos modelos dentro de él que sean equivalentes por simulación.

Para analizar la determinización de MTSs es importante tener en cuenta que los MTS son caracterizados por sus implementaciones; a partir de esta noción pueden desprenderse diferentes interpretaciones de las propiedades que la determinización de MTS debería cumplir. En nuestro trabajo analizamos tres posibles interpretaciones: *Preservación de Trazas Requeridas y Posibles*, *Preservación de Implementaciones* y *Preservación de Propiedades.*

Preservación de Trazas Requeridas y Posibles

Proposition 2 *Dado M MTS, se dice que M' es la versión determinística de M si valen las siguientes tres propiedades M' es determinístico ($isDet_{LTS}(M'^{+}) \equiv true$),* ReqTr*(M)*=ReqTr*(M')* *y* PosTr*(M)*=PosTr*(M').*

Proposition 3 *Given M an MTS, we say that M' is a deterministic version of M if the following three properties holds: M' is deterministic ($isDet_{LTS}(M') \equiv true$),* ReqTr*(M)*=ReqTr*(M')* *and* PosTr*(M)*=PosTr*(M').*

Proposition 4 *Given M, M' MTSs and I_M, $I_{M'}$ their implementation sets, we say M' is a deterministic version of M, and we note $M' = Det_{MTS}(M)$ if and only if $I_{M'} = DetC(I_M)$*

Proposition 5 *Given M, M' MTSs, and I_M, $I_{M'}$ their implementation sets, we state that M' is a deterministic version of M (notation $M' = Det_{MTS}(M)$) if and only if the following hold simultaneously:*

1. $DetC(I_M) \subseteq I_{M'}$
2. $\forall N\ MTS,\ M' \preceq N \Rightarrow DetC(I_M) \not\subseteq I_N$

Theorem 1 *Given MTSs M and M', if it is true that $M' = Det_{MTS}(M)$ then $\forall \phi\ FLTL$ holds the following properties,*

$$M \models \phi \ \equiv \ true \ \Leftrightarrow M' \models \phi \equiv true \tag{3.1}$$

$$M \models \phi \ \equiv \ false \ \Leftrightarrow M' \models \phi \equiv false \tag{3.2}$$

Intuitivamente determinizar modelos preservando transiciones requeridas y posibles significa que el comportamiento posible del modelo original es descrito por su versión determinística y tienen el mismo conjunto de trazas requeridas y posibles. En primer lugar esto significa que si el modelo original M admite una traza requerida ω ($\omega \in$ReqTr(M)) entonces ω también pertenecerá al conjunto de trazas requeridas de M' ($\omega \in$ ReqTr(M')). En segundo lugar si la traza ϕ pertenece a las trazas maybe de M ($\phi \in$ $MaybeTr(M)$) entonces es cierta alguna de las siguientes: 1) Si no existe una traza ω en las requeridas de M con la misma secuencia de etiquetas que ϕ entonces ϕ también pertenecerá al conjunto de trazas maybes de M'. 2) Si en cambio existe una traza requerida ω en M tal que está compuesta por la misma secuencia de etiquetas que ϕ entonces ω pertenecerá al conjunto de trazas requeridas de M' y por lo tanto también pertenecerá a las posibles de M'. En este último caso cabe notar que la traza maybe ϕ no formará parte de las trazas maybe de M'. En otras palabras, ella es eliminada como traza maybe en el modelo M'.

El modelo $\mathcal{O}$ (figura 3.10) es la versión determinística de $\mathcal{N}$ (figura 3.9). Desde su estado inicial $\mathcal{N}$ admite dos transiciones no determinísticas, $0 \xrightarrow{a}_m 1$ y $0 \xrightarrow{a}_r 4$. En el modelo $\mathcal{O}$ esas transiciones son transformadas en una única transición $0 \xrightarrow{a}_r 1$ con el fin de preservar las trazas requeridas. Por otro lado, la traza $a \rightarrow u \rightarrow c \rightarrow \ldots$ pertenece a las trazas maybe del modelo $\mathcal{N}$, por lo tanto con el fin de preservar las trazas posibles del modelo original (en el determinizado) debemos transformar las transiciones $1 \xrightarrow{u}_r 2$ y $2 \xrightarrow{c}_r 0$ de requeridas a maybes para preservar las trazas posibles del modelo, es por esto que el modelo final admite las transiciones $1 \xrightarrow{u}_m 2$ y $2 \xrightarrow{c}_m 0$.

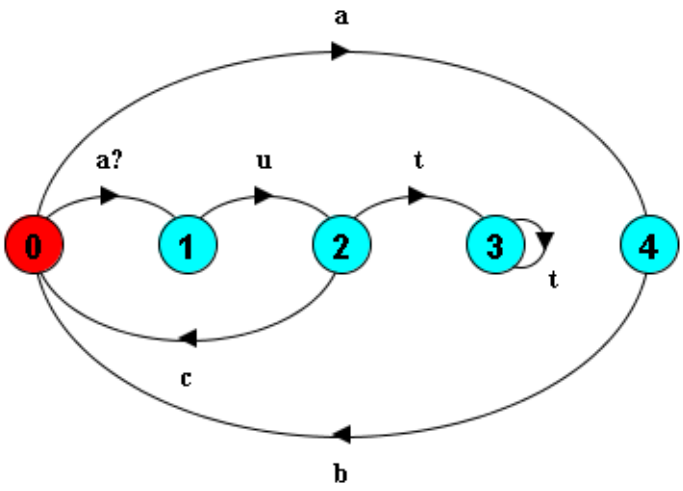

Figura 3.9: Modelo $\mathcal{N}$

Si bien ésta es, intuitivamente, una forma adecuada para la determinización, presenta algunos problemas en el mundo de los MTS. En la figura 3.11 vemos una implementación del modelo determinístico $\mathcal{O}$ que no es implementación de $\mathcal{N}$. Es decir, el conjunto de implementaciones de $\mathcal{N}$ es diferente al de su versión determinística $\mathcal{O}$. Este problema es originado por la propagación de transiciones maybe. En nuestro ejemplo, la transición $0 \xrightarrow{a}_m 1$ en $\mathcal{N}$, se determiniza junto con su par $0 \xrightarrow{a}_r 4$ pero

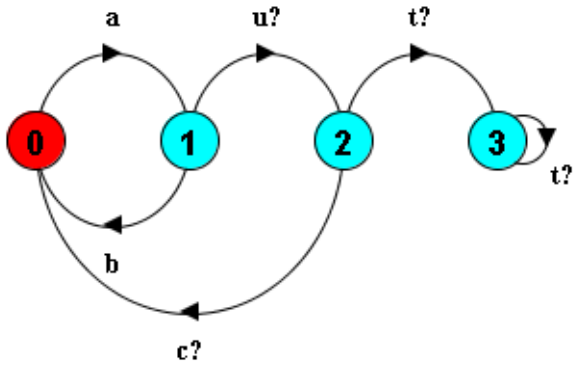

Figura 3.10: Modelo $\mathcal{O}$, versión determinística de $\mathcal{N}$

las transiciones siguientes a $0 \xrightarrow{a}_m 1$, se eliminan y se reemplazan por transiciones maybe sobre la misma etiqueta. La propagación de estas transiciones maybe hace que en el modelo $\mathcal{O}$ uno pueda, siguiendo las transiciones $0 \xrightarrow{a}_r 1$, $1 \xrightarrow{u}_m 2$ llegar al estado 2 que admite dos transiciones maybe salientes y por lo tanto permite obtener *implementaciones* que no son posibles para el modelo $\mathcal{N}$; por ejemplo, podríamos dar una implementación de $\mathcal{O}$ que la transición $2 \xrightarrow{c}_m 0$. Para el modelo $\mathcal{N}$ esto no es posible, debido a que las transiciones salientes desde el estado 2 son requeridas y por lo tanto, no hay posibilidad de dar una implementación que prohiba alguna de ellas.

Debido a estos problemas entendemos que preservar las trazas requeridas y posibles únicamente, no es una aproximación completa para la determinización de MTS y debido a esto evaluamos otras opciones para la determinización.

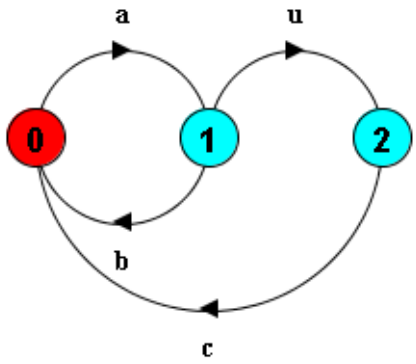

Figura 3.11: Implementación de $\mathcal{O}$

Preservación de implementaciones

Teniendo en cuenta que los MTS son caracterizados por su conjunto de implementaciones, un requerimiento deseado para la determinización es que preserve las implementaciones determinísticas del modelo a determinizar, en otras palabras, dados el modelo M no determinístico y M' su determinización, quisieramos que tomando

el conjunto de las implementaciones de M y aplicando determinización de LTS a cada uno de ellos, el conjunto resultante sea igual al conjunto de las implementaciones de M'. Dado un modelo M y su conjunto de implementaciones I_M, mostramos una intuición gráfica de lo que esperamos de la determinización por preservación de implementaciones en la figura 3.3.2 tomando .

Definición 27 *Dado I un conjunto de LTS, definimos $DetC(I)$ de la siguiente forma:*

$$DetC(I) = \{i' \in LTS \mid \exists i \in I,\ i' = Det_{LTS}(i)\}$$

Proposition 6 *Dados M, M' MTSs y I_M, $I_{M'}$ sus respectivos conjuntos de implementaciones, se dice que M' es la versión determinística de M, notado $M' = Det_{MTS}(M)$ si y sólo si $I_{M'} = DetC(I_M)$*

Proposition 7 *Dados M, M' MTSs y I_M, $I_{M'}$ sus respectivos conjuntos de implementaciones, se dice que M' es la versión determinística de M, notado $M' = Det_{MTS}(M)$ si y sólo si $I_{M'} = DetC(I_M)$*

$$\begin{array}{ccc} M & \overset{det_{MTS}}{\longrightarrow} & M' \\ \downarrow & & \uparrow \\ I & \overset{DetC}{\longrightarrow} & I' \end{array}$$

Analizando los problemas que encontramos con la *Preservación de Trazas Requeridas y Posibles* e intentando satisfacer la proposición 7 modificamos la operación de determinización. Entendemos que el problema en el caso anterior esta relacionado con la determinización de un conjunto de transiciones en el cual alguna es maybe, debido a que "propagamos" la marca maybe a todas sus sucesoras. Es por esto último que no preservamos el conjunto de implementaciones y por lo tanto nuestra modificación se basa en no propagar la marca de maybe hacia todas las siguientes, sino que lo hacemos únicamente para las transiciones "adyacentes" a ella. De forma se preservan las trazas requeridas y posibles y además evitamos la generación de nuevas implementaciones.

El modelo $\mathcal{P}$ (figura 3.12) es la versión determinística de $\mathcal{N}$ (figura 3.9) según preservación de implementaciones. El conjunto de implementaciones de $\mathcal{N}$ determinizadas es igual al conjunto de implementaciones de $\mathcal{P}$.

Analizando esta solución en detalle encontramos que tiene problemas en algunos casos. Según creemos, esto se debe fundamentalmente que la determinización altera la estructura de las ramas de los modelos. Básicamente la estructura de ramificaciones de las transiciones se modifica junto con el conjunto de trazas maybe y por lo tanto no parece razonable pensar que el conjunto de implementaciones se mantenga inalterado.

Un caso en el cual se hacen evidentes los problemas es el que mostramos en el modelo $\mathcal{S}$ (figura 3.14) el cual es la versión determinística $\mathcal{R}$ (figura 3.13). Toda implementación de $\mathcal{S}$ que tome la transición $0 \overset{a}{\longrightarrow}_m 1$ como $0 \overset{a}{\longrightarrow}_r 1$ requiere a continuación

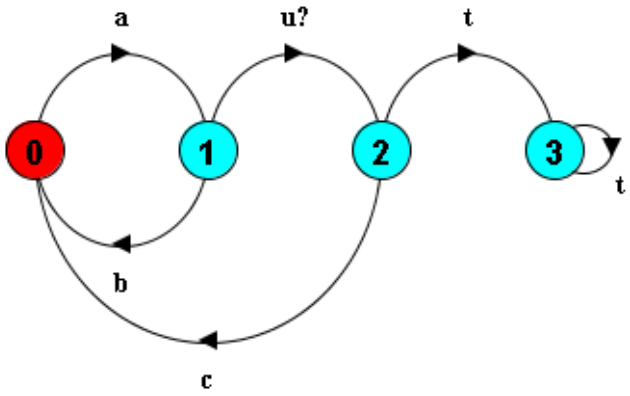

Figura 3.12: Modelo $\mathcal{P}$

las transiciones $1 \xrightarrow{c}_r 2$ y $1 \xrightarrow{t}_r 3$ necesariamente. El modelo $\mathcal{T}$ (figura 3.15)es una implementación de $\mathcal{S}$ y es simple ver que no es implementación de $\mathcal{R}$.

Luego de investigar esta solución no logramos una definición de determinización que cumpla con la proposición 7.

Preservación de Propiedades

Hasta aquí hemos analizado dos nociones de determinización: la *Preservación de Trazas Requeridas y Posibles* y la *Preservación de implementaciones* y ninguna de ellas satisfizo nuestras expectativas. Finalmente, en este caso presentamos una definición que según nuestro análisis parece ser la más apropiada para la determinización de MTS.

Proposition 8 *Dados M, M' MTSs. Llamaremos I_M y $I_{M'}$ sus respectivos conjuntos de implementaciones. Decimos que M' es la versión determinística de M, notado $M' = Det_{MTS}(M)$ si y sólo si valen simultaneamente las siguientes afirmaciones:*

1. $DetC(I_M) \subseteq I_{M'}$
2. $\forall N\ MTS,\ M' \preceq N \Rightarrow DetC(I_M) \not\subseteq I_N$

Intuitivamente, un modelo determinizado debería tener todas sus implementaciones determinísticas y además debería ser el más refinado posible tal que su conjunto de implementaciones incluya al conjunto formado por la determinización de las implementaciones del modelo original.

De esta proposición se desprende una propiedad muy interesante que tiene que ver con que si bien existe esta posible diferencia entre los conjuntos de implementaciones sucede que si una propiedad lineal evalúa a verdadero en el modelo original entonces también evalúa a verdadero en el modelo determinizado y lo mismo pasa si

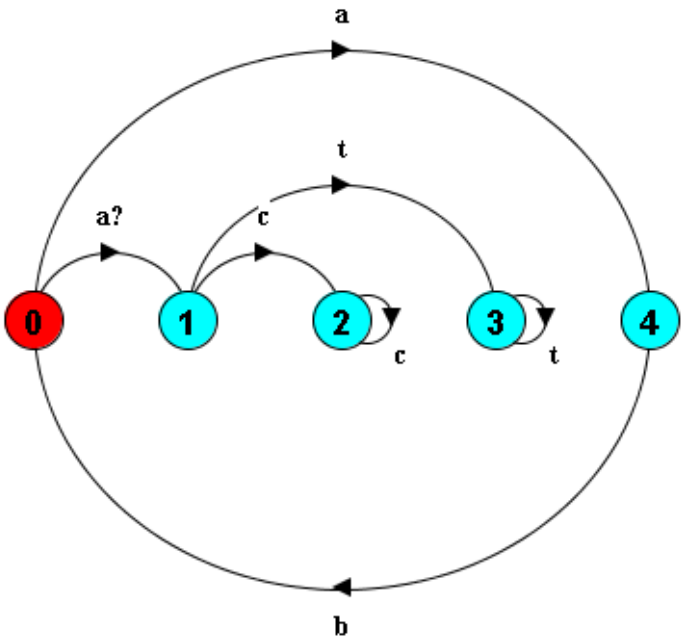

Figura 3.13: Modelo $\mathcal{R}$

la propiedad evaluara a falso. Esto se debe a que el model checking está definido en función de las versiones optimista y pesimista del modelo (definiciones 29 y 30), y al ser la determinización una operación que preserva trazas requeridas y posibles, sucede que las versiones optimista y pesimista del modelo original y del determinizado coinciden. En el caso de las propiedades que evalúan a maybe, no necesariamente se mantienen, debido justamente a que el conjunto de implementaciones del modelo determinizado es mayor que la determinización de las implementaciones del modelo original.

Theorem 2 *Dados M y M' MTSs, si $M' = Det_{MTS}(M)$ entonces $\forall \phi$ FLTL valen las siguientes propiedades,*

$$M \models \phi \equiv true \Leftrightarrow M' \models \phi \equiv true \quad (3.3)$$
$$M \models \phi \equiv false \Leftrightarrow M' \models \phi \equiv false \quad (3.4)$$

Demostración 1 *Por definición sabemos que,*

$M \models \phi \equiv true \Leftrightarrow M^+ \models \phi \Leftrightarrow \forall \pi \in Tr(M^+)\pi \models \phi$

Sabemos que la determinización preserva las trazas requeridas y posibles, entonces vale que,

PosTr(M)=PosTr(M'), *y* PosTr(M)=Tr(M^+) *y* PosTr(M')=Tr(M'^+), *entonces* Tr(M^+)=Tr(M'^+), *y* $\forall \pi \in$Tr$(M^+)\pi \models \phi \Leftrightarrow \forall \pi \in$Tr$(M'^+)\pi \models \phi$, *es decir que* $M \models \phi \equiv true \Leftrightarrow M' \models \phi \equiv true$.

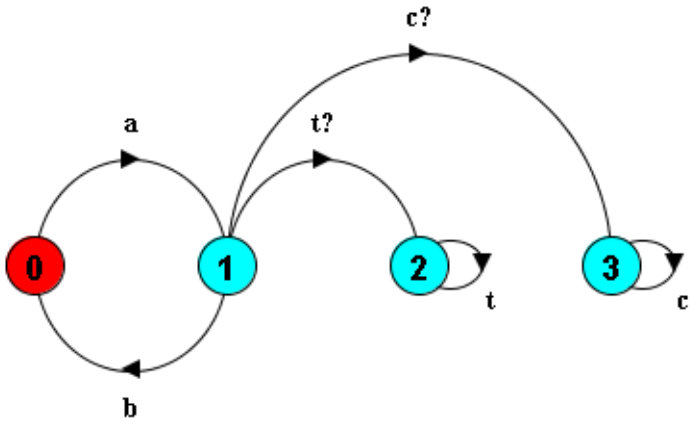

Figura 3.14: Modelo $\mathcal{S}$

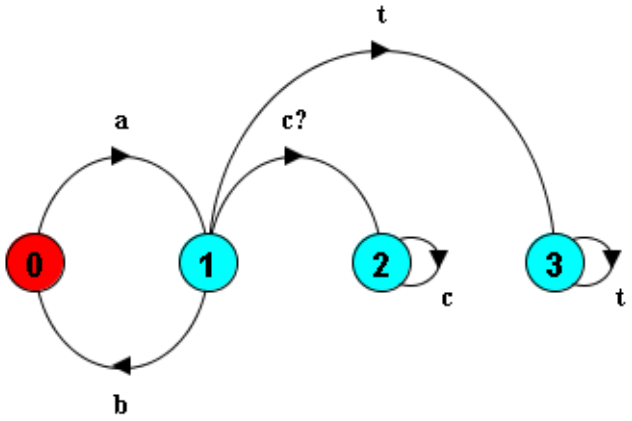

Figura 3.15: Modelo $\mathcal{T}$

Ahora veamos que, $M \models \phi \equiv false \Leftrightarrow M' \models \phi \equiv false$, $\text{ReqTr}(M)$=$\text{ReqTr}(M')$, *y*

Si M^- *es no vacio,* $M \models \phi \equiv false \Leftrightarrow M^- \not\models \phi \Leftrightarrow \exists \pi \in Tr(M^-)\pi \not\models \phi$, *sabemos que* $\text{ReqTr}(M)$=$\text{Tr}(M^-)$ *y* $\text{ReqTr}(M')$=$\text{Tr}(M'^-)$, *entonces* $\exists \pi \in \text{Tr}(M^-)\pi \not\models \phi \Leftrightarrow \exists \pi \in \text{Tr}(M'^-)\pi \not\models \phi$ *es decir que,*

Si M^- *es vacío, entonces* $M \models \phi \equiv false \Leftrightarrow M^+ \models \neg\phi \Leftrightarrow \forall \pi \in \text{Tr}(M'^+)\pi \models \neg\phi$, *pero como ya mostramos* $\text{Tr}(M^+)$=$\text{Tr}(M'^+)$ *y por lo tanto vale que* $\forall \pi \in \text{Tr}(M^+)\pi \models \neg\phi \Leftrightarrow \forall \pi \in \text{Tr}(M'^+)\pi \models \neg\phi$.

Por lo tanto $M \models \phi \equiv false \Leftrightarrow M' \models \phi \equiv false$.

□

Nuestra implementación del algoritmo de determinización corresponde al pseudo-código que mostramos a continuación:

Algoritmo 3

```
1: procedure DETERMINIZAR(MTS mtsEntrada)
2:     MTS mtsDeterminizado
3:     EstadoCompuesto estadoActual = multipleState(initialState(toDeterminize))
4:     int codigoEstadoActual = codificar(estadoActual)
5:     int codigoProximoEstado = initialState(mtsEntrada)
6:     while codigoEstadoActual < codigoProximoEstado do
7:         change=false
8:         for all accion en acciones(mtsEntrada) do
9:             if cantidad de transiciones por accion desde algun subestado de currentState then
10:                boolean esRequerida = falso
11:                EstadoCompuesto nuevoEstado
12:                for all subEstado en currentState do
13:                    for all transicion por accion desde subEstado do
14:                        agregarSubEstado(nuevoEstado, destino(transicion)
15:                        esRequerida = esRequerida or tipo(transicion)=requerida
16:                        esRequerida = esRequerida or !propagaMaybe(subEstado)
17:                        if tipo(transicion)=maybe y #transiciones(subEstado)> 1 then
18:                            marcarMaybe(estadoDestino(transicion))
19:                        end if
20:                    end for
21:                end for
22:                int codigoNuevoEstado = agregarEstado(nuevoEstado)
23:                agregarTransicion(mtsDeterminizado, codigoEstadoActual, accion, codigoNuevoEstado, esRequerida)
24:            end if
25:        end for
26:    end while
27: end procedure
```

Theorem 3 *El algoritmo 3 recibiendo un MTS M como entrada, da como resultado un MTS M' tal que es la versión determinística de M según la definición 8.*

Demostración 2 *En primer lugar probaremos que $DetC(I_M) \subseteq I_{M'}$ En esta parte de la demostración trabajaremos tomando el algoritmo por pasos. Mostraremos que la inclusión vale para un paso y luego veremos como esto es extensivo a los siguientes; por último veremos que el algoritmo termina.*

De aquí en adelante utilizaremos la notación I_k para nombrar a una implementación del modelo k (e.g I_M es una posible implementación de M).

Sea $M = (S, L, \Delta^r, \Delta^p, s_0)$ MTS y $\ell \in L$

Para cualquier M input del algoritmo tenemos los siguientes posibles casos:

1. ***Caso: La relación de refinamiento entre M e I_M no contiene estados***

modificados por $det1$
Si el algoritmo en un paso no modificó los estados entonces significa que M y M' son iguales, por lo tanto tienen las mismas implementaciones.

2. ***Caso: La relación de refinamiento R entre M e I_M contiene estados modificados por*** $det1$**,**
Sin perdida de generalidad podemos asumir que existen m_1, m_2, m_3 estados de M y al menos im_1, im_2 estados de I tales que M tiene la transición $m_1 \xrightarrow{\ell}_p m_2$, $m_1 \xrightarrow{\ell}_p m_3$ e I tiene la transición $im_1 \xrightarrow{\ell} im_2$, donde (m_1, i_{m_1}) y (m_2, i_{m_2}) están en R.

   a) ***Caso: No hay transiciones no determinísticas en I entre los estados modificados por*** $det1$

   Sea $M' = (S', L', \Delta'^r, \Delta'^p, s'_0)$ e I LTS, tal que $M' = Det_{MTS}(M)$, $M \preceq_R I$, y sean m_2 y m_3 estados de M modificados por det1 y i_{m_1} y i_{m_2} estados de I tales que $(m_1, i_{m_1}), (m_2, i_{m_2}) \in R$ y además $m_1 \xrightarrow{\ell}_r m_3$.
   Det1 transforma el estado m_1 en el estado $m'_1 = [m_1]$, el estado m_2 en $m'_2 = [m_2]$ y $\forall \ell \in L$ $(m_2 \xrightarrow{\ell}_k m_i \Leftrightarrow [m_2] \xrightarrow{\ell}_k [m_i]) \wedge (m_3 \xrightarrow{\ell}_k m_j \Leftrightarrow [m_2] \xrightarrow{\ell}_k [m_j])$, con $k \in \{r, m\}$ e i, j naturales.
   Entonces, por lo anterior, podemos tomar una implementación I' de M' ($M' \preceq_T I'$), i_{m_1}, i_{m_2} estados de I' tal que $(m'_1, i_{m_1}), (m'_2, i_{m_2}) \in T$ y usando que $m'_2 = [m_2]$ también vale que $(m'_2, [i_{m_i}]) \in T$ para todo i tal que $i_{m_2} \xrightarrow{\ell}_k i_{mi}$. Entonces $I' = Det_{LTS}(I)$.

   b) ***Caso: Desde i_{m_1} hay transiciones no determinísticas salientes sobre algún ℓ y ninguna es un ciclo.***

   Sea $M' = (S', L', \Delta'^r, \Delta'^p, s'_0)$ e I LTS, tal que $M' = Det_{MTS}(M)$, $M \preceq_R I$, y sean m_2 y m_3 estados de M modificados por det1 y i_{m_1}, i_{m_2} y i_{m_3} estados de I tales que $(m_1, i_{m_1}), (m_2, i_{m_2}), (m_3, i_{m_3}) \in R$ y ademas $m_1 \xrightarrow{\ell}_m m_2 \wedge m_1 \xrightarrow{\ell}_r m_3$.
   Det1 transforma el estado m_1 en el estado $m'_1 = [m_1]$, los estados m_2, m_3 en $m'_2 = [m_2, m_3]$ y $\forall \ell \in L$ $(m_2 \xrightarrow{\ell}_k m_i \Leftrightarrow [m_2] \xrightarrow{\ell}_k [m_i]) \wedge (m_3 \xrightarrow{\ell}_k m_j \Leftrightarrow [m_2] \xrightarrow{\ell}_k [m_j])$, con $k \in \{r, m\}$ e i, j naturales.
   Entonces por lo anterior, podemos tomar una implementación I' de M' ($M' \preceq_T I'$), i_{m_1}, i_{m_2} estados de I' tal que $(m'_1, i_{m_1}), (m'_2, i_{m_2}) \in T$ y usando que $m'_2 = [m_2] = [m_3] = [m_2, m_3]$ también vale que $(m'_2, [i_{m_i}]) \in T$ para todo i tal que $i_{m_2} \xrightarrow{\ell}_k i_{mi} \vee i_{m_3} \xrightarrow{\ell}_k i_{mi}$. Entonces $I' = Det_{LTS}(I)$.

   c) ***Caso: Desde i_{m_1} hay transiciones no determinísticas salientes sobre algún ℓ y una de las transiciones es un ciclo.***
   Este caso es análogo al anterior pero tomando $m_1 = m_2$ y $i_{m_1} = i_{m_2}$ y es simple ver que vale la misma implicación.

Hasta aquí mostramos cómo aplicando el algoritmo de determinización de MTS en un paso se modifica el modelo original de forma tal que la propiedad $DetC(I_M) \subseteq I_{M'}$ es válida.

Es simple ver que el algoritmo en cada paso cumple con lo demostrado hasta aquí, además dado que el algoritmo itera sobre los estados del modelo determinizado que están acotados por dos veces la cantidad de estados del modelo original. Por lo tanto el algoritmo termina y genera un modelo determínistico que satisface $DetC(I_M) \subseteq I_{M'}$.

En segundo lugar queremos probar que $\forall N\ MTS,\ M' \preceq N \Rightarrow DetC(I_M) \not\subseteq I_N$. Existen dos casos en los cuales el algoritmo modifica el modelo de forma tal de afectar su conjunto de implementaciones. En primer lugar tenemos el momento en el que se decide determinizar transiciones en las cuales haya por lo menos una maybe, en segundo lugar tenemos el caso en el que se decide la propagación de las transiciones maybe sobre las transiciones siguientes de una maybe determinizada. Solo en estos puntos podríamos tomar alguna decisión respecto de generar un modelo con mayor o menor nivel de refinamiento. Es decir, si no se están manipulando transiciones maybe el algoritmo se comporta exactamente igual que el algoritmo clásico de determinización de LTS [25]. Por lo explicado hasta aquí, analizaremos los casos en los cuales por lo menos alguna de las transiciones a determinizar esta marcada como maybe.

- *Todas las transiciones no determinísticas son maybe.*
  Tomemos sin perdida de generalidad dos transiciones $m \xrightarrow{\ell}_m n$ y $s \xrightarrow{\ell}_m t$, en este caso es claro que si tomáramos en el modelo determinizado la transición $[m,s] \xrightarrow{\ell}_r [n,t]$ estaríamos generando un modelo que no admite como implementación una en la cual se prohíba dicha transición, por lo que no valdría que $DetC(I_M) \not\subseteq I_N$. Lo mismo pasaría en el caso de no propagar la condición de maybe sobre las transiciones siguiente de n y t, porque en ese caso si tomáramos una implementación que admita la transición $[m,s] \xrightarrow{\ell} [n,t]$ necesariamente debería admitir tanto las siguientes de n como las de t por lo que el modelo original admitiría implementaciones que el determinizado no.

- *Alguna transición no deterministica es requerida. Tomemos las transiciones $m \xrightarrow{\ell}_m n$ y $s \xrightarrow{\ell}_r t$, en este caso es claro que si tomáramos en el modelo determinizado la transición $[m,s] \xrightarrow{\ell}_m [n,t]$ obtendríamos un modelo que no admite como implementación una en la cual solo se permitan las transiciones a continuación de t y por ende no valdría que $DetC(I_M) \not\subseteq I_N$. Si en cambio tomamos $[m,s] \xrightarrow{\ell}_r [n,t]$ importa analizar cuales son las posibles formas de tratar la transición maybe "'eliminada"'. En el caso de no propagar de ninguna forma las transiciones maybe, el modelo generado no tendría en su conjunto de implementaciones aquella que admite únicamente las transiciones sucesoras de t por lo que no valdría que $DetC(I_M) \not\subseteq I_N$*

□

3.4. Deadlocks en MTS

En esta sección analizaremos la semántica de los deadlocks en el contexto de los MTS. Presentaremos un algoritmo de chequeo de las posibles alternativas para deadlocks sobre MTS.

Asegurar modelos libres de deadlock es uno de los requerimientos principales para el diseño, análisis y validación de sistemas. En el contexto de los LTS, un modelo tiene deadlock si tiene un estado de deadlock. Un estado de deadlock es aquel que no tiene transiciones salientes. Las técnicas de model checking trabajan sobre modelos de trazas infinitas y por lo tanto libres de deadlock. En nuestro trabajo desarrollamos la teoría para el análisis de deadlocks en el contexto de los MTS.

Nuestro trabajo cubre dos aspectos, en primer lugar desarrollamos la definición para deadlock sobre MTS y en segundo lugar analizamos los algoritmos existentes con el fin de entender si era posible extenderlos para que detecten deadlocks sobre MTS.

Respecto de la teoría, en principio no está claro que significa que un MTS sea libre de deadlock o no, ni siquiera es simple ver si esa pregunta tiene sentido. Teniendo en cuenta que los MTS son caracterizados por su conjunto de implementaciones, desarrollamos una interpretación de los deadlocks en el mundo de los MTS.

Definición 28 (Libre de deadlocks) *Sea M un MTS, decimos que M es libre de deadlock si y sólo si $\forall L$ LTS, $M \preceq L \Rightarrow L$ es libre de deadlock,*

Intuitivamente un MTS es libre de deadlocks si y sólo si todas sus implementaciones son libres de deadlock (Definición 28). De esto último desprende que en el mundo de los MTS deberíamos poder responder más preguntas que solamente si un modelo es o no libre de deadlocks. Entonces es interesante poder responder si un MTS está en alguna de las siguientes tres afirmaciones:

- Todas las implementaciones son deadlock free.
- Todas las implementaciones tienen deadlock.
- Existen implementaciones deadlock free y no deadlock free.

Antes de implementar un algoritmo analizamos el algoritmo implementado en *LTSA* [25]. Encontramos que éste no es posible de modificar para buscar deadlocks en MTS debido a que el algoritmo busca deadlocks en el modelo sobre las transiciones del LTS y por lo tanto las modificaciones posibles buscarían deadlocks sobre algún subconjunto de trazas (i.e. posibles o requeridas) en el MTS. Entonces en el caso de tener un modelo con deadlock lo único que podemos decir es que existe al menos una implementación que tiene deadlock y en el caso contrario solo podemos decir que existe al menos una implementación sin deadlock. El algoritmo planteado de esta forma no puede responder las preguntas interesantes como: si todas las implementaciones de M

son libres de deadlock o si todas son implementaciones con deadlock. Estas preguntas son interesantes en sí mismas y mucho más para la verificación de propiedades FLTL, para lo cual es condición necesaria que el modelo tenga implementaciones deadlock free.

En primer lugar nuestro algoritmo (4) verifica si todas las implementaciones del modelo pasado como parámetro M tienen deadlock utilizando el método *todasImplsDeadlock* (línea 2), en caso negativo verifica si M tiene todas sus implementaciones libres de deadlock. El método *todasImplsDeadlock* (línea 12) tiene un ciclo principal que recorre todas las trazas a deadlock (línea 14) y analiza si el deadlock es inevitable, es decir, si toda implementación de M tendrá deadlock. Un deadlock es inevitable si se corresponde con alguna de las siguientes situaciones: 1) El deadlock es alcanzable por una traza requerida (e.g figura 3.17). 2) El deadlock es alcanzable por trazas maybe tales que no existe implementación posible sin deadlock. Mostramos un ejemplo de esta situación en la figura 3.16. El método *todasImplsSinDeadlock* (línea 21) basándose en el teorema 4 simplemente verifica si todo estado en M tiene al menos una transición requerida saliente.

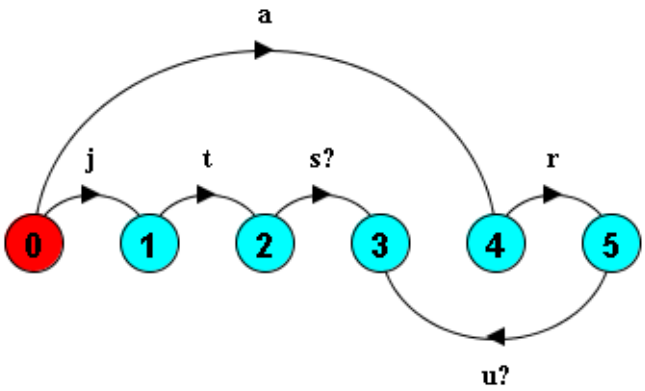

Figura 3.16: Modelo $\mathcal{U}$

Algoritmo 4

```
1:  procedure DEADLOCKSTATUS(MTS mtsEntrada)
2:      if todasImplsDeadlock(mtsEntrada) then
3:          devolver 1
4:      else
5:          if todasImplsSinDeadlock(mtsEntrada) then
6:              devolver 2
7:          else
8:              devolver 3
9:          end if
10:     end if
11: end procedure
12: procedure TODASIMPLSDEADLOCK(MTS mtsEntrada)
13:     Conjunto alcanzablesPorRequeridas = calcularAlcanzablesPorReq(mtsEntrada)
14:     for all traza ∈ trazasADeadlock(mtsEntrada) do
15:         if esDeadlockInevitable(traza, mtsEntrada, alcanzablesPorRequeridas) then
16:             devolver verdadero
```

```
17:         end if
18:     end for
19:     devolver falso
20: end procedure
21: procedure TODASIMPLSSINDEADLOCK(MTS mtsEntrada)
22:     for all estado ∈ estados(mtsEntrada) do
23:         if cantidadTransisionesRequeridas(estado, mtsEntrada)==0 then
24:             devolver falso
25:         end if
26:     end for
27:     devolver verdadero
28: end procedure
```

un modelo tal que independientemente del refinamiento que se tome no llegue a un estado de deadlock. Si en cambio no se encuentran estados marcados y se llega, retrocediendo, solo a estados con bifurcaciones sin marcar se puede saber que existen implementaciones deadlock free así (e.g aquellas que no incluyen a ningún estado del "camino" al estado de deadlock) como también se tienen implementaciones con deadlock, en particular M^+ será una implementación con deadlock.

Theorem 4 *Sea $M = (S, L, \Delta^r, \Delta^p, s_0)$ un MTS y C_M su conjunto de implementaciones, entonces vale que, todo estado de M tiene al menos una transición saliente requerida si y sólo si toda implementación de M es libre de deadlock.*

Demostración 3 *$\Leftarrow$)*
Tomemos $N = (S_N, L_N, \Delta, sn_0)$ un LTS tal que $M \preceq_R N$, sabemos que todo estado en M tiene al menos una única transición requerida saliente y supongamos que existe en N un estado t tal que es estado de deadlock. Como $M \preceq_R N$, existe un s estado de M tal que $(s,\ t) \in R$ y ademas por definición de refinamiento tiene que pasar que $\forall \ell \in Act$ vale, $(s \xrightarrow{\ell}_r s') \Rightarrow (\exists t' \cdot t \xrightarrow{\ell}_r t' \ \wedge \ (s', t') \in R)$ y $(t \xrightarrow{\ell}_p t') \Rightarrow (\exists s' \cdot s \xrightarrow{\ell}_p s' \ \wedge \ (s', t') \in R)$, y por lo tanto s solo puede ser o estado de deadlock o tener únicamente transiciones maybe salientes.

$\Rightarrow$)
Si todo estado s en M tiene al menos una transición requerida saliente y para toda implementación N de M vale que $M \preceq_R N$, entonces $\forall \ell \in Act$ vale, $(s \xrightarrow{\ell}_r s') \Rightarrow (\exists t' \cdot t \xrightarrow{\ell}_r t' \ \wedge \ (s', t') \in R)$ por lo tanto todo estado en N tiene al menos una transición requerida saliente, por lo tanto N no tiene estados de deadlock.

□

En las figuras 3.17, 3.18 y 3.19 vemos los modelos $\mathcal{L}$, $\mathcal{M}$ y $\mathcal{N}$ de los cuales, $\mathcal{L}$ tiene todas sus implementaciones con estados de deadlock, $\mathcal{M}$ no tiene implementaciones con deadlock y $\mathcal{N}$ tiene implementaciones con deadlock y libres de deadlock.

Es importante notar que para realizar model checking de propiedades es necesario como mínimo que el modelo tenga alguna implementación libre de deadlock. En este sentido el algoritmo de model checking depende de que primero se realice un chequeo de implementaciones libres de deadlock.

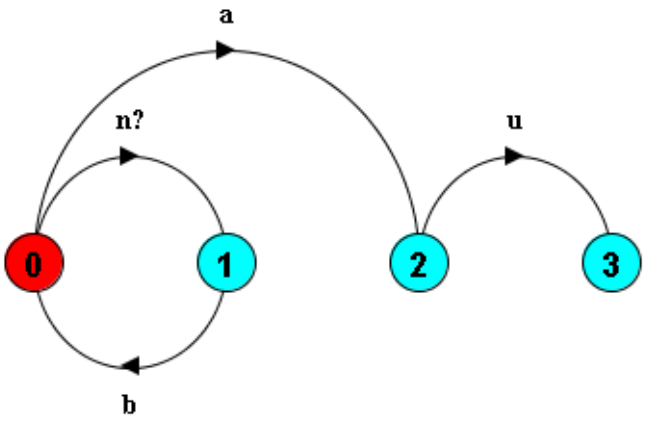

Figura 3.17: Modelo $\mathcal{L}$

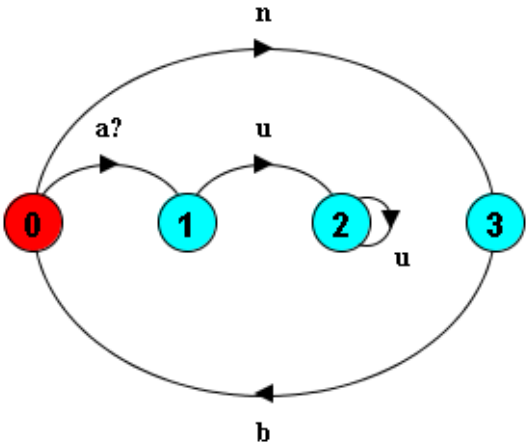

Figura 3.18: Modelo $\mathcal{M}$

3.5. Model Checking de FLTL en MTS

En esta sección definiremos los operadores *optimista* (M^+) y *pesimista* (M^-), así como también daremos la definición formal de model checking para MTS y luego veremos una propiedad muy importante: la distributividad de los operadores optimista y pesimista sobre la composición en paralelo de MTS.

El model checking es una técnica automatizada y exhaustiva de verificación de modelos. El problema del model checking puede describirse de la siguiente forma: Dado un modelo M que representa un sistema concurrente y una formula en lógica temporal

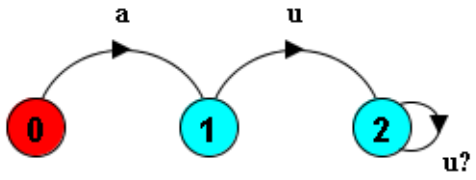

Figura 3.19: Modelo $\mathcal{N}$

ϕ que expresa cierta propiedad se busca saber si es cierto que el modelo satisface la formula ($M \models \phi$).

En el caso de los MTS verificar si una propiedad es satisfecha por el modelo significa, evaluar si es satisfecha por todas las implementaciones libres de deadlock del modelo.

Como parte de nuestro trabajo revisamos la definición de model checking sobre MTS [10, 1], así como también las definiciones de MTS optimista (M^+) y pesimista (M^-) [1]. Estas definiciones presentaban errores que corregimos en este trabajo. El caso de los modelos optimista y pesimista, su definiciones no contemplaban la eliminación de transiciones a deadlock. Por ejemplo, en el caso de la versión optimista se transformaba el comportamiento posible en comportamiento del LTS optimista pero no se eliminaban las posibles transiciones a deadlock. En función de que estos modelos son las bases del model checking de MTS y este a su vez requeiere modelos libres de deadlock es que estas operaciones eliminan las trazas a deadlock de los modelos resultantes.

En la figura 3.20 vemos un modelo tal que su versión optimista presenta admite una traza a deadlock ($0 \xrightarrow{b}_r 1$). Su versión optimista es la que mostramos en la figura 3.21.

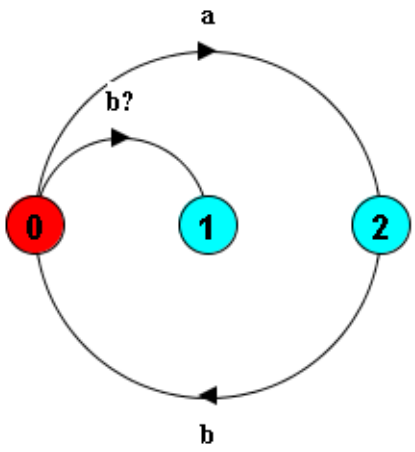

Figura 3.20:

Respecto de la definición de model checking dada en [1] encontramos que no era completa ya que no tenía en cuenta el caso en el cual la versión optimista sea vacía.

A continuación presentamos las definiciones finales.

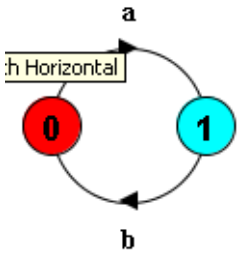

Figura 3.21:

Definición 29 (M^+) *Sea $M = (S, L, \delta_r, \delta_p, s_0)$ definimos M^+ como el LTS que admite todas las trazas posibles de M, es decir* PosTr*(M)=*Tr*(M^+) y además se eliminan todas las transiciones a estados de deadlock que pudieran quedar.*

Definición 30 (M^-) *Sea $M = (S, L, \delta_r, \delta_p, s_0)$ definimos M^- como el LTS que admite todas las trazas requeridas de M, es decir* ReqTr*(M)=*Tr*(M^-) y además se eliminan todas las transiciones a estados de deadlock que pudieran quedar.*

Theorem 5 *Sea M un MTS. Es cierto que:*

M^+ es el LTS vacío entonces M no tiene implementaciones libres de deadlock.

Demostración 4 *M^+ por definición es el LTS que admite las trazas posibles que no llevan a deadlock. Por lo tanto si M^+ es el LTS vacío significa que toda traza posible conduce a deadlock y esto pasa únicamente no existen implementaciones deadlock free.*

□

Theorem 6 (Model-checking) *Given a MTS M:*

1. $\|\phi\|^M = \mathbf{t} \Leftrightarrow \begin{cases} M^+ \models \phi, \text{ if } M^+ \text{ it's not the empty LTS} \\ \mathbf{t}, \text{ otherwise} \end{cases}$
2. $\|\phi\|^M = \mathbf{f} \Leftrightarrow \begin{cases} M^+ \models \neg\phi, \text{ if } M^- \text{ it is the empty LTS and } M^+ \text{ it is not the empty LTS} \\ M^- \not\models \phi, \text{ otherwise} \end{cases}$

otherwise, $\|\phi\|^M = \bot$.

Theorem 7 (Model-checking) *Para un MTS M:*

1. $\|\phi\|^M = \mathbf{t} \Leftrightarrow \begin{cases} M^+ \models \phi, \text{ si } M^+ \text{ si no es el LTS vacío} \\ \textbf{verdadero}, \text{ sino} \end{cases}$

2. $\|\phi\|^M = \mathbf{f} \Leftrightarrow \begin{cases} M^+ \models \neg\phi, \textit{ si } M^- \textit{ es el LTS vacío y } M^+ \textit{ no es el LTS vacío} \\ M^- \not\models \phi, \textit{ sino} \end{cases}$

en otro caso, $\|\phi\|^M = \bot$.

Demostración 5 *Consideremos la condición* 1. $\|\phi\|^M = \mathbf{t}$ *por definición pero significa que todas las trazas en* PosTr*(M) satisfacen* ϕ*. Esto es cierto por vacuidad en el caso que* M^+ *sea el LTS vacío. Si* M^+ *no es vacío entonces, podemos preguntarnos si* $M^+ \models \phi$ *lo cual vale debido a que* PosTr*(M)*=Tr*(*M^+*)*.

Consideremos la condición 2*, de izquierda a derecha. Si* $\|\phi\|^M =$ **verdadero** *entonces o existe una traza requerida de* M *que refuta* ϕ *o todas las trazas posibles de* M *refutan* ϕ*. Si* π *es una traza requerida de* M *que refuta* ϕ*, entonces el conjunto* $ReqTr(M)$ *no es vacío y* M^- *no es el LTS vacío. Además como* $\pi \in Tr(M^-)$ *sabemos que vale que* $M^- \not\models \phi$*. Si toda traza posible de* M *refuta* ϕ*, entonces o bien una de ellas es una traza requerida, lo cual nos lleva al primer caso o bien no existe ninguna traza requerida, en cuyo caso* M^- *es el LTS vacío. Debido a que todas las trazas de* M *refutan* ϕ *y que* $PosTr(M) = Tr(M^+)$*, entonces tenemos que* $M^+ \models \neg\phi$*.*

Consideremos ahora la condición 2*, de derecha a izquierda. Primero asumiremos que* M^- *es el LTS vacío y que* $M^+ \models \neg\phi$*. Esto significa que todas las trazas de* M^+ *refuta* ϕ*. Como* PosTr*(M)*=Tr*(*M^+*)*, *sabemos que todas las trazas posibles de* M *refutan* ϕ*, lo cual por la definición 19 significa que* $\|\phi\|^M = \mathbf{f}$*. Ahora asumiremos que* M^- *no es el LTS vacío y que* $M^- \not\models \phi$*. Entoces existe una traza en* $\pi \in Tr(M^-)$ *que refuta* ϕ*. Como* ReqTr*(M)*=Tr*(*M^-*)* *entonces existe una traza requerida de* M *que refuta* ϕ*, lo cual por la definición 19 significa que* $\|\phi\|^M = \mathbf{f}$*.*

□

Es importante notar que un MTS satisface una propiedad si y sólo si toda implementación del mismo la satisface. Es simple ver que la cantidad de implementaciones de un MTS puede ser infinita. La técnica de model checking sobre MTS podría en ciertos casos, ser una tarea imposible. El teorema 7 hace factible el model checking sobre MTS como una técnica independiente del modelo en particular, dado que para cualquier modelo y para cualquier propiedad FLTL son necesarios únicamente dos chequeos LTS.

La demostración de este teorema puede verse en [29].

La modificación más importante a las definiciones de MTS optimista (M^+) y pesimista (M^-) que incorporamos en este trabajo es la eliminación de las trazas a deadlock en ambas versiones. Esto se debe a que como hemos mostrado éstas operaciones son la base del model checking para MTS, por lo tanto es necesario trabajar con modelos libres de deadlock.

Theorem 8 (Distributividad + respecto de la composición) *Dados los modelos $M_1, \ldots, M_n$ MTS, es cierto que,*

$$(M_1||\ldots||M_n)^+ \sim (M_1^+||\ldots||M_n^+)$$

Demostración 6 *Sea R la relación identidad, veamos que cumple con la definición de relación de bisimulación. Todas las transiciones en $(M_1||\ldots - ||M_n)^+$ son transiciones posibles en $M_1||\ldots||M_n$ porque provienen de sincronizar por cualquiera de las reglas de sincronización (MD, TD; TT; MT o MM). Las transiciones requeridas de los $M1\ldots Mn$ sincronizan por TD o TT, se mantienen tanto para $(M_1||\ldots||M_n)^+$ como para $(M_1^+||\ldots||M_n^+)$. Las transiciones que sincronizan por MD, MT, MM en $M1||\ldots||M_n$ quedan como requeridas en $(M_1||\ldots||M_n)^+$ y esas mismas también aparecen en $(M_1^+||\ldots||M_n^+)$. Dado que aplicando + sincronizan por TD y TT, por lo que las transiciones en $(M_1||\ldots||M_n)^+$ y $(M_1^+||\ldots||M_n^+)$ son las mismas, por lo tanto R es una relación de bisimulación.*

□

Theorem 9 (Distributividad − respecto de la composición) *Dados los modelos $M_1, \ldots, M_n$ MTS, es cierto que,*

$$(M_1||\ldots||M_n)^- \sim (M_1^-||\ldots||M_n^-)$$

Demostración 7 *Este teorema se demuestra en forma dual al teorema 8*

□

El resultado de los teoremas 8 y 9 es muy útil debido a si uno tuviera varios modelos representando diferentes vistas parciales de un componente, no es necesario componerlos utilizando semántica MTS para obtener el modelo resultante. Uno puede primero aplicar el operador + o − y luego componer los LTS resultantes. Por lo tanto es posible trabajar sobre LTS aprovechando el desarrollo, la experiencia y (en un plano mas implementativo) las optimizaciones sobre ellos.

Capítulo 4

MTSA: Implementación

En este capítulo detallaremos las cuestiones técnicas que hacen a la herramienta *MTSA*: motivación, decisiones de diseño, arquitectura de la herramienta y la funcionalidad implementada.

4.1. Modal Transition Systems Analyser

Modal Transition Systems Analyser (MTSA) es una herramienta que soporta construcción, elaboración y análisis de Modal Transition Systems. La desarrollamos en Java como una extensión de la ya conocida y probada Labelled Transition Systems Analyser (LTSA) [25]. Esto se debe fundamentalmente a dos razones: 1) Los MTS son la extensión natural de los LTS, esto favorece la reutilización y modificación tanto de estructuras de datos como de algoritmos. 2) Por lo visto en la subsección sección 7 podemos aprovechar la experiencia, el desarrollo y la optimización sobre los algoritmos de model checking implementados en LTSA.

4.1.1. Arquitectura

La arquitectura MTSA (figura 4.1) nos permite desacoplar la funcionalidad de MTSA respecto de LTSA. Esto nos permite integrar nuevas funcionalidades de forma transparente, casi sin modificar el núcleo de LTSA.

La arquitectura (Fig. 4.1) está formada por tres capas: La interfaz de usuario, el núcleo (o core) y las mtsa-commons.

La interfaz de usuario implementa las diferentes vistas de la herramienta (Editor FSP, Output, Gráfica, Animación, etc.). La capa *mtsa-commons* está formado por librerías desarrolladas por nosotros (fsp2mts) y por terceros (commons-lang) que nos permiten compartir y reutilizar funcionalidad entre los diferentes módulos de la aplicación.

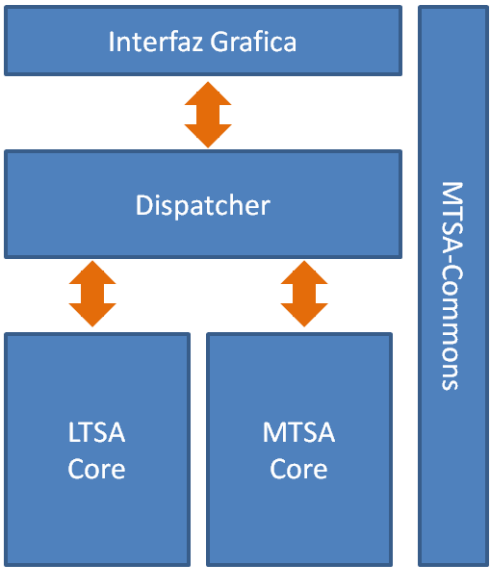

Figura 4.1: Arquitectura MTSA

El Core de la herramienta está compuesto por tres módulos: *LTSA Core*, *MTSA Core* y *Dispatcher*.

LTSA Core es el resultado de encapsular la funcionalidad encargada del procesamiento de los modelos dentro de LTSA. Este módulo cumple tres objetivos primordiales:

- Mantener clara y acotada la interfaz de LTSA.
- Minimizar las modificaciones de código en LTSA.
- Actualizar o modificar funcionalidad en LTSA independientemente de MTSA.

Algunas de las funcionalidades de LTSA que reutilizamos y/o modificamos son:

- Análisis sintáctico.
- Reducción de ordenes parciales.
- Alcanzabilidad de estados.
- Transformación de FLTL a autómatas de Büchi [9].
- Model checking de LTS.

MTSA Core contiene las estructuras de datos y los algoritmos específicos de MTS. En este módulo implementamos gran parte de la funcionalidad propia de MTSA que detallaremos en la sección 4.2.

El módulo *Dispatcher* actúa de fachada [8] entre la interfaz gráfica, el MTSA Core y el LTSA Core. Evita puntos de comunicación directa entre MTSA Core y LTSA Core, la interacción entre ellos también se realiza a través del Dispatcher. Si el usuario trabaja con LTS, el Dispatcher envía el pedido al LTSA Core; en caso contrario al MTSA Core.

Tomar LTSA como base nos permite aprovechar su estabilidad, pudiendo focalizarnos en la corrección de los algoritmos propios de MTSA, los cuales son el resultado de la teoría desarrollada en nuestro trabajo.

Por último esta arquitectura nos permite proyectar para un futuro trabajo la separación completa de MTSA de LTSA.

4.2. Funcionalidad

En esta sección mostraremos las funcionalidades principales de *MTSA*.

- **Escritura y visualización de modelos.**
  MTSA provee un *editor FSP* en el cual se puede escribir los modelos utilizando una extensión del lenguaje llamado *Finite State Processes* (FSP), lenguaje basado en texto usado originalmente en LTSA [25]. En la figura 4.2 mostramos el *editor FSP* con un ejemplo de código. El mismo declara un proceso llamado `LLAVE`, utilizando el prefijo de acción (`‘‘->’’`) y el operador de opción (`|`) para modelar el comportamiento esperado. Las acciones con el símbolo `?` representan las acciones *maybe*. El modelo describe una llave de luz en la cual se garantiza que es posible alternadamente prender `prenderLuz` y apagar `apagarLuz` la luz, pero todavía no esta definido si puede apagarse (prenderse) una llave apagada (prendida).

  MTSA permite compilar los modelos utilizando el menú o la barra de herramientas (figura 4.2)

  Luego de compilar el usuario puede ver de forma gráfica el modelo resultante en la *vista Draw*. En esta vista es posible manipular el modelo alterando su forma (ancho y alto) para lograr una mejor vista del mismo. En la figura 4.3 mostramos la representación gráfica del modelo de la llave de luz obtenida a partir de compilar el código de la figura 4.2.

- **Composición de modelos.**
  MTSA soporta la construcción de modelos en forma *composicional*, cuya definición formal puede verse en la definición 1. Es posible componer modelos en paralelo al estilo CSP [14] para construir nuevos modelos que describan el resultado de ejecutar en forma concurrente los modelos de dos componentes diferentes de forma asincrónica pero sincronizándose a través de acciones compartidas. Esta forma de composición recibe el nombre de *composición en paralelo* y a sido el enfoque principal del modelado composicional de comportamiento.

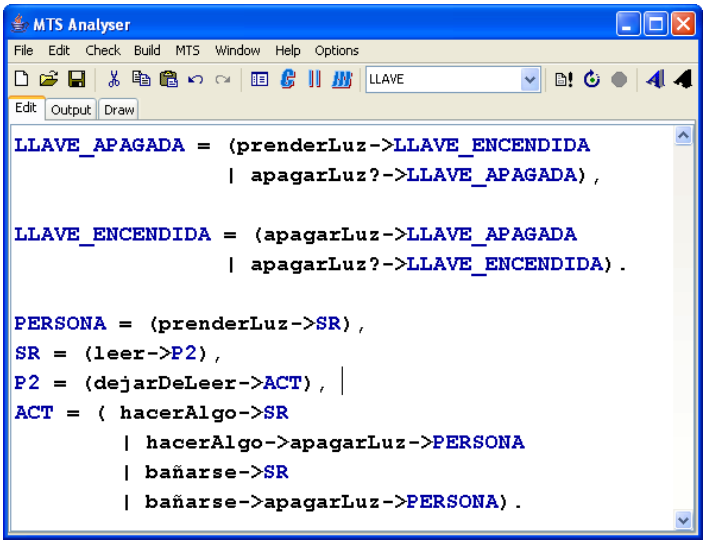

Figura 4.2: Llave de luz - Editor FSP

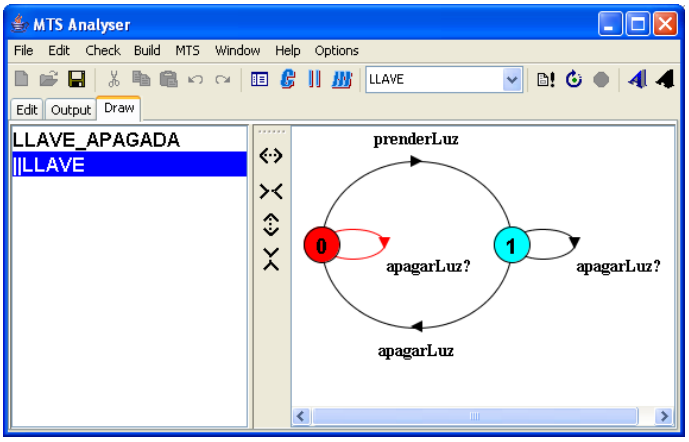

Figura 4.3: Llave de luz - Vista Draw

En la figura 4.5 presentamos el modelo resultante de la composición en paralelo de los modelos `PERSONA` y `LLAVE` (figuras 4.4 y 4.3 respectivamente). El código FSP de éstos modelos está capturado en la imagen 4.5.

Además de la composición en paralelo existe un mecanismo de composición alternativo llamado *merging*. En el contexto de la elaboración de modelos es interesante la composición de diferentes vistas parciales (i.e MTSs) del mismo componente para obtener un modelo más "completo" que cualquiera de los originales. El *merge* de dos modelos debería devolver un MTS que fuera su refinamiento común mínimo (i.e least common refinement). En otras palabras, este modelo debería ser el menos refinado posible tal que garantice el comportamiento requerido y prohibido de los modelos a ser compuestos.

En esta versión de MTSA implementamos dos operadores de merging: `+cr` y `+ca` [30].

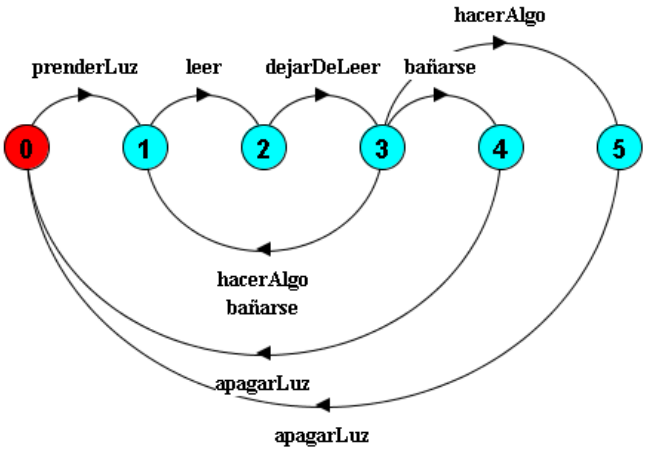

Figura 4.4: Modelo `PERSONA`

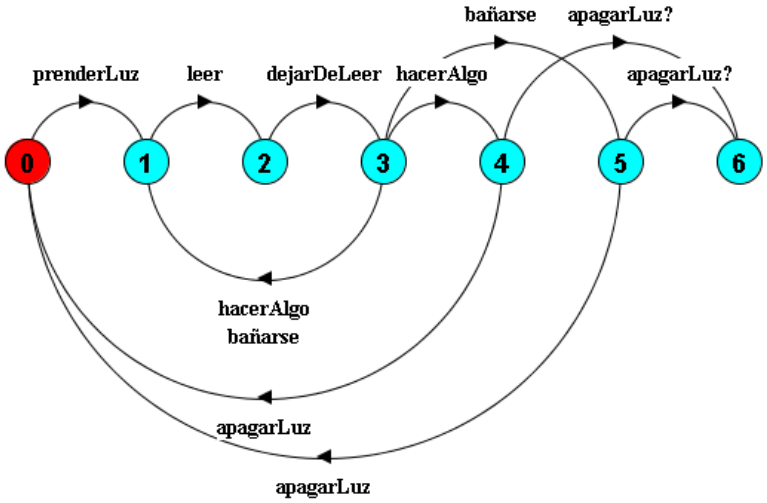

Figura 4.5: Composición entre `PERSONA` y `LLAVE`

Los modelos `LegalAccess` (figura 4.6) y `PrivateAccess` (figura 4.7) representan las politicas de acceso a un sistema de webmail. El primero de ellos describe habilitación e inhabilitación de usuarios al webmail. El segundo muestra que los usuarios deben autenticarse para poder usar el webmail. En la figura vemos el merge entre ellos utilizando el operador `+cr`.

- **Chequeo de refinamiento.**
  Incorporamos el chequeo de refinamiento de modelos MTS a la herramienta. Para mejorar la usabilidad implementamos la interfaz del chequeo de refinamiento entre modelos como una ventana en la cual se seleccionan los modelos a chequear. Ademas puede seleccionarse la semántica a utilizar y por último es posible chequear equivalencia entre modelos seleccionando el checkbox *check both sides*, la imagen 4.9 muestra esta ventana. Mostramos la salida del chequeo de refinamiento de los modelos `LegalAccess` y `PrivateAccess` en la figura 4.10. Estos modelos no son equivalentes por simulación.

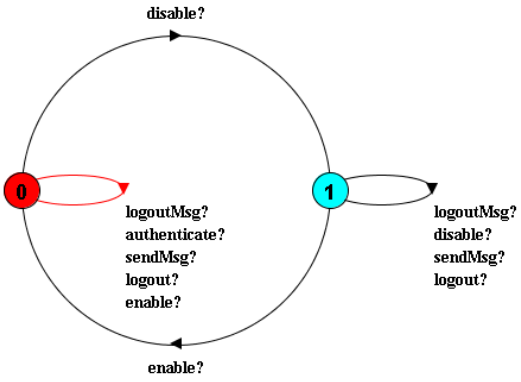

Figura 4.6: Modelo PrivateAccess

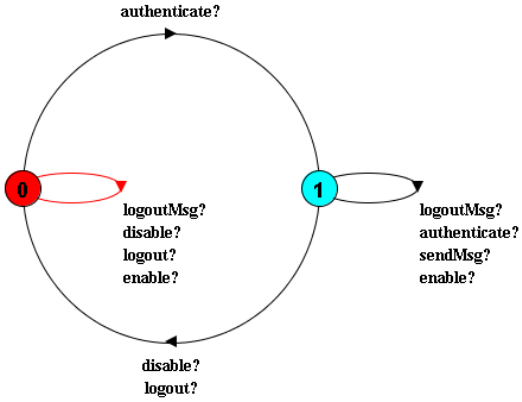

Figura 4.7: Modelo de acceso restringido

- **Operador `constraint`.**

  A pesar de que los operadores de composición ayudan y facilitan la construcción de modelos MTS complejos, construir los modelos a ser compuestos sigue siendo una tarea difícil, costosa y que requiere un nivel considerable de experiencia. Con la intención de alivianar este trabajo incorporamos en MTSA algunos mecanismos automáticos para generar (i.e sintetizar) modelos a partir de especificaciones declarativas de requerimientos (e.g. [28, 19]) en forma de propiedades. Los detalles sobre estas técnicas son presentados en [29].

  Las propiedades pueden ser pensadas como *requerimientos* que acotan el espacio de comportamientos posibles que debería admitir el sistema. Un modelo sintetizado a partir de propiedades caracteriza todos los posibles comportamientos que no violan dichas propiedades. En función de esto entendemos que un modelo con estas características da el *límite superior* del comportamiento que el sistema proveerá una vez implementado.

  MTSA soporta la síntesis de modelos a partir de propiedades por medio del operador `constraint`. Incorporamos a este operador como parte de nuestra extensión al FSP original de LTSA. A continuación damos una definición formal del operador.

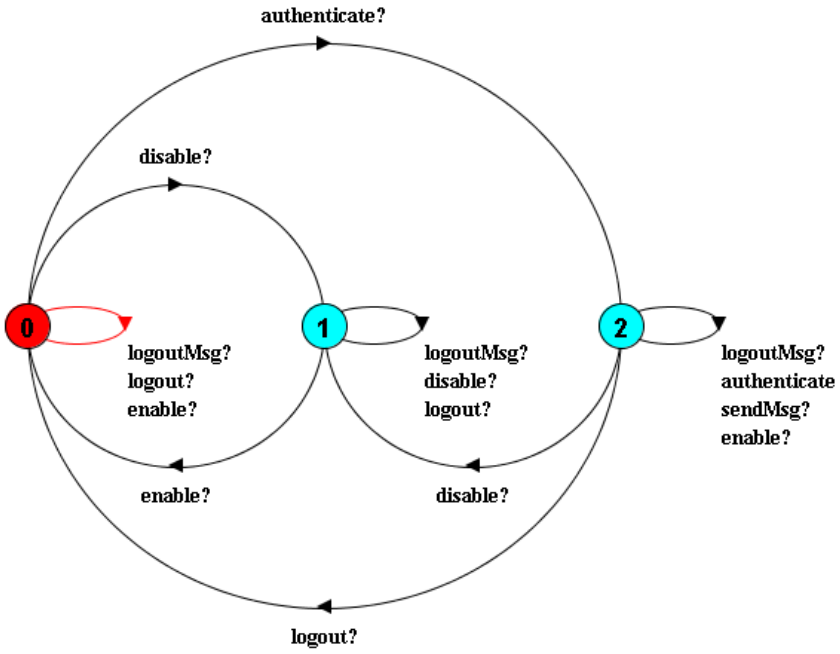

Figura 4.8: Merge sobre operador `+CR`

Definición 31 *Dado* $M = (S_M,\ L_M,\ \Delta^r_M,\ \Delta^p_M,\ s_{M0})$ *y* $N = (S_N,\ L_N,\ \Delta^r_N,\ \Delta^p_N,\ s_{N0})$ *MTSs, decimos que* N *es el resultado de aplicar el operador* contraint *sobre* M *si vale lo siguiente:*

1. $S_N = S_M$
2. $L_N = L_M$
3. $\Delta^p_N = \Delta^p_M$
4. $s_{N0} = s_{M0}$
5. $\forall s \in S_M,\ \exists s',\ s'' \in S_M\ \ \exists \ell,\ \ell' \in L_M\ |$
$$(s' \neq s'')\ \wedge (s \xrightarrow{\ell}_r s') \in \Delta^p_M \wedge (s \xrightarrow{\ell'}_r s'') \in \Delta^p_M \Rightarrow$$
$$(s \xrightarrow{\ell}_m s') \in \Delta^p_N \wedge (s \xrightarrow{\ell'}_m s'') \in \Delta^p_N$$

El uso del operador esta ejemplificado en la última linea de código FSP en la figura 4.11. Mostramos el modelo resultado gráfico de compilar dicho FSP en la figura 4.12.

- **Operador *abstract*.**
En su forma más simple y ampliamente utilizada los escenarios son declaraciones *existenciales* (i.e proveen ejemplos) del comportamiento esperado de un sistema; en otras palabras, son secuencias interacciones que se espera que el sistema admita.

  Los escenarios son inherentemente descripciones parciales, en general no se asume que el comportamiento que ellos describen sea una versión completa de lo que

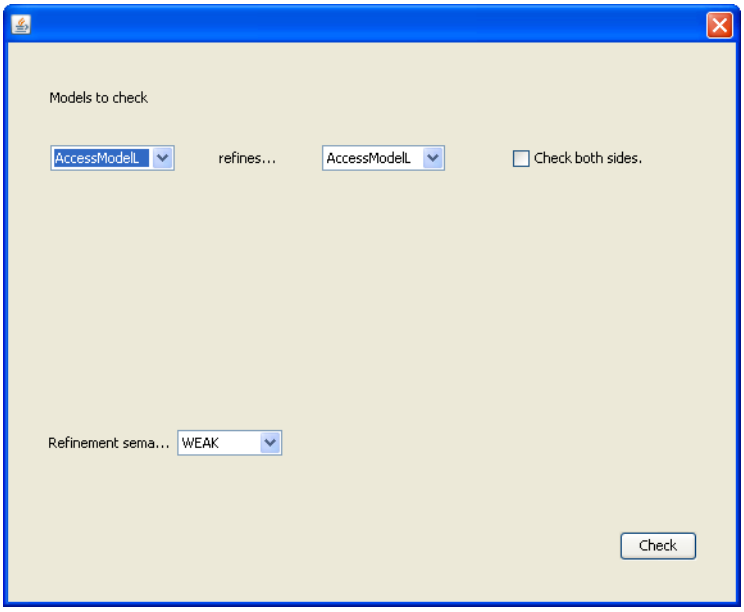

Figura 4.9: Ventana de selección de refinamiento

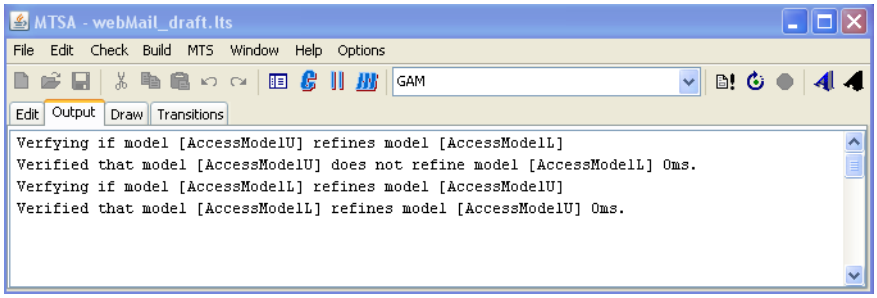

Figura 4.10: Salida chequeo refinamiento entre LegalAccess y PrivateAccess

se espera que el sistema soporte a ningún nivel de abstracción. Por lo tanto un modelo de comportamiento sintetizado a partir de escenarios debería proveer el *limite inferior* a partir del cual es posible identificar el comportamiento que el sistema proveerá pero que no fue capturado por los escenarios.

Tomando esto último y entendiendo que los modelos MTS son ideales para trabajar iterativamente modelando el comportamiento de sistemas, implementamos en MTSA un operador que llamamos `asbtract`. Este operador nos permite extender un modelo a su versión más abstracta posible. El resultado aplicar el operador `abstract` sobre un MTS es el modelo más abstracto posible tal que admite como requerido el comportamiento del modelo original y deja indefinido el resto. A continuación damos una definición formal del operador.

Definición 32 *Dados* $M = (S_M, L_M, \Delta^r_M, \Delta^p_M, s_{M0})$ *y* $N = (S_N, L_N, \Delta^r_N, \Delta^p_N, s_{N0})$ *MTSs, decimos que* N *es el resultado de aplicar el operador* abstract *sobre* M *si vale lo siguiente:*

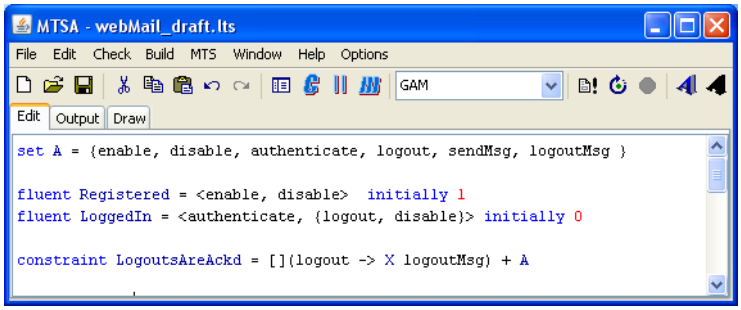

Figura 4.11: Ejemplo operador `Constraint`

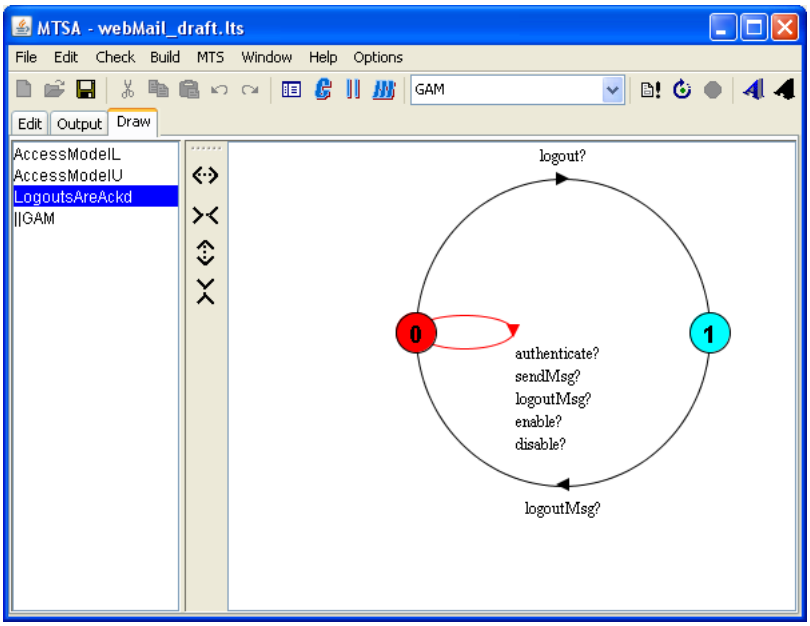

Figura 4.12: Modelo `+cr`

1. $S_N = S_M \cup \{t\},\ t \notin (S_M \cup S_N)$
2. $L_N = L_M$
3. $\Delta^r_N = \Delta^r_M$
4. $\forall \ell \in L_M,\ \forall s \in S_M,\ \neg\exists s' \in S_M \mid (s \xrightarrow{\ell}_p s') \in \Delta^p_M \Rightarrow s \xrightarrow{\ell}_m t \in \Delta^p_N$
5. $s_{N0} = s_{M0}$

Ejemplificamos la forma de uso del operador `abstract` en el FSP de la figura 4.14. El modelo *escenario*1 en la figura 4.13 representa la versión original del modelo. En la figura 4.15 mostramos el resultado de aplicar el operador `abstract` a *escenario*1. La diferencia entre los modelos radica en que este último incorpora como maybe el comportamiento (basado en las acciones en el alfabeto) que el escenario no menciona.

- **Merge. Operadores *+ca* y *+cr***
  Implementamos algoritmos para las versiones *+ca* y *+cr* del merge tal cual son

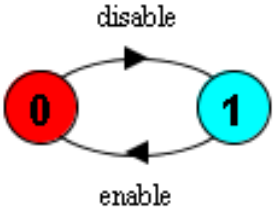

Figura 4.13: Escenario nº1

```
set A = {enable, disable, authenticate, logout, sendMsg, logoutMsg }

Scenario1 = (disable -> enable -> Scenario1) + A.
```

Figura 4.14: FSP (`abstract`) Escenario nº1

definidos en [30]. Respecto de la implementación fue necesario desarrollar el calculo de la relación de consistencia [1].

Si el usuario desea trabajar con más de dos modelos definimos que los operadores se asocian de izquierda a derecha. Por lo tanto el resultado de los siguientes códigos FSP es equivalente:

- Código con asociación implícita: `M = (A +cr B +cr C).`
- Código con asociación explícita: `M = ((A +cr B) +cr C).`

- **Alcanzabilidad de estado de error.**
  Si el usuario desea verificar ausencia de deadlock debe utilizar la funcionalidad de chequeo de deadlock. Las posibles respuestas que MTSA puede dar, son las siguientes:

  - Si el modelo no tiene estados de deadlock
    - Si el modelo no tiene estado de error o éste no es alcanzable
      En este caso presenta en la salida que no hay errores.
    - Si el modelo tiene estado de error y éste es alcanzable
      En este caso presenta en la salida una traza al estado de error.
  - Si el modelo tiene estados de deadlock
    - Si el modelo no tiene estado de error o éste no es alcanzable
      En este caso presenta en la salida una traza al estado de deadlock.
    - Si el modelo tiene estado de error y éste es alcanzable
      En este caso presenta en la salida la primer traza que encuentre, ya sea al estado de error o al de deadlock.

  En MTSA decidimos implementar éstas dos funcionalidades en forma separada. De forma tal que si el usuario quiere chequear alcanzabilidad de estados de error en MTSA puede utilizar la opción de menú *check* → $safety$ y si lo que busca es efectivamente chequear si el modelo MTS tiene implementaciones libres de deadlock entonces deberá utilizar la opción de menú *check* → *deadlock*.

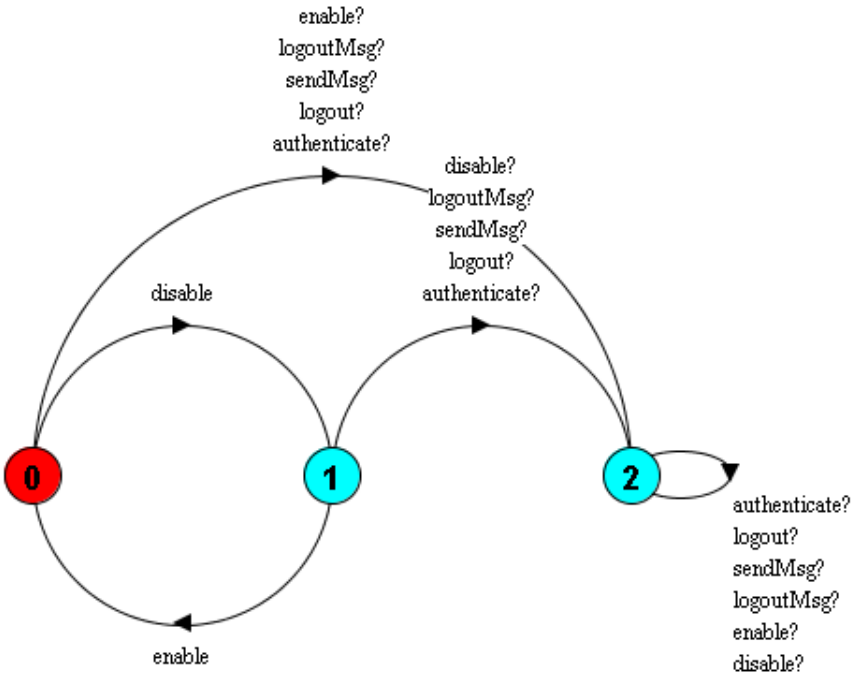

Figura 4.15: Versión `abstract` del Escenario n°1

- **Checkeo FLTL.**
  Implementamos el chequeo de formulas FLTL basándonos en nuestra definición (sección 3.5). El algoritmo 5 muestra el pseudo-código del chequeo FLTL que implementamos.

Algoritmo 5

```
1:  procedure MODELCHECKING(MTS mts, Formula f)
2:      optimista ← obtenerOptimista(mts)
3:      if esVacio(optmimista) then
4:          devolver verdadero
5:      else
6:          if satisface(optimista, f) then
7:              devolver verdadero
8:          else
9:              pesimista ← obtenerPesimista(mts)
10:             if esVacio(pesimista) then
11:                 negF ← ¬ f
12:                 if satisface(optimista, negF) then
13:                     devolver falso
14:                 end if
15:             else
16:                 if ¬satisface(pesimista, f) then
17:                     devolver falso
18:                 end if
19:             end if
```

```
20:          end if
21:      end if
22:      devolver maybe
23: end procedure
```

A modo de ejemplo analizaremos los modelos descritos por el código FSP en la figura 4.17 y tendremos en cuenta la formula LTL P, descrita en la misma imagen. El FSP declara la propiedad P y los MTS $TRUE\_MODEL$, $FALSE\_MODEL$ y $MAYBE\_MODEL$. En primer lugar veamos cómo se traduce la formula P, si la pensamos en función de los modelos que la satisfacen, P pide que en toda traza, si hay una transición por la etiqueta "a" siempre tiene que pasar que la transición siguiente esté etiquetada con "b". Entonces para los modelos del ejemplo vale que:

1. $TRUE\_MODEL \models P \equiv true$
2. $FALSE\_MODEL \models P \equiv false$
3. $MAYBE\_MODEL \models P \equiv maybe$

El predicado 1 es cierto porque la versión optimista $TRUE\_MODEL+$ admite como únicas trazas posibles las descritas por la siguiente expresión regular $(ab)^+$, por lo que es cierto que siempre que hay una transición por la etiqueta "a" debe haber luego una transición por la etiqueta "b".

El predicado 2 es cierto porque el modelo no tiene "b" en su alfabeto por lo tanto ni su versión optimista ni su versión pesimista admitirán una traza con "b".

El predicado 3 es cierto porque la versión optimista $MAYBE\_MODEL+$ admite la traza $a \rightarrow n \rightarrow \ldots$ que claramente no satisface P. Y por otro lado, la versión pesimista $MAYBE\_MODEL-$ satisface P por lo tanto, el modelo $MAYBE\_MODEL$ evalúa a maybe chequeado contra P.

La salida de la herramienta para cada caso es el que se muestra en las figuras 4.18, 4.19 y 4.20 respectivamente. En el caso de estudio en la sección 5 veremos un ejemplo completo del chequeo de propiedades FLTL sobre modelos MTS.

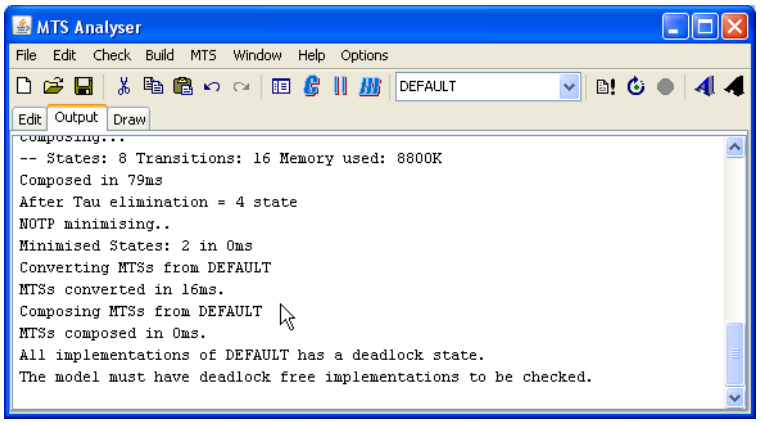

Figura 4.16: Salida de chequeo FLTL: Modelo sin implementaciones libres de deadlock

```
MAYBE_MODEL_ = (a?->n->MAYBE_MODEL_ | a->b->MAYBE_MODEL_ ).
FALSE_MODEL_ = (a?->n->FALSE_MODEL_).
TRUE_MODEL_ = (a?->b->TRUE_MODEL_).

||TRUE_MODEL = TRUE_MODEL_.
||MAYBE_MODEL = MAYBE_MODEL_.
||FALSE_MODEL = FALSE_MODEL_.

//assert P = [](X a -> b)||(X a -> n))
assert P = [](X a -> b)
```

Figura 4.17: Modelo y Propiedad a chequear

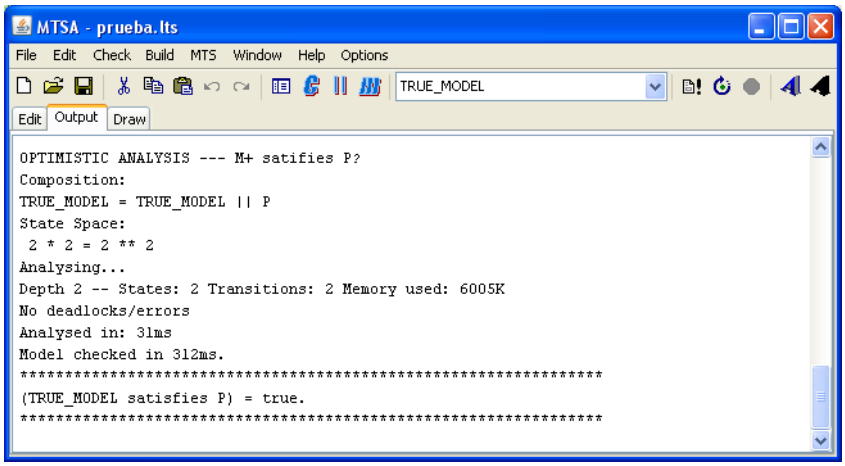

Figura 4.18: Salida de chequeo FLTL: Verdadero

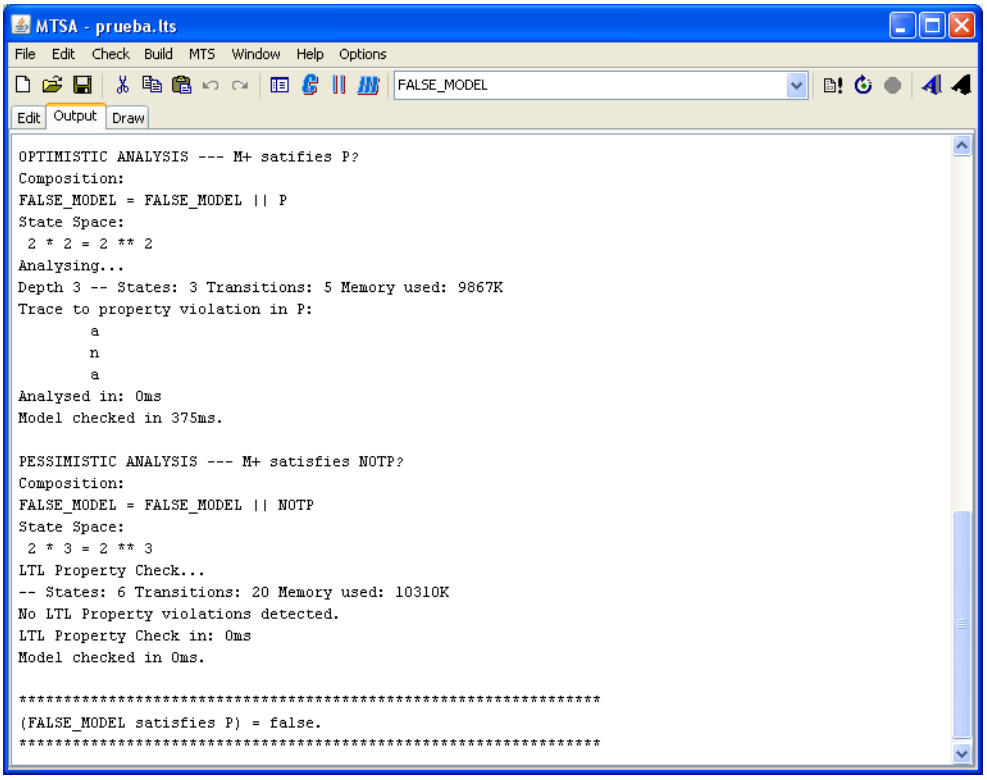

Figura 4.19: Salida de chequeo FLTL: Falso

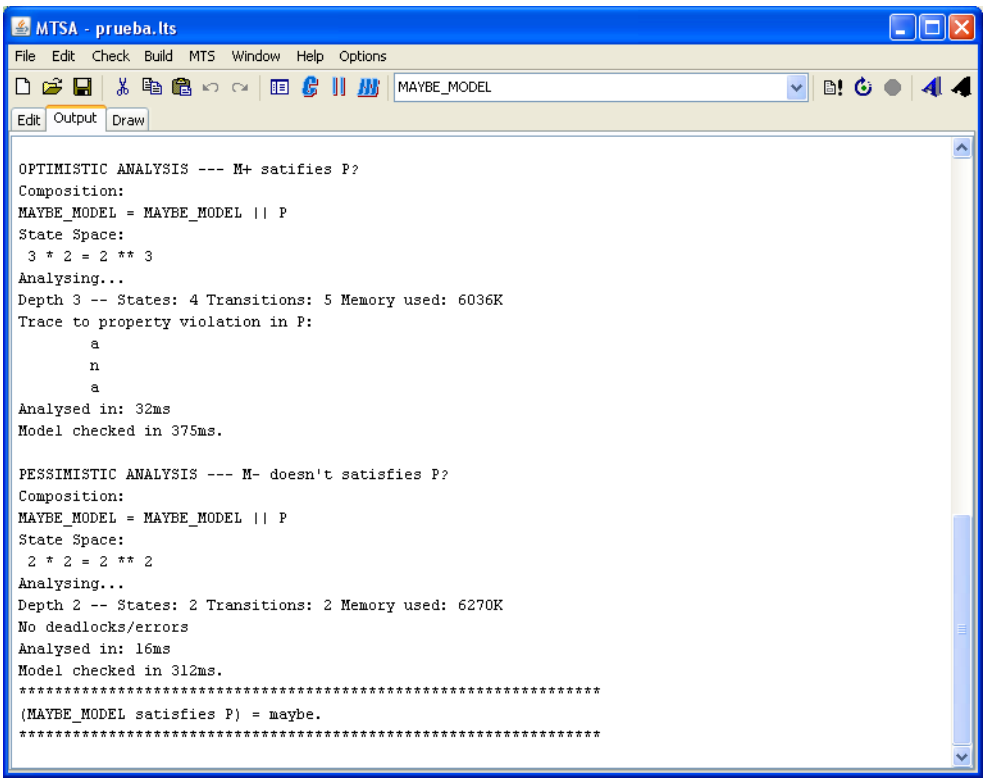

Figura 4.20: Salida de chequeo FLTL: Maybe

Capítulo 5

Caso de estudio

En este capítulo aplicaremos nuestros resultados a un caso de estudio basado en el controlador de sistema de desagüe de una mina [21]. En la sección 5.1 describiremos la idea general del caso de estudio y de lo que se espera del sistema de bombeo. Luego en la sección 5.2 trabajaremos sobre un escenario y analizaremos una propiedad de "'safety"' sobre él. Por último en la sección 5.3 aplicaremos las técnicas que generamos a varias propiedades y utilizando MTSA mostraremos un ciclo de diseño, análisis y verificación de modelos.

5.1. Descripción

En este sistema se usa un controlador de bombeo para prevenir que el desagüe de una mina supere un límite dado y por lo tanto inunde la mina. Para evitar explosiones, el bombeo sólo puede ocurrir si no hay gas metano en el ambiente dentro de la mina. El controlador monitorea los niveles de agua y de metano por medio de dos sensores y controla la bomba para garantizar las propiedades de seguridad del sistema de bombeo. Además el controlador es responsable de prender y apagar una luz de peligro dentro de la mina para reflejar la presencia o ausencia de gas en la mina.

El sistema de bombeo esta compuesto por cuatro componentes: *SensorNivelAgua, SensorMetano, Bomba y SistemaControl*. El sistema de bombeo completo es la composición en paralelo de estos tres componentes. *SensorNivelAgua* modela el sensor de nivel de agua en la mina e incluye suposiciones acerca de cómo se espera que el nivel de agua vaya cambiando entre los estados bajo, medio y alto. *SensorMetano* lleva el control de cuando hay o no metano presente en la mina, mientras que *Bomba* modela la bomba física que puede ser prendida y apagada.

En este caso de estudio trabajamos asumiendo que contamos con un escenario del funcionamiento esperado del *SistemaControl* dado por el responsable de la mina y algunas propiedades dadas por un experto en minas. La idea del caso de estudio era partir del escenario e ir haciéndolo evolucionar. Básicamente queríamos ver si el control

satisface las propiedades dadas y en función de los resultados continuar el trabajo sobre los modelos.

El código completo de este caso de estudio está disponible en el apéndice 5.

A lo largo de esta sección al referirnos a la operación de merge, salvo que se aclare lo contrario estaremos hablando del operador de merge $+cr$ descrito en la sección 4.2.

5.2. Diseño, verificación y extensión. Iteración I

Asumimos que el experto en la mina además de darnos el escenario nos ha dado algunas propiedades que se espera que el sistema de control cumpla.

Para escribir las propiedades hemos utilizado una serie de fluents. Estos se ven en la primera parte de la figura 5.1. Las expresiones FLTL que representan las propiedades pueden verse en la figura 5.2. `P1` pide que la bomba solo puede ser prendida si estaba apagada. `P2` pide que la bomba solo puede ser apagada solo si esta prendida.

```
const False = 0
const True  = 1
range Bool  = False..True

fluent PumpOn = <switchOn, switchOff> initially False
fluent MethanePresent = <methAppears, methLeaves> initially False
fluent AtHighWater = <highWater, {lowWater, medWater}> initially False
fluent AtLowWater = <lowWater, {medWater, highWater}> initially True
```

Figura 5.1: Escenario Bomba

```
set A = {switchOn, switchOff, methAppears,
 methLeaves, highWater, lowWater,
 medWater, switchDLOff, switchDLOn, tick}

assert P1 = [](PumpOn -> (X !switchOn W !PumpOn)) + A
assert P2 = [](!PumpOn -> (X !switchOff W PumpOn)) + A
assert P3 = []((AtHighWater && !MethanePresent) -> X PumpOn) + A
assert P4 = []((AtLowWater || MethanePresent) -> X !PumpOn) + A
```

Figura 5.2: Propiedades deseadas del Sistema de Control

En este primer intento nuestra idea fue la de partir del escenario e ir trabajando con él y las propiedades de forma tal de obtener un modelo consistente con los requerimientos. Al trabajar con escenarios es posible que estos tengan errores. Puede pasar que en realidad el problema no provenga del modelo sino de la propiedad. Parte del trabajo de modelado es entender cual es el origen de estos problemas.

El código completo de la primera iteración puede verse en el apéndice 5.

Como parte del proceso de modelado, si partimos de un FSP que es el resultado de un escenario quisiéramos conseguir el modelo más abstracto posible que admita como mínimo el comportamiento descrito por el escenario. Para esto aplicamos a

$EscenarioIT1$ el operador `abstract`. Obtuvimos un modelo de 44 estados, el cual fue bastante difícil de seguir y entender. Un modelo con 44 estados es difícil visualizar e inspeccionar. Tener modelos lo más pequeños posibles, para facilitar el análisis, reducir los tiempos de las operaciones como merge, abstracción y evitar la explosión de estados al componer modelos. Por esto y para evitar una posible explosión de estados a medida que fueramos avanzando en el proceso de modelado, aplicamos minimización sobre el modelo abstracto. El operador `minimal` nos devolvió un modelo de 22 estados, el cual es más manejable y entendible. Tener

Tomando el modelo minimizado comenzamos a trabajar con las propiedades. En esta iteración nos interesaba analizar nuestro modelo generado a partir del escenario. Nos referiremos al escenario minimizado como $EscenarioMinimizado$. El primer paso fue verificar si $EscenarioMinimizado$ satisface la propiedad `P1` ($EscenarioMinimizado \models$`P1`). El resultado del chequeo nos mostró que el modelo tiene implementaciones que satisfacen `P1` e implementaciones que no lo hacen, por lo tanto el resultado entregado por MTSA fue *maybe*. Por lo tanto tiene que pasar que la versión optimista del modelo no satisface la propiedad y la versión pesimista si lo hace. A partir de la información de la salida de MTSA vimos que la versión optimista de $EscenarioMinimizado$ no satisface `P1` debido a que admite trazas con secuencias de la forma $\ldots \rightarrow switchOn \rightarrow switchOn \ldots$, en particular MTSA muestra $\pi = switchOn \rightarrow switchOn \ldots$ como traza que viola la propiedad.

Analizando el modelo $EscenarioMinimizado$ observamos que admite estas secuencias debido a haber aplicado el operador `abstract` sobre $EscenarioAbstracto$. En el escenario original desde el estado 0 la acción $switchOn$ esta prohibida por lo tanto luego de aplicar el operador `abstract` desde el estado 0 se habilita una transición maybe por $switchOn$ al estado trampa desde el cual están habilitadas transiciónes sobre todas las acciones del alfabeto, en particular sobre $switchOn$.

Siendo el resultado del análisis maybe, entendimos que `P1` es satisfecha por algunas implementaciones $EscenarioMinimizado$ y por otras no. Por lo tanto tenía sentido construir el MTS que la representa y mergearlo con $EscenarioMinimizado$. El resultado de esta operación es un modelo más refinado que $EscenarioMinimizado$ pero que admita únicamente implementaciones que satisfagan `P1`. El modelo $Constraint\_P1$ es el resultado de aplicar el operador `constraint` a `P1`. Mergeamos los modelos $Constraint\_P1$ y $EscenarioMinimizado$ utilizando el operador `+cr`.

Por último, intentamos ver si el resultado de aplicar `+cr` es verdaderamente refinamiento de $Constraint\_P1$ y $EscenarioMinimizado$. Esto lo llevamos a cabo utilizando la ventana de chequeo de refinamiento que provee MTSA (sección 4.2). Como resultado de estas operaciones, obtuvimos un modelo, al que llamaremos $FinalIteracionI$, que es refinamiento de $Constraint\_P1$ y $EscenarioMinimizado$ y, por lo tanto, verifica la propiedad `P1`.

5.3. Diseño, verificación y extensión. Iteración II

En esta iteración intentaremos validar e incorporar las propiedades restantes y obtener un modelo del sistema de control de bombeo que cumpla con los requerimientos.

Antes de comenzar con el análisis de las propiedades `P2`, `P3` y `P4` sobre el modelo $FinalIteracionI$, le aplicamos el operador de minimización sin éxito. El modelo y su versión minimizada tienen la misma cantidad de estados (29).

Continuando con el caso de estudio verificamos si $EscenarioMinimizado$ satisface la propiedad `P2`. La respuesta de MTSA fue negativa (i.e $EscenarioMinimizado^{+} \not\models P2 \wedge EscenarioMinimizado^{-} \not\models P2$). Tomando la información de la salida de MTSA (figura 5.3) obtuvimos dos trazas que violan la propiedad. Para el caso optimista es $\pi = methAppears \rightarrow switchOff \ldots$ y para el caso pesimista es $\pi' = medWater \rightarrow highWater \rightarrow lowWater \rightarrow methAppears \rightarrow switchOff \ldots$.

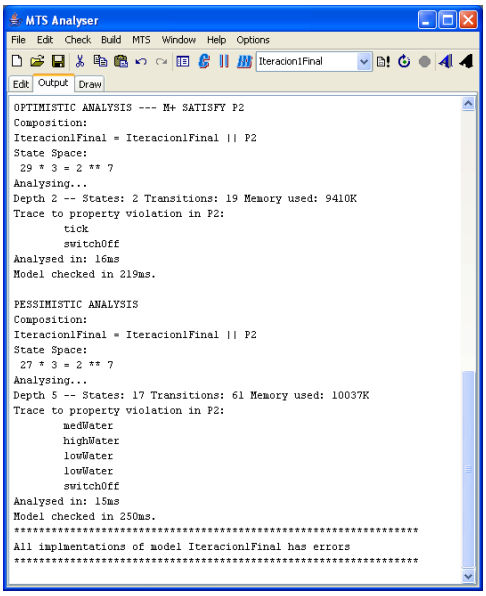

Figura 5.3: Salida de la validación de `P2` sobre $Iteracion1Final$

Analizamos las trazas dadas por MTSA y el modelo $EscenarioIT1$. Entendimos que la traza requerida π' describe un comportamiento que no es el esperado para el sistema. Luego de la acción $highWater$ no puede pasar que aparezca una transición etiquetada con la acción $lowWater$ debido a que justamente se espera que luego de detectar nivel de agua alto se prenda la bomba y recién en ese momento el nivel comience a bajar. En la figura 5.3 vemos parte del código FSP del escenario. En la línea en que se define el modelo $Q1$ vemos que hay una transición requerida por $highWater$ hacia el modelo $Q2$ (`highWater ->Q2`) y luego $Q2$ admite como requerida una transición por $lowWater$ hacia $Q3$.

Para corregir este problema de diseño decidimos prohibir la transición $Q2 \xrightarrow[r]{lowWater}$

```
EscenarioIter1 = Q0,
Q0 = ({switchDLOff, tick} -> Q0 |medWater -> Q1
|methAppears -> Q20),
Q1 = (lowWater -> Q0 |{switchDLOff, tick} -> Q1
|highWater -> Q2 |methAppears -> Q7),
Q2 = (switchOn -> Q3  | lowWater ->Q4),
Q3 = ({switchDLOff, tick} -> Q3 |medWater -> Q4
|methAppears -> Q21),
```

⋮

Figura 5.4: Fragmento de código de EscenarioIter1.

$Q3$ en $EscenarioIT1$ debido a que no tenía sentido según nuestro entendimiento del problema. De esta forma obtuvimos el modelo $EscenarioCorregido$ cuyo código mostramos en el apéndice 5.

Al igual que en la primera iteración al modelo $EscenarioCorregido$ le aplicamos el operador `abstract` y obtuvimos un modelo de 44 estados. Despues de minimizarlo conseguimos un modelo con 22 estados. Luego chequeamos nuevamente `P1` y obtuvimos resultado *maybe* por lo tanto dado que era posible, mergeamos $EscenarioCorregido$ y `P1` y conseguimos el modelo $IT2v1$ (i.e Iteración 2, versión 1).

Sobre $IT2v1$ chequeamos las propiedades `P3` y `P4` ambas con resultado *maybe*, por lo tanto las mergeamos (`IT2v1 +cr P1 +cr P2`) para obtener un modelo que acepte únicamente implementaciones que describan el comportamiento de las cuatro propiedades así como también del escenario original. $FinalIteracion2$ modelo final de este merge

Componiendo en paralelo los modelos $FinalIteracion2$, $SensorNivelAgua$ y $SensorMetano$ obtuvimos el modelo $SistemaDeControl$ que describe el comportamiento de $Iteracion2Final$ y el comportamiento del ambiente ($SensorNivelAgua||SensorMetano$).

A $SistemaDeControl$ le aplicamos el operador de minimización sin lograr reducir la cantidad de estados del modelo. Por lo tanto, el modelo más refinado que es posible lograr con los datos con los que comenzamos es $SistemaDeControl$.

En este caso de estudio mostramos la utilidad de las técnicas que desarrollamos, para ayudar al diseñador a construir, elaborar y analizar MTSs. Vimos como utilizando el operador *abstract* aplicado a un escenario conseguimos una visión más general en la cual no prohibimos comportamiento sino que lo incorporamos como indefinido. Además mostramos la utilidad que puede tener el operador *constraint* para lograr, a partir de una propiedad, un modelo que la represente siendo el más abstracto posible. Como parte del proceso de modelado redujimos la cantidad de estados de los modelos intermedios utilizando el operador de minimización (`minimal`). Luego aprovechamos la operación de merge (`+cr`) para restringir las implementaciones de un modelo a un conjunto que cumpla con la propiedad deseada. Por último, utilizamos la vista de visualización de modelos en MTSA para seguir y analizar las trazas de los modelos y ganar entendimiento sobre ellos e incluso corregir problemas.

Capítulo 6

Conclusiones

6.1. Resumen de contribuciones

En este trabajo hemos estudiado y trabajado sobre diversos temas relacionados con la manipulación de MTS.

- Definimos la relación de equivalencia entre MTS. Trabajamos sobre la definición formal de minimización, determinización y análisis de deadlocks sobre MTS. Además desarrollamos los algoritmos correspondiente a cada una de ellas.
- Estudiamos las definiciones de MTS optimista (M^+), pesimista (M^-) y model checking sobre MTS . Encontramos algunos errores en ellas y las corregimos.
- Continuando el trabajo [29] desarrollamos las operaciones *abstract* y *constraint*. Las cuales dan soporte y ayuda a los diseñadores y modeladores en la tarea de construir modelos mas rápida y ´fácil.
- Implementamos algoritmos para la composición en paralelo de MTSs y para los operadores de merge +CA y +CR [30].
- Hemos desarrollado Modal Transition Systems Analiser. En ella implementamos los algoritmos para componer en paralelo, determinizar y minimizar MTS. Así como también incluye la funcionalidad de visualización, animación y manipulación en general de MTS. Como parte de la extensión del lenguaje FSP, incluimos los operadores *abstract* y *constraint*. Finalmente MTSA permite realizar model checking y análisis de deadlock sobre MTS.

Por último realizamos un caso de estudio en el cual pusimos a prueba la utilidad de las técnicas y operaciones que desarrollamos.

6.2. Trabajo relacionado

En este trabajo hemos desarrollado y extendido diferentes cuestiones teórico-prácticas relacionadas con la ingeniería del software y el modelado de comportamiento.

Modelos de comportamiento: El modelado de sistemas es una técnica muy útil en el proceso de validar tanto diseños como requerimientos.

Según Clarke [5] la especificación es el proceso de describir a los sistemas y a sus propiedades. La especificación formal usa un lenguaje sintaxis y semántica definidas matemáticamente. El tipo de propiedades que más éxito ha tenido en conjunto con la especificación ha sido el de las propiedades de comportamiento.

Hay diferentes métodos y aproximaciones a la especificación de sistemas. En particular existe mucho trabajo realizado para modelar comportamiento concurrente de sistemas. Hoare en [14] da un lenguaje para lograr la composición en paralelo de procesos secuenciales (CSP).

Milner desarrolló el calculo de sistemas de comunicación (CCS) [27]. El lenguaje incluye primitivas para describir la composición en paralelo, la elección de alternativas y la restricción de alcance.

Los diagramas de estado (statecharts) fueron desarrollados por Harel en [11]. Las expresiones del lenguaje son interpretadas como LTS. Por lo tanto la bisimulación es utilizada como semántica para la equivalencia de modelos.

Los Labelled Transition Systems fueron descriptos por Keller en [20]. En ese trabajo Keller dió dos formalismos para modelar el procesamiento paralelo en sistemas. El primero fue el llamado modelo conceptual abstracto, formalismo que definió los LTS. Además Clarke da una forma de modelar programas paralelos que llamo *modelo de programas paralelos*. El trabajo da una definición de principio de inducción sobre LTS. Además se formaliza el concepto de deadlock y se estudia la forma y la dificultad de encontrar deadlocks.

En [22], Krüger muestra como los requerimientos (basados en escenarios) capturados en etapas tempranas utilizando Message Sequence Charts (MSCs) pueden ser traducidos en descripciones basadas en estados por ejemplo en Diagramas de transición de estados (StateCharts).

Modelos Parciales: Los modelos descritos anteriormente asumen descripciones con información completa respecto del funcionamiento esperado de los sistemas a modelar.

La noción de extender un sistema de transiciones (e.g LTS) con una segunda relación de transición que capture el comportamiento desconocido del comportamiento fue propuesta originalmente por [24], y en paralelo por [6]. Larsen y Thomsen introdujeron los MTS como solución la limitación de completitud de los LTS y probaron que la lógica de Hennessy-Milner [13] caracteriza el refinamiento de MTS.

Huth en [17] introduce los Kripke MTS (KMTSs), los cuales son una versión basada en estados de los MTS. Los KMTSs tienen dos relaciones de transición, como en un MTS pero a diferencia de ellos los KMTSs tienen los estados etiquetados con un conjunto de proposiciones con semántica tri-valuada.

Model Checking

Luego de la formalización de model checking dada en [4] y la creación de los MTS [24]. Se intentó lograr una definición de model checking sobre MTS. Las primeras nociones fueron dadas 'por Huth en [18]. En esta tesis tomamos el trabajo realizado en [30] y corregimos las definiciones de M^+, M^- y model checking para MTS.

Síntesis de modelos: La síntesis de modelos de comportamiento a partir de descripciones basadas en escenarios o especificaciones de requerimientos no solo ayuda a reducir significativamente el esfuerzo en la construcción de modelos sino que también proveen un puente entre la visión del análisis de los requerimientos y el razonamiento del diseño del sistema al nivel de la arquitectura.

En [29] se propone una nueva técnica para la construcción de modelos de comportamiento parcial, en particular MTS, a partir de una combinación de propiedades de "'safety"' y escenarios.

Krüger [22] muestra como sintetizar diagramas de transición de estados (StateCharts) a partir de Message Sequence Charts (MSCs).

Minimización: Trabajamos sobre un algoritmo de minimización para MTS. Estudiamos los propuestos en [15] y [2]. En [15] se propone un algoritmo de minimización de maquinas de estado basado en estructuras llamadas de cociente. Se crean nuevas estructuras con estados representados por las clases de equivalencia de los estados originales.

El algoritmo planteado en [2] toma la noción de estructuras cociente pero cambia la estructura de datos sobre la que trabaja y la semántica de la minimización. No toma equivalencia de trazas sino que trabaja sobre la noción de equivalencia por simulación.

Nosotros trabajamos tomando la estructura cociente y tomando como semántica la equivalencia por simulación pero sobre MTS por lo que hemos extendido ambos algoritmos.

Merge: Hussain y Huth [16] estudiaron el problema de encontrar un refinamiento común entre múltiples MTS. Ellos pusieron el foco en la complejidad de los procedimientos como calculo de consistencia, satisfacibilidad y validez.

En el trabajo [30] se definen los operadores `+CR` y `+CA` y se dan sus fundamentos teóricos formales.

En [1] Brunet consigue modelos que son mas generales a los propuestos por Hussain y Huth dado que es posible mergear estos modelos aún con diferentes vocabularios y transiciones τ. A su vez estos son menos generales en cuanto a que Hussain y Huth pueden trabajar con restricciones híbridas (e.g., restringir el numero de estados en los

cuales es evaluada una proposición) y calcular el conjunto de todos los pares consistentes.

6.3. Trabajo a futuro

A raíz del trabajo que realizamos en esta tesis hemos abierto varias líneas de investigación para nuevos trabajos de investigación:

- Chequeo de Propiedades de Progreso: Hemos trabajado ampliamente en el model checking de MTS pero conseguimos hacerlo únicamente para propiedades de safety. No tenemos claro que significa chequear progreso en un MTS. En principio entendemos que el model checking basado en las versiones optimista y pesimista de un MTS, no funciona para verificar propiedades de progreso.
- Performance de la herramienta: Los algoritmos que implementamos en la herramienta son correctos pero son bastante ineficientes, por lo que deberíamos trabajar sobre ellos y lograr que la herramienta sea lo mas performante posible.
- Mejora de lenguaje gráfico de modelado: Actualmente MTSA solo permite la construcción de modelos via FSP, síntesis u operadores composicionales. Sería muy interesante contar con una herramienta visual para edición de modelos. Una posible línea de investigación podría ser la integración con herramientas existentes para modelado gráfico.
- Estudiar análisis de modelos tomando trazas Fair: La definición de model checking que desarrollamos en este trabajo es válida siempre y cuando no se tomen las trazas fair del modelo. El model checking basado en las versiones optimista y pesimista de un modelo no se comporta como esperamos utilizando trazas fair. En la figura 6.3 mostramos el modelo $\mathcal{Z}$ que si tomamos trazas fair satisface la formula `[]<>b` (siempre eventualmente ocurre b). Según nuestro trabajo si realizaramos un chequeo de la propiedad `[]<>b` sobre $\mathcal{Z}$ MTSA devuelve verdadero. Pero podemos existen implementaciones de $\mathcal{Z}$ que no satisfacen la formula, por ejemplo la que mostramos en la figura 6.3. Esto se debe a que el hecho de tomar trazas fair sobre el modelo optimista nos hace analizar menos trazas y por ende el análisis no es concluyente.

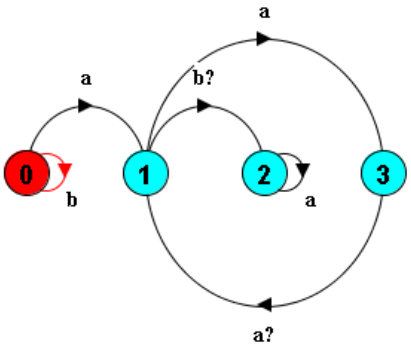

Figura 6.1: Modelo $\mathcal{Z}$

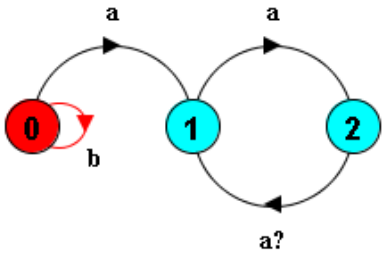

Figura 6.2: Implementación de $\mathcal{Z}$

Bibliografía

[1] G. Brunet. "A Characterization of Merging Partial Behavioural Models". Master's thesis, Univ. of Toronto, 2006.

[2] D. Bustan and O. Grumberg. Simulation based minimization. In *Conference on Automated Deduction*, pages 255–270, 2000.

[3] M. Chechik, B. Devereux, S. Easterbrook, and A. Gurfinkel. "Multi-Valued Symbolic Model-Checking". *ACM TOSEM*, 12(4), October 2003.

[4] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.

[5] E. Clarke and J. Wing. "Formal Methods: State of the Art and Future Directions". *ACM Computing Surveys*, 28(4):626–643, December 1996.

[6] D. Dams. *Abstract Interpretation and Partition Refinement for Model Checking*. PhD thesis, Technische Universiteit Eindhoven, 1996.

[7] D. Fischbein and S. Uchitel. "On Consistency and Merge of MTS". In *Submitted to International Workshop on Living with Uncertainty, ASE'07*, 2007.

[8] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 1995.

[9] D. Giannakopoulou and F. Lerda. From states to transitions: Improving translation of LTL formulae to Buchi automata. *Proceedings of the 22nd IFIP WG*, 6, 2002.

[10] D. Giannakopoulou and J. Magee. "Fluent Model Checking for Event-Based Systems". In *ESEC/FSE'03*, 2003.

[11] D. Harel. "StateCharts: A Visual Formalism for Complex Systems". *Science of Comp. Program.*, 8:231–274, 1987.

[12] D. Harel, H. Kugler, and A. Pnueli. "Synthesis Revisited: Generating Statechart Models from Scenario-Based Requirements". In *FMs in Soft. and Systems Modeling*, pages 309–324, 2005.

[13] M. Hennessy and R. Milner. Algebraic laws for nondeterminism and concurrency. *Journal of the ACM (JACM)*, 32(1):137–161, 1985.

[14] C. Hoare. *Communicating Sequential Processes.* Prentice-Hall, New York, 1985.

[15] J. Hopcroft and J. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison Wesley, 1979.

[16] A. Hussain and M. Huth. "On Model Checking Multiple Hybrid Views". In *1st Int. Symp. on Leveraging Applications of FMs*, pages 235–242, 2004.

[17] M. Huth, R. Jagadeesan, and D. Schmidt. "A Domain Equation for Refinement of Partial Systems". *Mathematical Structures in Computer Science*, 2003. (In press.).

[18] M. Huth and S. Pradhan. "Model-Checking View-Based Partial Specifications". *Electronic Notes in Theoretical Computer Science*, 45, November 2001.

[19] R. Kazhamiakin, M. Pistore, and M. Roveri. "Formal Verification of Requirements using SPIN: A Case Study on Web Services". In *SEFM'04*, 2004.

[20] R. Keller. "Formal Verification of Parallel Programs". *CACM*, 19(7):371–384, 1976.

[21] J. Kramer, J. Magee, and M. Sloman. "CONIC: an Integrated Approach to Distributed Computer Control Systems". *IEE Proceedings*, 130(1):1–10, 1983.

[22] I. Krüger, R. Grosu, P. Scholz, and M. Broy. From mscs to statecharts. In *Distributed and Parallel Embedded Systems*, 1999.

[23] A. Kucera and R. Mayr. Why is simulation harder than bisimulation? In *CONCUR '02: Proceedings of the 13th International Conference on Concurrency Theory*, pages 594–610, London, UK, 2002. Springer-Verlag.

[24] K. Larsen and B. Thomsen. "A Modal Process Logic". In *LICS'88*, pages 203–210, 1988.

[25] J. Magee and J. Kramer. *"Concurrency - State Models and Java Programs"*. John Wiley, 1999.

[26] R. Milner. "An Algebraic Definition of Simulation between Programs". In *Proceedings of 2nd International Joint Conference on Artificial Intelligence*, pages 481–489, 1971.

[27] R. Milner. *A Calculus of Communicating Systems.* Springer-Verlag, 1980.

[28] C. Ponsard, P. Massonet, A. Rifaut, J. Molderez, A. van Lamsweerde, and H. T. Van. "Early Verification and Validation of Mission-Critical Systems". In *FMICS'04*, 2004.

[29] S. Uchitel, G. Brunet, and M. Chechik. "Behaviour Model Synthesis from Properties and Scenarios". In *ICSE'07*, 2007.

[30] S. Uchitel and M. Chechik. "Merging Partial Behavioural Models". In *FSE'04*, pages 43–52, 2004.

[31] P. Wolper and P. Godefroid. Partial-order methods for temporal verification. In *CONCUR '93: Proceedings of the 4th International Conference on Concurrency Theory*, pages 233–246, London, UK, 1993. Springer-Verlag.