Universidad de Buenos Aires
Facultad de Ciencias Exactas y Naturales
Departamento de Computación

Síntesis concurrente de controladores para juegos definidos con objetivos de Generalized Reactivity(1)

Tesis presentada para optar al título de
Licenciada en Ciencias de la Computación

María Virginia Brassesco

Director: Nicolás D'Ippolito

Buenos Aires, Septiembre de 2017

SÍNTESIS CONCURRENTE DE CONTROLADORES PARA JUEGOS DEFINIDOS CON OBJETIVOS DE GENERALIZED REACTIVITY(1)

Se describe un algoritmo concurrente para resolver juegos de tipo: *Generalized Reactivity (1)*, con aplicación a síntesis de controladores. El algoritmo esta basado en Jurdzinski's *Small Progress Measures*. Utiliza estructuras de acceso concurrente para optimizar las lecturas y cuenta con bloqueos de escritura. El siguiente trabajo muestra la implementación del algoritmo de estabilización de punto fijo de forma concurrente, para cálculo de *ranking*. La implementación se realizó y verificó sobre la herramienta MTSA [DL15]. Se analizaron implementaciones conocidas basadas en *Small Progress Measures* [HPK11] [vdPW08a]. Se adaptó la implementación para poder calcular *ranking* utilizando los teoremas conocidos de equivalencias entre juegos de paridad y GR1 [GWT02]. Se reportan comparaciones sobre los resultados obtenidos.

Palabras claves: Concurrencia, Diseño sintetizado, LTS, Controladores, Reactividad Generalizada, Pequeña medida de progreso.

CONCURRENT SYNTHESIS FOR CONTROLLERS DESIGNED WITH GENERALIZED REACTIVITY (1)

This work describes a concurrent algorithm to solve Generalized Reactivity (1) games with its application to controller synthesis. The algorithm is based on Jurdzinski's Small Progress Measures (SPM). It uses concurrent structures to optimize reads and some write locks. The following work shows the implementation of the fixed point recucurrent algorithm to compute the ranking in a concurrent way. The implementation was done and verified on MTSA tool [DFCU08b]. Different known implementations based on SPM where analized [vdPW08a]. The implementation was adapted to calculate the ranking using known theorems based on GR1 and parity games equivalencies [GWT02]. A result comparisson was also reported.

Keywords: Concurrent, Design Synthesis, Controllers, Generalized Reactivity, Small Progress Measures.

AGRADECIMIENTOS

A mi familia por apoyarme siempre y llevarme por el camino del estudio y el progreso.

A mis amigas que se bancan que hable de computación en nuestras reuniones.

A mi director por darme la oportunidad de trabajar en un laboratorio en Japón bajo la supervisión de Kenji Tei. Y por seguir apoyándome en mi desarrollo académico.

A Nir Piterman por cuestionar todos mis resultados para conseguir los mejores.

A Ezequiel que a fuerza de ponerme cara de "ezequiel" logró hacerme programar cosas que funcionan con tests que tengan sentido.

A todo LAFHIS por el espacio y las charlas motivadoras.

A mi presidenta, Cristina Fernández, por garantizarme el contexto social y económico para que pudiese estudiar y desarrollarme en la universidad pública.

Al Centro de Cómputos de Alto Rendimiento (CeCAR) por concederme acceso de uso computacional de sus recursos que me permitieron ejecutar todas las pruebas expuestas en este trabajo.

Todo pasa y todo queda, pero lo nuestro es pasar, pasar haciendo caminos, caminos sobre la mar.

Índice general

1.. Introducción 1
1.1. Motivación 1
1.2. Contribuciones 2

2.. Fundamentos teóricos 4
2.1. El Mundo y la Máquina 4
2.2. Sistema de Transición Etiquetado (Labelled Transition System) 5
2.3. Lógica Lineal Temporal de Fluents (Fluent Linear Temporal Logic) 6
2.4. Problemas de síntesis de controladores 7
2.5. Juegos de dos jugadores 8
2.6. Resolución de problema de control 9
2.6.1. Control LTS SGR(1) a juegos GR(1) 9
2.6.2. Traduciendo la estrategia a un Controlador LTS 10
2.6.3. Algoritmo Secuencial SGR(1) 11
2.7. Procesos de estados Finitos (Finite State Process) 13
2.8. Modelado del Ambiente para controladores 15
2.9. Modelado de los Objetivos (o misión) de los controladores 15
2.10. Síntesis de un controlador 16
2.10.1. Seguimiento de estabilización 17

3.. Algoritmo concurrente SGR(1) 20
3.1. Acceso concurrente a objetos compartidos 20
3.1.1. Repensar la Concurrencia 20
3.1.2. Sincrónico, concurrente y paralelo 20
3.1.3. Administración de acceso a Memoria 21
3.2. Pseudocódigo de la implementación 23
3.3. Intuición de correctitud 24
3.3.1. Definiciones para mostrar correctitud 24
3.3.2. Explicación de la intuición 26
3.4. Correctitud del Algoritmo Concurrente 26

4.. Implementación 28
4.1. Sobre *Modal Transition System Analyser* 28
4.1.1. Descarga y Compilación 28
4.2. Modo de uso de MTSA 29
4.2.1. Interfaz Gráfica 29
4.2.2. Consola 29
4.3. Diagramas de Secuencia Implementación Concurrente 30
4.3.1. Inicio de ejecución. Principal 30
4.3.2. Creación de trabajadores, primera sincronización 31
4.3.3. Forma de consumo y escritura de cada trabajador 31
4.3.4. Análisis de estados de la Cola 32
4.3.5. Actualización de ranking 32

4.4. Detalle de clases implementadas 32
4.4.1. package ltsa.MultiCore 35
4.4.2. package MTSSynthesis.ar.dc.uba.util 35
4.4.3. package MTSSynthesis.controller.game.model 37
4.4.4. package MTSSynthesis.controller.game.model 37

5.. Evaluación de la implementación 39
5.1. Metodología 39
5.2. Análisis previo 40
5.3. Casos de Estudio 41
5.3.1. Árbol 41
5.3.2. Clique 42
5.3.3. Escalera 42
5.3.4. Centro médico 43
5.4. Implementación de Evaluación 43
5.4.1. Detalles de hardware 43
5.4.2. Código de pruebas 43
5.4.3. Profiling 44

6.. Resultados 45
6.1. Comparación de tiempos 45
6.1.1. Árbol 45
6.1.2. Clique 47
6.1.3. Escalera 47
6.1.4. Centro Médico y Ruteo 49
6.2. Uso de CPU 49
6.3. Tiempos de procesamiento 51
6.4. Iteraciones requeridas para alcanzar estabilidad 52

7.. Conclusiones 53
7.1. Análisis final 53
7.2. Trabajo futuro 53

Apéndice 55

A.. Tablas de resultados 56
A.1. Escalera 10.000 estados 57
A.2. Clique 1000 estados 58
A.3. Árbol 100 ramas x 20 longitud 59
A.4. Árbol 20 ramas x 100 longitud 60
A.5. Medical 40 enfermedades 61
A.6. Transporte 62

1. INTRODUCCIÓN

1.1. Motivación

Cuando trabajamos en ingeniería de requerimientos, nuestra tarea es relevar y documentar los objetivos y el funcionamiento del ambiente, para elaborar un conjunto de requerimientos cuya máquina a construir debe cumplir. Teniendo en cuenta esta frase, podemos formalizarla de la siguiente forma: si R son los Requerimientos de la máquina, D las asunciones del dominio y G los objetivos que la máquina deberá satisfacer, podemos formular la siguiente ecuación: $R, D \models G$.

Entonces, un problema clave de la ingeniería de requerimientos puede ser visto como un problema de síntesis, tal que dado un modelo del entorno y la especificación de los objetivos, permite generar automáticamente un modelo con la descripción del comportamiento operacional. Más específicamente, todas las trazas controlables (def. 2.4) del sistema deben satisfacer el objetivo pedido. La técnica que resuelve esta ecuación mediante los modelos mencionados es llamada síntesis de controladores y está siendo estudiada exhaustivamente en varios aspectos de la ingeniería de los requerimientos. Es sabido además que la síntesis se ha usado también para generar controladores para robots [KGFP07].

Por lo tanto, un problema de control consiste en generar automáticamente una máquina que restrinja la ocurrencia de eventos controlables -descritos en lógica temporal- basado en los eventos del mundo que se producen. Es decir, teniendo la especificación de un ambiente, asunciones, objetivos y un conjunto de acciones controlables, podemos definir una máquina cuyo comportamiento concurrente con el ambiente satisfaga las asunciones y los objetivos del sistema. [PR89, DIBPU13]

La construcción de dichos modelos es una de las principales herramientas en el diseño de sistemas concurrentes, debido a que nos facilita la detección de errores de diseño en las primeras etapas de desarrollo. Por otro lado, estos modelos de comportamiento pueden resultar complejos de construir. Utilizar la técnica de síntesis anteriormente mencionada facilita la construcción tomando modelos pequeños y propiedades lógicas, que suelen ser más sencillas de especificar y de validar. Las propiedades se suelen expresar como formulas de lógica temporal. En sistemas basados en eventos, los estados caracterizan el comportamiento que se origina a partir de estos estados. En éste contexto, resulta más natural utilizar la noción de fluentes para poder combinar el estado y la acción. Según se definen en [GM03] permiten especificar propiedades del mundo que valen a partir de la ocurrencia de cierta acción y dejan de cumplirse ante la ocurrencia de otra acción.

Para generar automáticamente un controlador lo que se hace es traducir la especificación del ambiente y los objetivos en juegos de dos jugadores con condiciones de ganada apropiados [GTW02]. Estos juegos están basados en grafos donde se define: una arena (V_0, V_1, E) y una condición de ganada. En esta tesis consideramos los juegos infinitos de 2 jugadores, de suma cero e información perfecta. Un jugador se moverá representando al ambiente y el otro: el controlador, tratará de cumplir el objetivo descrito en la condición de ganada. El perder representará que no es posible encontrar un controlador. Como se mencionó más arriba, toda traza que resulte de la composición del ambiente con el controlador obtenido, satisfará los requerimientos.

En escenarios de sistemas reactivos donde alguno de los componentes puede fallar,

se podrían considerar las fallas como adversario (controlados por el ambiente). Debemos tener en cuenta que ésta opción podría resultar en un problema sin solución, aún cuando exista un controlador posible. La configuración propuesta en [DIBPU11] resuelve esto considerando una noción muy restrictiva de falla y estableciendo condiciones justas en los casos de falla.

Las distintas condiciones de ganada dependen del tipo de juego. Las condiciones más simples son las que piden alcanzar cierto nodo en el grafo o visitar nodos que satisfagan una propiedad. Pero podemos agregar no solo llegar a cada nodo, sino visitarlo infinitamente. O bien, evitar ciertos nodos y visitar otros -también de manera infinita. Como se puede ver, las condiciones de ganada puede ser tan complejas como queramos. Nombraremos a cada juego de acuerdo a su condición de ganada. Por ejemplo, los juegos GR(1) (o Streett(1)), son los que cumplen: $\bigwedge_i \Box\Diamond p_i \Rightarrow \bigwedge_j \Box\Diamond q_j$

La síntesis de controladores es un problema de alta complejidad algorítmica. Para decir quién gana un nodo en juegos de pares es UP $\bigcap$ co-UP [Jur98]. Esto quiere decir que aún no fue hallada una solución genérica de tiempo polinomial para éste problema. Más aún, hay que tener en cuenta que el juego construido a partir del ambiente y los objetivos de ganada crecerá exponencialmente. Depende directamente de la cantidad de objetivos propuestos.

En lo que respecta al "hardware", recordemos que en 1965 Gordon Moore estableció una ley que durante 50 años fue uno de los pilares fundamentales de la electrónica. Sin embargo, los límites del silicio han comenzado a ralentizar la evolución de la tecnología [BVW13]. Si bajamos más el tamaño de los transistores, llegará un punto en el que estos dejarán de funcionar y de comportarse como se espera. Las empresas están realizando investigaciones utilizando nuevos materiales y nuevos procesos de fabricación.

Según las estimaciones del ITRS (International Technology Roadmap for Semiconductors), el aumento de la escala de integración y, por tanto, la evolución de los chips se está produciendo, desde 2010, cada 3 años y si bien la capacidad de los chips ha seguido aumentando, en gran medida, ha sido así gracias al uso de tecnologías con múltiples cores (núcleos).

Luego, la tendencia actual es desarrollar unidades de procesamiento, que al ser utilizadas en conjunto y de manera concurrente nos permitan procesar un mayor flujo de información en menor tiempo. Para poder aprovechar éstas nuevas capacidades los algoritmos que antes corrían de manera secuencial deberán adaptarse para ejecutar de manera concurrente, con o sin bloques sincrónicos.

1.2. Contribuciones

A pesar de la necesidad de mejorar los tiempos de síntesis y del giro a nivel de procesamiento hacia tecnologías de múltiples núcleos, se encuentra poca investigación en el área de paralelización de síntesis de juegos. Se conoce el trabajo realizado en [vdPW08b] que implementa y evalúa un sintetizador basado en *small progress measures* [Jur00] (SPM) sobre multi-core.

Es sabido que para seguir agrandando la capacidad de procesamiento en las computadoras actuales y así intentar una alternativa para combatir la ley de Moore, la tendencia es utilizar más núcleos que en su conjunto tienen una mayor capacidad. El algoritmo implementado y la idea original de resolución de problema de control GR(1) no hace uso de ésta modalidad. Los algoritmos suelen pensarse como una secuencia ordenada de pasos a

repetir una y otra vez encadenados. No siempre es posible ejecutar dos instrucciones en desorden o a la par y seguir obteniendo el mismo resultado.

La transformación de un algoritmo secuencial a uno concurrente es una tarea muchas veces desafiante para el diseñador de software. Si bien la intuición nos sugiere pensar que al ejecutar en paralelo lograremos mejores tiempos, esto esta altamente relacionado con cuán interconectados se encuentran los datos a usar. El acceso al mismo objeto a la vez por más de una tarea puede ocasionar colisiones de acceso que ralentizan el total de ejecución. Por lo tanto, ejecutar en paralelo, no siempre garantiza resultados con cómputos más rápidos.

En este trabajo se ha diseñado e implementado el primer algoritmo paralelo para el cómputo de soluciones a problemas de control con condiciones GR(1). Más específicamente, el trabajo se basa en el diseño de un algoritmo concurrente para el computo de estabilización de punto fijo en juegos combinatorios, basados en turnos, de suma cero con condición de ganada GR(1). El algoritmo permite la ejecución concurrente de la actualización de valores (rankings, ver sección 2.11) en todo el espacio de estados de forma sincrónica y con bloqueo de escritura sobre cada estado. El algoritmo ha sido implementado como una extensión de la herramienta MTSA [DL15] ampliamente usada para resolver problemas de control GR(1). Se ha evaluado en diversos casos de estudio mostrando una variedad de resultados.

MTSA se puede utilizar para sintetizar controladores definidos usando fórmulas en lógica temporal para describir el comportamiento del ambiente y los objetivos del controlador. La síntesis se realiza simulando ganar un juego de dos jugadores. Los objetivos se definen con asunciones del dominio y garantías a satisfacer. En este trabajo se utilizaron juegos descritos con objetivos basados en fórmulas GR [PPS06a].

$\wedge_i \square\Diamond A_i \Rightarrow \wedge_j \square\Diamond G_j$

En suma, parece razonable continuar y extender el análisis del algoritmo de síntesis para disminuir los tiempos de obtener controladores que cumplan los objetivos del sistema conjuntamente con el entorno. La implementación sugerida en [HKP12] desarrolla una alternativa de procesamiento concurrente que no depende de una pre-configuración del ambiente o de los requerimientos para realizar la síntesis. Buscamos analizar la efectividad de ese algoritmo para sintetizar juegos con condiciones de ganada *Generalized Reactivity(1)*, y comprender si en este caso también se cumple el postulado de mejora lineal observado para juegos de pares.

Distribución de temas

Juegos y concurrencia computacional son presentados en la sección de conocimientos teóricos: 2 y 2.6. En la sección 2.8 se pueden ver ejemplos de uso teóricos. Antes de describir la implementación se detallan datos relevantes sobre programación concurrente, sección 3.1. Luego se puede ver en detalle la implementación en la sección 4. A continuación, en la misma sección, se describe el uso de la herramienta donde se implementó y evaluó el trabajo. En la sección 5 se describe la forma de evaluar la hipótesis de éste trabajo y se encuentran los casos de prueba ejecutados. Por último, en la sección 6 se muestra los gráficos de los tiempos obtenidos usando concurrencia y en la sección 7 se realiza una comparación entre los resultados originales y los nuevos en el cálculo del ranking.

2. FUNDAMENTOS TEÓRICOS

2.1. El Mundo y la Máquina

Los primeros conceptos que detallaremos estarán involucrados con la ingeniería de los requerimientos. Los puntos de vista más relevantes son los de Zave y Jackson ([ZJ97, Jac95a, Jac95b]) por un lado, y los de Letier y Van Lamsweerde ([VLL00, VL01]) por el otro. Ambos puntos de vista distinguen a los problemas del *Mundo* y las soluciones de la *Máquina* como fundamentales para reconocer si las operaciones de la máquina solucionan los problemas planteados en el mundo. De hecho, el efecto de la máquina en el mundo y las suposiciones que hacemos acerca de este mundo son fundamentales para el proceso de toma de requerimientos. En el lado del mundo definimos una serie de problemas que existen en el mundo real; que serán solucionados al construir una máquina. Fácilmente podremos notar que existen componentes en la máquina que interactúan directamente con el mundo siguiendo normas y procesos conocidos. Éstas, forman parte de la intersección entre el mundo y la máquina. Por ejemplo un taladro, un brazo robótico o las reglas de procesamiento para cada elemento que entra en una línea de producción (ver la Figura 2.1).

Así mismo, es esperado que la máquina proponga una solución al problema. Por ejemplo, en la Figura 2.1 podemos ver que la célula de producción debe procesar cada producto, sólo si está disponible en la bandeja de entrada en ese momento. Con la sentencia $EnBandeja[p] \rightarrow get.Bandeja[p]$ mostramos que se espera que el brazo del robot sólo podrá tomar los productos de la bandeja cuando estén listos. Finalmente, los fenómenos compartidos entre el mundo y máquina, es decir, los que se encuentran en la intersección, representa a la *interfaz*, donde la máquina interactúa con el mundo. También, podemos definir a los fenómenos del mundo como el *modelo del entorno* ya que el conjunto de éstos describen los eventos que suceden en el mundo real.

Las sentencias que detallan los distintos fenómenos, tanto en el mundo como en la máquina pueden variar en *alcance* y en *forma* [PM95, Jac95a]. Además, éstas pueden estar en modo *indicativo* u *optativo*. En otros trabajos, como en [vL09], las sentencias utilizadas son *descriptivas* y *prescriptivas*.

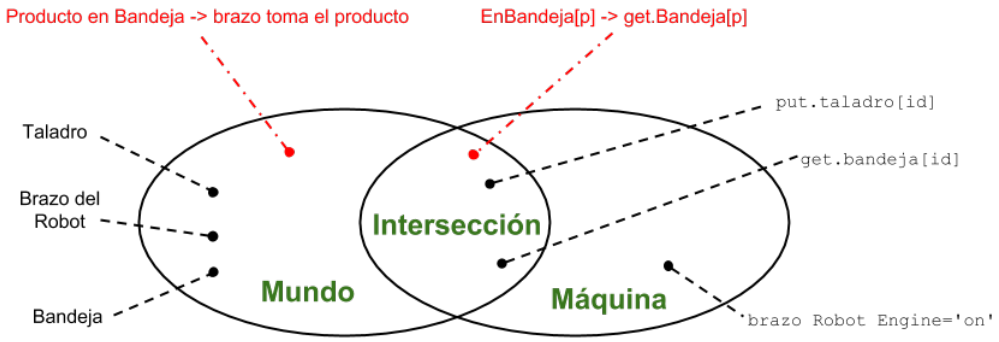

Fig. 2.1: El Mundo y la Máquina

- Sentencias descriptivas: representan propiedades que son independientes de cómo se comporta el sistema. Se usan en modo *indicativo*. No pueden ser cambiadas ni removidas. Por ejemplo, podrían decirnos que hay distintos tipos de taladros pero todos ellos responden a los mismos comandos.

- Sentencia prescriptivas: afirman propiedades deseables que pueden estar presentes o no. Deben estar aplicadas por los componentes del sistema. Normalmente, pueden cambiar fortaleciéndose o debilitándose, o incluso pueden ser eliminadas. Por ejemplo, en la fábrica, querríamos que se ensamblen productos y luego se coloquen en alguna bandeja de salida porque no es deseable que los productos se acumulen en las máquinas ensambladoras.

Anteriormente, fue mencionado que los estados pueden variar en su alcance. Ambos tipos de sentencias pueden referirse a características de la máquina que no son compartidas por el mundo. En otras ocaciones, las sentencias pueden referirse a fenómenos compartidos por el mundo y la máquina. Más precisamente, una *propiedad de dominio* es una sentencia descriptiva sobre el mundo. Durante todo este trabajo, vamos a llamar *modelo ambiente*, al conjunto de propiedades del dominio de un problema particular.

Por otro lado, un *supuesto de ambiente* es una sentencia que podría no suceder y debe ser satisfecha por el ambiente. Un requisito de software, o *requisito* de forma abreviada, es una sentencia prescriptiva que la máquina deberá satisfacer independientemente de cómo se comporta el problema detallado en el mundo y deben ser elaboradas en términos de fenómenos compartidos entre el mundo y la máquina.

Para finalizar y siguiendo lo publicado en [VL01, VLL00] podremos determinar a una acción como supervisada/controlable si dicha acción es supervisada/controlable por la máquina. En este trabajo, llamaremos a las acciones supervisadas como acciones no controlables, ya que están controladas por el ambiente.

2.2. Sistema de Transición Etiquetado (Labelled Transition System)

En esta sección vamos a dar una notación para los Sistemas de Transiciones Etiquetados o Labelled Transition System (LTS), la cual usaremos durante este trabajo. Dichos sistemas, son muy usados actualmente para modelar y analizar comportamiento en sistemas concurrentes y distribuidos. Un LTS es un sistema de transiciones de estados donde cada uno de ellos está etiquetado con una acción. El conjunto de todas las acciones que posee un LTS es llamado alfabeto.

Definición 2.1. *(Sistema de Transición Etiquetado)[Kel76] Sea States un conjunto universal de estados, Act un conjunto universal de etiquetas. Un Sistema de Transición Etiquetado (LTS) es una tupla* $E = (S_E, A_E, \Delta_E, s_{E_0})$*, donde* $S_E \subseteq States$ *es un conjunto finito de estados,* $A_E \subseteq Act$ *es un alfabeto finito,* $\Delta_E \subseteq (S_E \times A_E \times S_E)$ *es una relación, y* $s_{E_0} \in S_E$ *es el estado inicial.*

Si $(s, \ell, s') \in \Delta_E$ diremos que ℓ está activo desde s en E. Diremos también que un LTS E es *determinístico* si $\forall_{(s,\ell,s'),(s,\ell,s'')\in\Delta_E}$ implica $s' = s''$. Para un estado s definiremos $\Delta_E(s) = \{\ell \mid (s, \ell, s') \in \Delta_E\}$.

Definición 2.2. *(Composición en Paralelo) Sean* $M = (S_M, A_M, \Delta_M, s_{M_0})$ *y* $E = (S_E, A_E, \Delta_E, s_{E_0})$ *LTSs. Una Composición en Paralelo(||) es un operador simétrico tal que* $E||M$ *es*

el LTS definido de la siguiente manera $E||M = (S_E \times S_M, A_E \cup A_M, \Delta, (s_{E_0}, s_{M_0}))$*, donde* Δ *es la relación más pequeña que satisface las siguientes reglas, donde* $\ell \in A_E \cup A_M$*:*

$$\frac{(s,\ell,s') \in \Delta_E}{((s,t),\ell,(s',t)) \in \Delta} \ \ell \in A_E \backslash A_M \qquad \frac{(t,\ell,t') \in \Delta_M}{((s,t),\ell,(s,t')) \in \Delta} \ \ell \in A_M \backslash A_E$$

$$\frac{(s,\ell,s') \in \Delta_E,\ (t,\ell,t') \in \Delta_M}{((s,t),\ell,(s',t')) \in \Delta} \ \ell \in A_E \cap A_M$$

Definición 2.3. *(LTS Legal) Dado* $E = (S_E, A_E, \Delta_E, s_{E_0}), M = (S_M, A_M, \Delta_M, s_{M_0})$ *LTSs, y* $A_{E_u} \in A_E$*. Decimos que* M *es un LTS Legal para* E *con respecto a* A_{E_u} *si para todos* $(s_E, s_M) \in E||M$ *sucede lo siguiente:* $\Delta_{E||M}((s_E, s_M)) \cap A_{E_u} = \Delta_E(S_E) \cap A_{E_u}$*.*

Intuitivamente, un LTS M es un LTS *Legal* para el LTS E con respecto a A_{E_u}, si para todos los estados en la composición $(s_E, s_M) \in S_{S||M}$ se cumple que, una acción $\ell \in A_{E_u}$ es deshabilitada en (s_E, s_M) si y sólo si también esta deshabilitada en $s_E \in E$. En otras palabras, M no restringe a E con respecto a A_{E_u}.

Definición 2.4. *(Traza) Sea un LTS* $E = (S, A, \Delta, s_0)$*. Una secuencia* $\pi = \ell_0, \ell_1, ...$ *es una traza en* E *si existe una secuencia* $s_0, \ell_0, s_1, \ell_1, ...$ *donde para todo* i *tenemos* $(s_i, \ell_i, s_{i+1}) \in \Delta$*.*

Definición 2.5. *(Estados Alcanzables) Sea un LTS* $E = (S_E, A_E, \Delta_E, s_0)$*. Un estado* $s \in S_E$ *es alcanzable (desde el estado inicial) en* E *si existe una secuencia* $s_0, \ell_0, s_1, \ell_1, ...$ *donde para cada* i *tenemos* $(s_i, \ell_i, s_{i+1}) \in \Delta$ *y* $s = s_{i+1}$*. Nos referimos al conjunto de todos los estados alcanzables en* E *como* $Reach(E)$*.*

En el transcurso de esta tesis, vamos a estudiar sólo aquellos LTSs E donde todos sus estados $s \in S_E$ son alcanzables.

2.3. Lógica Lineal Temporal de Fluents (Fluent Linear Temporal Logic)

La Lógica Lineal Temporal (LTL) está siendo ampliamente usada en la ingeniería de los requerimientos [KPR04, GM03, VLL00, LvL02]. La motivación para escoger a las LTL de fluents es que éstas proveen un framework uniforme para especificar propiedades basados en estados sobre modelos basados en eventos [GM03]. Fluent Linear Temporal Logic (FLTL) [GM03] es una lógica de tiempo lineal, temporal, para razonar acerca de fluents. Un *fluent* Fl es definido por un par de conjuntos y un valor booleano: $Fl = \langle I_{Fl}, T_{Fl}, Init_{Fl} \rangle$, donde $I_{Fl} \subseteq Act$ es el conjunto de acciones iniciadoras, $T_{Fl} \subseteq Act$ es el conjunto de acciones finales y $I_{Fl} \cap T_{Fl} = \emptyset$. Un fluent puede ser inicializado con valor *true* o *false* indicado por $Init_{Fl}$. Toda acción $\ell \in Act$ induce un fluent, que notaremos $\ell = \langle \ell, Act \backslash \{\ell\}, false \rangle$. Por último, el alfabeto de un fluent es el que se obtiene mediante la unión del conjunto de acciones iniciadoras y el conjunto de acciones finales

Sea $\mathcal{F}$ el conjunto de todos los posibles fluents sobre Act. Una fórmula FLTL se define inductivamente utilizando los conectores booleanos estándar y operadores temporales como el **X** (próximo), **U** (antes fuerte) de la siguiente manera:

$$\varphi ::= Fl \mid \neg\varphi \mid \varphi \ \vee \ \psi \mid \mathbf{X}\varphi \mid \varphi \ \mathbf{U} \ \psi \tag{2.1}$$

donde $Fl \in \mathcal{F}$. Para comodidad sintáctica, vamos a introducir las operaciones de $\wedge$, $\Diamond$ (finalmente) y $\Box$ (siempre). Sea Π el conjunto de trazas infinitas sobre Act, diremos que la traza $\pi = \ell_0, \ell_1, ...$ satisface un fluent Fl en la posición i, notado $\pi, i \vDash Fl$, si y sólo si una de las siguientes condiciones es válida:

- $Init_{Fl} \wedge (\forall j \in \mathbb{N} : 0 \leq j \leq i \rightarrow \ell_j \notin T_{Fl})$
- $\exists j \in \mathbb{N} : (j \leq i \wedge \ell_j \in I_{Fl}) \wedge (\forall k \in \mathbb{N} : j < k \leq i \rightarrow \ell_k \notin T_{Fl})$

Dada una traza infinita π, la fórmula que satisface φ en la posición i, denotada como $\pi, i \vDash \varphi$, es definida a continuación como se muestra en la semántica para el operador de satisfacción:

$$\begin{array}{lcl} \pi, i \vDash \neg\varphi & \triangleq & \neg(\pi, i \vDash \varphi) \\ \pi, i \vDash \varphi \vee \psi & \triangleq & (\pi, i \vDash \varphi) \vee (\pi, i \vDash \psi) \\ \pi, i \vDash \mathbf{X}\varphi & \triangleq & \pi, 1 \vDash \varphi \\ \pi, i \vDash \varphi \mathbf{U} \psi & \triangleq & \exists j \geq i : \pi, j \vDash \psi \wedge \forall i \leq k < j : \pi, k \vDash \varphi \end{array}$$

Diremos que φ se cumple en π, denotado como $\pi \vDash \varphi$, si $\pi, 0 \vDash \varphi$. Una fórmula $\varphi \in$ FLTL es cierta si un LTS E (denotado como $E \vDash \varphi$) si ésta es cierta en toda traza infinita producida por E.

2.4. Problemas de síntesis de controladores

Los problemas de síntesis de controladores son aquellos que producen una máquina que restringe las ocurrencias de los eventos controlables, basada en las observaciones de los eventos no controlables que han ocurrido. Dicha máquina, al ser desplegada con un ambiente adecuado, logra satisfacer el conjunto de objetivos del sistema. Cabe destacar, que estos objetivos se cumplirán si se satisfacen las suposiciones que se hacen sobre el ambiente. Resumiendo, tendremos una especificación del ambiente, suposiciones, objetivos, y un conjunto de acciones controlables. Resolver el *problema de síntesis de control* es hallar una máquina, que al trabajar concurrentemente con el ambiente, satisface las suposiciones del dominio, y satisface el conjunto de objetivos del sistema.

Hecha esta introducción definiremos el problema de síntesis de control para modelos basados en eventos de la siguiente manera. Dada un LTS que detalla el comportamiento del ambiente, un conjunto de eventos controlables, un conjunto de formulas FLTL que describen los objetivos del sistema, el problema de control LTS consiste en encontrar un LTS que restringe solamente la ocurrencia de acciones controlables y garantiza que la composición paralela del ambiente con el LTS recién descripto estará libre de deadlocks [1] y que, si las presunciones del ambiente valen, satisfará también los objetivos del sistema.

Definición 2.6. *(Control LTS) Dada una especificación de un entorno en forma de un LTS E, un conjunto de acciones controlables $A_c \in Act$ y una fórmula LTL φ, la solución al problema de control LTS $\mathcal{E} =< E, A_c, \varphi >$ consiste en encontrar un LTS M de forma que M es legal a E sobre el conjunto de acciones no controlables $A_U = \overline{A_c}$, $E||M$ se encuentra libre de deadlocks, y para cada traza π en $E||M$ cumple que $\pi \vDash \varphi$.*

[1] Un LTS no tiene deadlocks si todos sus estados tienen transiciones salientes.

Ahora pasaremos a definir un subconjunto de problemas de control LTS que está determinado por aquellos problemas de control que son computables en tiempo polinómico. Identificaremos estos problemas como problemas de control LTS SGR(1) (Safe Generalized Reactivity(1)). Estos se construyen a partir de GR(1) y problemas de seguridad pero en modelos basados en eventos. Dichos problemas, constan de un modelo del ambiente E que será un LTS determinístico para asegurar que el controlador tenga una visión completa de los estados del ambiente.

Definición 2.7. *(Control LTS SGR(1)) un problema de control LTS* $\mathcal{E} =< E, A_C, \varphi >$ *es SGR(1) si* E *es determinístico, y* $\varphi = \Box\, \rho \wedge (\bigwedge_{i=1}^{n} \Box \Diamond\, \phi_i \implies \bigwedge_{j=1}^{m} \Box \Diamond\, \gamma_j)$*, y* ϕ_i, ρ *y* γ_j *son combinación booleana de fluents.*

2.5. Juegos de dos jugadores

Llamaremos juegos de dos jugadores a aquellos que consisten en dos jugadores, jugador 1 y jugador 2, donde el objetivo del jugador 1 es satisfacer una especificación independientemente de las acciones que el jugador 2 ejecute. Intuitivamente, el jugador 1 puede deshabilitar las acciones que él controla aunque no podrá deshabilitarlas todas ya que esto transformaría dicho estado a un estado de deadlock.

Durante el transcurso de esta tesis llevaremos los juegos de dos jugadores al marco de síntesis de controladores, donde el jugador 1 (el controlador) elige, del conjunto de acciones controlables, cual habilitar y el jugador 2 (el ambiente) elige qué acciones tomar libremente. Formalmente podemos definir un juego de dos jugadores de la siguiente manera:

Definición 2.8. *(Juego de dos jugadores) Un juego de dos jugadores es* $G = (S_g, \Gamma^-, \Gamma^+, s_{g_0}, \varphi)$*, donde* S *es un conjunto finito de estados,* $\Gamma^-, \Gamma^+ \subseteq S \times S$ *son conjuntos de transiciones no controlables y controlables respectivamente,* $s_{g_0} \in S$ *es el estado inicial, y* $\varphi \subseteq S^\omega$ *es la condición de ganada. Definimos* $\Gamma^-(s) = \{s' \mid (s, s') \in \Gamma^-\}$ *y análogamente para* Γ^+*. Un estado* s *es no controlable si* $\Gamma^-(s) \neq \emptyset$ *y controlable en el resto de los casos. Una jugada en* G *es una secuencia* $p = s_{g_0}, s_{g_1}, \ldots$ *Una jugada* p *terminada en* s_{g_n} *es extendida por el controlador eligiendo un subconjunto* $\gamma \subseteq \Gamma^+(s_{g_n})$*. Luego el ambiente elige un estado* $s_{g_{n+1}} \in \gamma \cup \Gamma^-(s_{g_n})$ *y agrega* $s_{g_{n+1}}$ *a* p*.*

Un detalle importante es que si para un estado controlable γ el conjunto de opciones del controlador es vacía, esto puede llevar a un deadlock. Esto será considerado como prohibido más adelante, ya que el controlador definirá este estado como un estado perdedor. Para un estado no controlable el controlador puede decidir deshabilitar todas las acciones controlables. Las elecciones del controlador son formalizadas como estrategias y estas reglas son las que el controlador aplicará. Por lo general, las estrategias son elegidas dependiendo de la historia. Esto puede verse en la estrategia utilizando un valor de memoria Ω y actualizando este valor de acuerdo a la evolución del juego.

Es importante destacar que los juegos con memoria son diferentes a los definidos en [PPS06b]. Piterman et al. definen un juego en el cual el ambiente elige su movimiento y recién luego de éste, el controlador podrá elegir cuál será el siguiente paso.

Definición 2.9. *(Estrategia con memoria) Una estrategia con memoria* Ω *para el controlador es un par de funciones* (σ, u)*, donde* Ω *es una memoria que tiene designado como valor inicial* $\omega_0, \sigma : \Omega \times S \to 2^S$ *tal que* $\sigma(\omega, s) \subseteq \Gamma^+(s)$ *y* $u : \Omega \times S \to \Omega$*.*

Intuitivamente, σ le informa al controlador cuáles estados debe habilitar como posibles sucesores y u define cómo actualizar la memoria en cada paso. Si Ω es finita, diremos que la estrategia usa memoria finita.

Definición 2.10. *(Consistencia y estrategia ganadora) una jugada finita o infinita* $p = s_0, s_1, ...$ *es consistente con* (ω, u) *si para cada* n *tenemos que* $s_{n+1} \in \sigma(\omega_n, s_n)$ *donde* $\omega_{i+1} = u(\omega_i, s_{i+1})$ *para toda* $i \geq 0$. *Una estrategia* (σ, u) *para el controlador desde el estado* s *es ganadora si cada jugada maximal empezando de* s *y consistente con* (σ, u) *es infinita y en* φ. *Diremos que el controlador gana el juego* G *si tiene una estrategia ganadora desde el estado inicial.*

Diremos que verificar si un controlador gana un juego G es resolver el juego G. Una vez definido un juego de dos jugadores, pasaremos a traducir un problema de síntesis de controladores a este tipo de juegos. La transformación se basa en generar una estrategia ganadora para el controlador. Si dicha estrategia existe, diremos que el problema de control es realizable [MPS95, RW89]. Resultados estudiados anteriormente [PR89], demuestran que si un controlador gana el juego G y φ es ω-regular, el juego puede ganarse utilizando una estrategia con memoria finita.

2.6. Resolución de problema de control

En esta sección explicaremos cómo una solución para un problema de control SGR(1) puede ser obtenida por construcción utilizando técnicas existentes de síntesis de controladores (basados en estados), llamados GR(1). [PPS06b]

La construcción de la máquina para un problema de control LTS SGR(1) está dividido en dos pasos. Primero, se crea un juego GR(1) G en representación del ambiente E, las suposiciones A_s, los objetivos O y el conjunto de acciones controlables A_C. Como segundo paso, se elabora una solución (σ, u) al juego GR(1) para construir una máquina M (i.e un controlador LTS) para $\mathcal{E}$. Esta solución al problema de control LTS SGR(1) $\mathcal{E}$ existe, si y sólo si, existe una solución al juego GR(1) G. Luego, podremos afirmar que el controlador LTS M creado a partir de (σ, u) es una solución a $\mathcal{E}$.

2.6.1. Control LTS SGR(1) a juegos GR(1)

Convertiremos el problema de control LTS SGR(1) a un juego GR(1). Dado un problema de control LTS SGR(1) $\mathcal{E} =< E, A_c, \varphi >$ construimos un juego GR(1) $G = (S_g, \Gamma^-, \Gamma^+, s_{g_0}, \varphi_g)$ tal que cada estado en S_g representa un estado en E y una valuación de todos los fluents que aparecen en A_s y en G.

Más precisamente, y por la definición de control LTS SGR(1) (definición 2.7) tendremos que $\varphi = \bigwedge_{i=1}^{m} \Box \Diamond \phi_i \implies \bigwedge_{j=1}^{m} \Box \Diamond \gamma_j$, $E = (S_e, A, \Delta_e, s_{e_0})$, $A_s = \bigwedge_{i=1}^{n} \Box \Diamond \phi_i$, $I = \Box \rho$ y $G = \bigwedge_{j=1}^{m} \Box \Diamond \omega_j$. Sea $fl = \{\dot{1}, ..., \dot{k}\}$ un conjunto de fluents usados en A_s y en G donde $\dot{i} = \langle I_i, T_i, Init_i \rangle$. Construimos al juego $G = (S_g, \Gamma^-, \Gamma^+, s_{g_0}, \varphi_g)$ de la siguiente manera:

Construimos S_g a partir de E de tal forma, que los estados en S_g corresponden a un estado en E y los valores de verdad de los *fluents* en φ. Formalmente, tenemos que $S_g = S_e \times \prod_{i=1}^{k} \{true, false\}$. Consideramos un estado $s_g = (s_e, \alpha_1, ..., \alpha_k)$. Dado un fluent fl_i, diremos que s_g satisface fl_i si α_i es *true* y s_g no satisface fl_i si no.

Además, definiremos las relaciones Γ^- y Γ^+ aplicando las siguientes reglas. Sea $s_g = (s_e, \alpha_1, ..., \alpha_k)$. Si s_g no satisface ρ (es decir, s_g es no seguro) no agregaremos los sucesores

a s_g. Si s_g satisface ρ, por cada transición $(s_e, l, s'_e) \in \Delta_e$ agregaremos $(s_g, (s'_e, \alpha'_1, ..., \alpha'_k))$ en Γ^β, donde β y α'_i cumplen las siguiente condiciones:

β	α'_i
es +: si $l \in A_C$,	es α_i: si $l \notin I_{fl_i} \cup T_{fl_i}$,
es -: si $l \notin A_C$.	es $true$: si $l \in I_{fl_i}$ o
	es $false$: si $l \in T_{fl_i}$.

El estado inicial s_{g_0} es $(s_{e_0}, initiall_1, ..., initiall_k)$.

Por último, construiremos la *condición de ganada* φ_g, definida como un conjunto infinito de trazas, para A_S y G de la siguiente manera: abusando de la notación denotaremos ϕ_i al conjunto de estados s_g tales que s_g satisface las asunciones ϕ_i y a γ_i al conjunto de secuencias que satisfacen $gr((\phi_1, .., \phi_n), (\gamma_1, ..., \gamma_m))$. De esta forma, obtendremos que $G = (S_g, \Gamma^-, \Gamma^+, s_{g_0}, \varphi_g)$ es un juego GR(1).

Cabe destacar que las propiedades de seguridad (*safety*) que son parte de la especificación no están contempladas en la *condición de ganada* φ_g del juego GR(1), pero se traducen a un problema de *deadlock avoidance* a la hora de construir Γ^- y Γ^+. De esta manera, la *condición de ganada* es $\square\ \rho \wedge (\bigwedge_{i=1}^{n} \square\Diamond\phi_i \Rightarrow \bigwedge_{j=1}^{m} \square\Diamond\omega_j)$.

2.6.2. Traduciendo la estrategia a un Controlador LTS

Ahora pasaremos a explicar como conseguir un controlador LTS a partir de una estrategia ganadora para el juego en GR(1). Intuitivamente, la transformación es de la siguiente manera: dado un problema de control LTS SGR(1) $\mathcal{E} = \langle E, H, A_C \rangle$, el juego $G = (S_g, \Gamma^-, \Gamma^+, s_{g_0}, \varphi_g)$ obtenido a partir de $\mathcal{E}$ y de la estrategia ganadora para G, construimos $M = (S_M, A, \Delta_M, s_{M_0})$ una solución para $\mathcal{E}$ traduciendo a estados de S_M un estado de S_g y un estado de la memoria dada por la estrategia ganadora.

Más formalmente, sea $E = (S_e, A, \Delta_e, s_{e_0})$, $fl = \{fl_1, ..., fl_k\}$ el conjunto de fluents que aparecen en φ, $G = (S_g, \Gamma^-, \Gamma^+, s_{g_0}, \varphi_g)$ el juego GR(1) construido a partir de E como explicamos anteriormente, y sea $\sigma : \Omega \times S_g \to 2^{S_g}$ y $u : \Omega \times S_g \to \Omega$ la estrategia ganadora para G. Construiremos la máquina $M = (S_M, A, \Delta_M, s_{M_0})$ de la siguiente manera.

Para construir $S_M \subseteq \Omega \times S_g$, consideremos dos estados $s_g = (s_e, \alpha_1, ..., \alpha_k)$ y $s'_g = (s'_e, \alpha'_1, ..., \alpha'_k)$. Decimos que esa acción ℓ es *posible* desde s_g hacia s'_g si:

1. $(s_g, s'_g) \in \Gamma^- \cup \Gamma^+$,
2. existe una acción ℓ tal que $(s_e, \ell, s'_e) \in \Delta_e$ y
3. para cada *fluent* fl_i vale alguna de las siguiente condiciones:
   - $\ell \notin I_{fl_i} \cup T_{fl_i}$ y $\alpha'_i = \alpha_i$,
   - $\ell \in I_{fl_i}$ y $\alpha'_i = true$, o
   - $\ell \in T_{fl_i}$ u $\alpha'_i = false$.

Para construir $\Delta_M \subset S_M \times A \times S_M$, consideremos la transición $(s_g, s'_g) \in \Gamma^-$. Por definición de Γ^- existe una acción $\ell \notin A_C$ tal que ℓ es posible desde s_g hacia s'_g. Si $s'_g \in \sigma(\omega, s_g)$ entonces para cada acción ℓ tal que ℓ es posible desde s_g hacia s'_g agregamos $((\omega, s_g), \ell, (u(\omega, s_g), s'_g))$ hacia Δ_M. De forma similar, consideramos una transición $(s_g, s'_g) \in \Gamma^+$. Por definición de Γ^+ existe una acción $\ell \in A_C$ tal que ℓ es posible desde s_g

hacia s'_g. Si $s'_g \in \sigma(\omega, s_g)$ entonces para cada acción ℓ tal que ℓ es posible desde s_g hacia s'_g agregamos $((\omega, s_g), \ell, (u(\omega, s_g), s'_g))$ hacia Δ_M.

El estado inicial de M está definido como $s_{M_0} = (\omega_0, s_{g_0})$ donde ω_0 es el valor inicial de la memoria Ω. De esta forma completamos la definición de M.

2.6.3. Algoritmo Secuencial SGR(1)

En esta sección, presentaremos el algoritmo implementado en la herramienta MTSA [DFCU08a] el cual está basado en las ideas de Juvekar y Piterman [JP06a].

Este algoritmo realiza una búsqueda de ciclos de estados que satisfacen todas las suposiciones pero no todos los objetivos restringiendo acciones controlables. De haber ciclos como estos podrían permitir trazas en las que el controlador pierde el juego GR(1). Para lograr evitar estos ciclos, el algoritmo busca para cada estado, una estrategia que garantice la satisfacción de todos los objetivos. Para esto, se configura un orden en el cual satisfacer los objetivos. El algoritmo, mediante la técnica de punto fijo computa la mejor forma en que cada estado puede satisfacer el siguiente objetivo. A su vez, mide la "calidad" de cada uno de los diferentes sucesores para satisfacer un objetivo mediante un sistema de rankings [Jur00]. El ranking de un sucesor particular mide la distancia (cantidad de transiciones utilizadas) al siguiente objetivo en términos de cantidad de veces que las suposiciones son satisfechas antes de alcanzar el objetivo. Si este número tiende a infinito, deduciremos que desde el estado actual existe una traza infinita en la cual las suposiciones del ambiente valen infinitamente, pero los objetivos no se satisfacen. Es así como el algoritmo reconoce estados que deben ser evitados para la construcción de la estrategia para el controlador.

Definición 2.11. *(Función de Ranking) Sea* $G = (S_g, \Gamma^-, \Gamma^+, s_{g_0}, \varphi)$ *donde* $\varphi = gr((\phi_1, ..., \phi_n), (\gamma_1, ..., \gamma_m))$. *Una función de ranking para un objetivo* γ_j *es una función* $R_j : S_g \to (\mathbb{N} \times \{1, ..., n\} \cup \{\infty\})$. *Intuitivamente,* $R_j(s_g) = (k, \ell)$ *significa que para alcanzar desde* s_g *a un estado en el cual* γ_j *vale, todos los caminos satisfacen la suposición* ϕ_ℓ *a lo sumo* k *veces.* $R(s) = \infty$ *significa que* s *es un estado perdedor, es decir, desde* s *no hay una estrategia para el controlador que pueda evitar una traza en la cual se satisface infinitamente las suposiciones, pero no satisface infinitamente a todos los objetivos.*

Los estados que pertenecen a objetivos de *safety* serán *dead end* de nuestro juego.

El algoritmo 1 computa un ranking estable en cada estado $s_g \in T$ si s_g es ganador para el controlador (es decir, $R_1(t) < \infty$). Conceptualmente, podemos separar el algoritmo en dos grandes instancias, inicialización y estabilización. El valor inicial del ranking para cada estado en el juego, junto a la cola de estados *pending* para ser procesados, se crean en la etapa de inicialización. Agregaremos un estado a *pending* si no satisface ningún objetivo y satisface las suposiciones. Todos los estados en cada función de ranking son inicializados con el valor $(0, 1)$. Este valor indica el menor ranking posible. Los estados que cumplen que $\Gamma^- \cup \Gamma^+ = \emptyset$ serán inicializados con el valor ∞. De esta manera, los estados cuyos rankings son ∞ son aquellos donde no se satisface ρ o son estados de *deadlock* en E.

La sección de estabilización es una iteración de punto fijo sobre la cola *pending* hasta que se vacía. La función `is_stable(state,g)` devuelve true si la g-esima función de ranking es estable para `state` (ver definición 2.12).

La función `unstable_pred(state,g)` devuelve un conjunto de pares de predecesores de `state` y un ranking g para el cual el ranking es inestable.

Decimos que un estado s_g es estable en R_j si todas las siguientes condiciones se cumplen:

Algorithm 1 para resolver juegos SGR(1)

```
1:  procedure SOLVEGAME(GAME=(STATES,TRANSITIONS),SAFE,GUARANTEES,ASSUMPTIONS)
2:      // Inicialización:
3:      for state : states do
4:          for g : guarantees do
5:              rank_g(state)← (0,1)
6:      for state : states do
7:          if state.isDeadEnd() then
8:              rank_g(state)← ∞
9:      // Encolar pendientes
10:     for state : states do
11:         if ∃ g : guarantees / state ∉ g ∧ state ∈ assumptions_1 then
12:             // create a new pair and push it
13:             pending.push(pair(state,g))
14:         if Γ⁻(state) = ∅ ∧ Γ⁺(state) = ∅ then
15:             for g : guarantees do
16:                 rank_g(state) ← ∞
17:                 pending.push(unstable_pred(state,g))
18:     // Estabilización:
19:     while !pending.empty() do
20:         (state,g) ← pending.pop()
21:         if rank_g(state) = ∞ then
22:             continue
23:         if isStable(rank_g(state)) then
24:             continue
25:         rank_g(state) ← increment(best(state,g),state,g)
26:         pending.push(unstable_pred(state,g))
```

Definición 2.12. is_stable(state,g),

- Si $s_g \in \gamma_j$ y $sr(s_g, j \oplus 1) = \infty$ entonces $R_j(s_g) = \infty$.
- Si $s_g \in \gamma_j$ y $sr(s_g, j \oplus 1) \neq \infty$ entonces $R_j(s_g) = (0, 1)$.
- Si $s_g \notin \gamma_j$, $R_j(s_g) = (k, l)$ y $s_g \in \phi_l$, entonces $R_j(s_g) \geq sr(s_g, j)$.
- Si $s_g \notin \gamma_j$, $R_j(s_g) = (k, l)$ y $s_g \notin \phi_l$, entonces $R_j(s_g) \geq sr(s_g, j)$.

La función `best(state,g)` devuelve el mejor ranking basado en sus sucesores (ver definición 2.2). Para eso utiliza la siguiente función $sr : S_g \to (\mathbb{N} \times \{1, ..., n\} \cup \{\infty\})$. Esta función también codifica el hecho que los estados de *deadlock* tienen ranking ∞. Además, notemos que usa un orden lexicográfico para los objetivos. Dado un estado s_g y un objetivo γ_j, $sr(s_g, j)$ está definida de la siguiente manera:

$$best(s_g, j) = \begin{cases} \infty & \text{si } \Gamma^+(s_g) \cup \Gamma^-(s_g) = \emptyset \\ min_{s'(g) \in \Gamma^+(s_g)} R_{j \oplus 1}(s'_g) & \text{si } s_g \text{ es controlable y } s_g \in \gamma_j \\ min_{s'(g) \notin \Gamma^+(s_g)} R_j(s'_g) & \text{si } s_g \text{ es controlable y } s_g \notin \gamma_j \\ max_{s'(g) \in \Gamma^-(s_g)} R_{j \oplus 1}(s'_g) & \text{si } s_g \text{ no es controlable y } s_g \in \gamma_j \\ max_{s'(g) \in \Gamma^-(s_g)} R_j(s'_g) & \text{si } s_g \text{ no es controlable y } s_g \notin \gamma_j \end{cases} \quad (2.2)$$

Por último, `increment((`k`,`ℓ`),state,g)` devuelve $(0, 1)$ si `state` está en γ_g, devuelve (k, ℓ) si `state` no está en $assumption_\ell$, y devuelve el mínimo valor mayor que (k, ℓ) en el resto de los casos. Notemos que `increment(`∞`,state,g)` es ∞, y si $n = max_\ell(|\phi_\ell - (\gamma_g)|)$ y `state` está en $\phi_m - \gamma_g$ entonces `increment((n,m),state,g)` es ∞. Este algoritmo calcula el mínimo ranking estable. Basados en ideas del mundo de autómatas de Büchi [EWS05, JP06b], este algoritmo puede ser implementado en $O(m \cdot n \cdot |S|^2)$.

2.7. Procesos de estados Finitos (Finite State Process)

A esta altura, ya hemos definido las LTSs definiendo sus componentes, como lo son, sus estados, sus acciones, sus transiciones y su estado inicial. Esta representación es adecuada para LTSs con pocos estados, pero se vuelve muy poco práctica a la hora de trabajar con LTSs de gran tamaño. Por esta razón, usamos una simple notación de álgebra de procesos llamada procesos de estados finitos (FSP: Finite State Process) para especificar LTSs. [MKG97, MK99]

FSP es un lenguaje de especificación de semántica bien definida en términos de LTSs que provee describirlos de manera concisa. Cada expresión FSP E puede ser relacionada a un LTS finito. Notaremos $lts(E)$ al LTS que corresponde a dicho FSP. A continuación discutiremos detalladamente la sintaxis del lenguaje FSP.

A modo de ejemplo, en la Figura 2.2, mostramos un código FSP que representa el funcionamiento de una planta nuclear.

En FSP, los nombres de los procesos empiezan con letras mayúsculas y las acciones con minúsculas. El código de la planta nuclear consta de dos procesos FSP, el primero, llamado `MAINTENANCE` modela el proceso de enviar un mensaje para que se realice el mantenimiento de la bomba refrigeradora y recibe la respuesta de dicho mensaje. Estas acciones se representan con las acciones `request` y `ok` respectivamente. Por otro lado,

```
MAINTENANCE = (request->ok->MAINTENANCE).

COOLER = STARTED,
STARTED = (stopPump->STOPPED | procedure->STARTED |
                  ok->STARTED),
STOPPED = (startPump->STARTED | procedure->STOPPED |
                  ok->STOPPED).

||COOLING_TOWER = (MAINTENANCE||COOLER).
```

Fig. 2.2: Ejemplo FSP

tenemos el proceso `COOLER` que posee como procesos auxiliares a los subprocesos `STARTED` y `STOPPED` que son locales al proceso FSP en donde están definidas. `COOLER` está definida para que inicialmente se comporte como `STARTED` puesto que queremos modelar que la bomba en estado inicial está prendida. Luego, podemos ejecutar diferentes acciones, `stopPump`, `procedure` y `ok`. `STARTED` está definido usando el operador de acción `->` y recursión. Por ejemplo, dicho proceso está definido para empezar ejecutando, o bien `procedure` o bien `ok`, acciones que nos llevan a seguir ejecutando como el proceso `STARTED`, o `stopPump` que nos llevará a ejecutar el proceso `STOPPED`.

A su vez, los FSP soportan distintos operadores de composición como la composición en paralelo. Dicha operación, denotada como $||$, está definida para preservar la semántica de la composición en paralelo de los LTS definidos en la definición 2.2. Por lo tanto, dados dos procesos FSP `P` y `Q`, tenemos: `lts(P||Q) = lts(P)||lts(Q)`.

Los procesos FSP que están definidos mediante una composición de dos procesos no auxiliares, son llamados procesos compuestos y sus nombres poseen el prefijo $||$. En nuestro ejemplo, la composición en paralelo entre los procesos FSP `MAINTENANCE` y `COOLER` se escribe como `||COOLING_TOWER = (MAINTENANCE||COOLER).`

Además, FSP posee palabras reservadas que se colocan antes de la definición de un proceso que fuerzan a la herramienta MTSA a realizar una operación más compleja al proceso. Un caso de estos, es la palabra reservada `minimal`, la cual, hace que MTSA construya un LTS minimal que respeta la semántica equivalente, esto implica que el LTS resultante será bisimilar al original; o la palabra reservada `deterministic`, que construye un LTS minimal con respecto a las trazas, nos asegura que el LTS resultante simulará el original, pero no viceversaa.

FSP también permite definir propiedades FLTL. Un fluent que marca aquellos estados donde la bomba está apagada puede ser expresada en lenguaje FSP mediante el siguiente código: `fluent IsStopped = <stopPump,startPump> initially 0`. Como dijimos anteriormente, la bomba empieza encendida, por lo tanto IsStopped es inicialmente falso, pasa a ser verdadero cuando sucede la acción `stopPump` y falso nuevamente cuando la acción `startPump` sucede.

Por último, FSP nos otorga facilidad para especificar LTSs y fórmulas FLTL. Este lenguaje es el que utilizaremos en los siguiente capítulos para definir modelos que representan ambientes y objetivos.

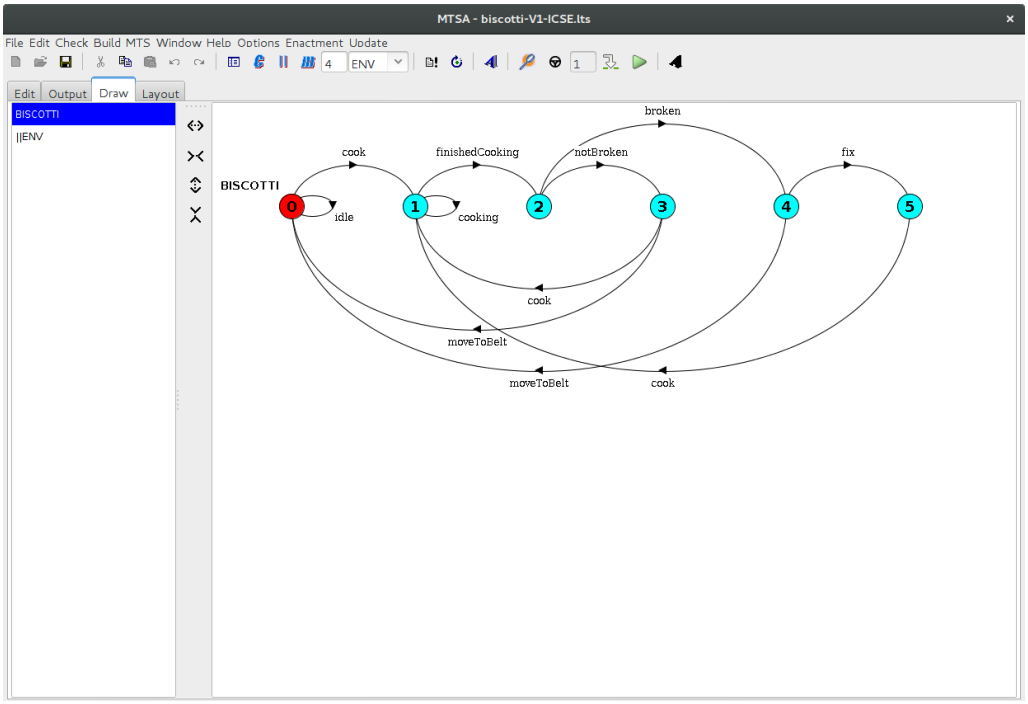

Fig. 2.3: Ventana de MTSA con LTS

2.8. Modelado del Ambiente para controladores

En la herramienta MTSA los ambientes de los modelos se describen utilizando una extensión del lenguaje FSP. La extensión a FSP provista por MTSA incluye los operadores tradicionales para describir el comportamiento de los modelos. Como por ejemplo prefijo ($->$), elección (|), composición secuencial (;), y composición paralela (||). Estas extensiones permiten modelar modelos de comportamiento parcial. Para más detalle ver [DFCU08b].

MTSA integra la funcionalidad para construir, analizar y elaborar LTS. Provee una interfaz gráfica para facilitar estas tareas. Se muestra en la figura 2.3 una imagen de la herramienta en su versión actual. En esa imagen se puede ver el LTS del ejemplo llamado Biscotti, que cocina elementos de cerámica.

Se puede ver que en el entorno de cocción las transiciones permiten: cocinar, terminar de cocinar, chequear si la pieza esta sana o rota y por último, arreglarlo o moverlo a la cinta de producción.

El objetivo esperado es que la cerámica salga sin roturas de la cocción y que se cocine dos veces. Luego, necesitamos que si se cumple la condición de seguridad la cerámica sea colocada en la cinta y se finalice el proceso.

2.9. Modelado de los Objetivos (o misión) de los controladores

Para explicar cómo modelar los objetivos damos a continuación un ejemplo de cocción de cerámica.

```
//Goals
controllerSpec G1 = {
    safety = {SUCCESSFULLY_COOKED_TWICE}
    failure = {Failures}
    assumption = {FinishCooking}
    liveness = {SuccessfullyCookedTwice}
    controllable = {Controllable}
}
```

Notemos que en la especificación mostrada se utilizan ciertas palabras clave. La lista de opciones es la siguiente:

assumption Asunciones expresadas como aserciones

safety Condiciones de seguridad

failure Fallas

liveness Garantías expresadas como propiedades o *fluents*

controllable Conjunto de transiciones controlables

Además la herramienta brinda soporte para verificar la compatibilidad de las asunciones utilizadas. Una vez que se genera el controlador se pueden usar otras herramientas de verificación que validan que no haya *deadlock* y las que condiciones de `safety` se cumplan.

Las condiciones de `liveness` y de `asumption` se escriben usando aserciones FLTL. Como queremos que nuestro controlador siempre continúe cocinando cerámicas, entonces en la asunción pedimos que siempre que se termina de cocinar, eventualmente se vuelve a cocinar.

En la condición de seguridad se pide que o bien cocinando y moviendo a la cinta si la pieza esta sana o bien, arreglando y volviendo a cocinar. Las condiciones de `safety` se pueden redactar usando un LTS o bien una propiedad LTL.

Además podemos codificar fallas, que será un conjunto de transiciones donde el sistema fallará. Esto no quiere decir que no exista un controlador resultante, sino que sigue intentando por otros caminos hasta que no falle. En nuestro ejemplo queremos que si se rompió se arregle, luego la falla es *roto*. Se puede leer más al respecto en [DIBPU11].

2.10. Síntesis de un controlador

Tomemos el ejemplo de la figura 2.4. Notar que el entorno tiene 4 estados: el estado inicial, dos estados intermedios y un estado que llamaremos estado trampa.

Las posibles transiciones están nombradas usando las letras “a, b, c, d, e, f”. Las transiciones controlables por el futuro controlador están marcadas con un signo +.

El objetivo del controlador esta descripto usando *fluents*. La notación se lee de la siguiente manera: donde dice $< e, All \backslash \{e\} >$, debe leerse coloquialmente como “se prende con e y se apaga con todos menos e”. Esto significa que, en este caso la garantía, será válida en todas las trazas donde eventualmente sólo se pase por una transición e. Por ejemplo: $b \rightarrow e \rightarrow e \rightarrow e...$

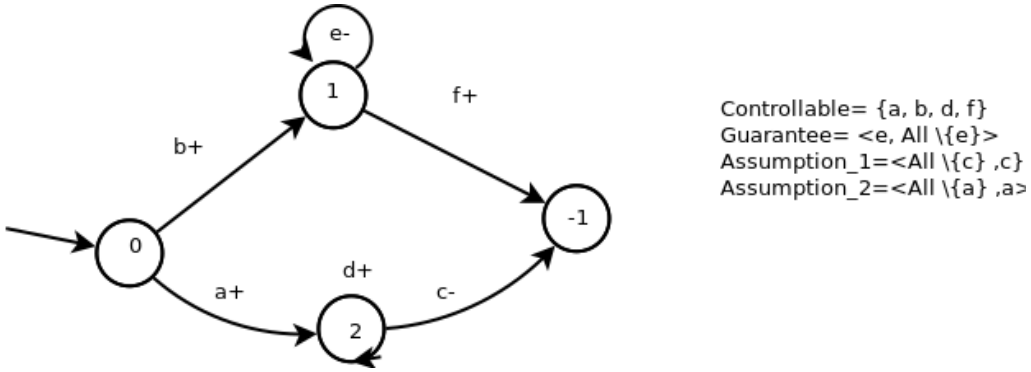

Fig. 2.4: Entorno y Objetivos

```
\\LTS
Cero = (b -> Uno | a -> Dos),
Uno = (e -> Uno | f -> Trampa),
Dos = (d -> Dos | c -> Trampa).

\\Entorno
set Control = {a,b,f,d}

fluent A = <a, Control\{a}>
fluent C = <c, Control\{c}>

assert Assume1 = !C
assert Assume2 = !A

fluent Guarantee = <e, Control\{e}>

\\Objetivos
controllerSpec Goal = {
    controllable ={Control}
    assumption = {Assume2, Assume1}
    liveness = {Guarantee}
}


controller ||C = Cero ~{Goal}.
```

Al componer el ambiente con los objetivos se obtiene el juego mostrado en la figura 2.5.

En el juego se puede observar que el único estado controlable (marcado con un círculo) es el inicial, todo el resto de los estados del juego son no controlables.

2.10.1. Seguimiento de estabilización

En esta sección haremos el seguimiento de la actualización de valores de ranking hasta su estabilización.

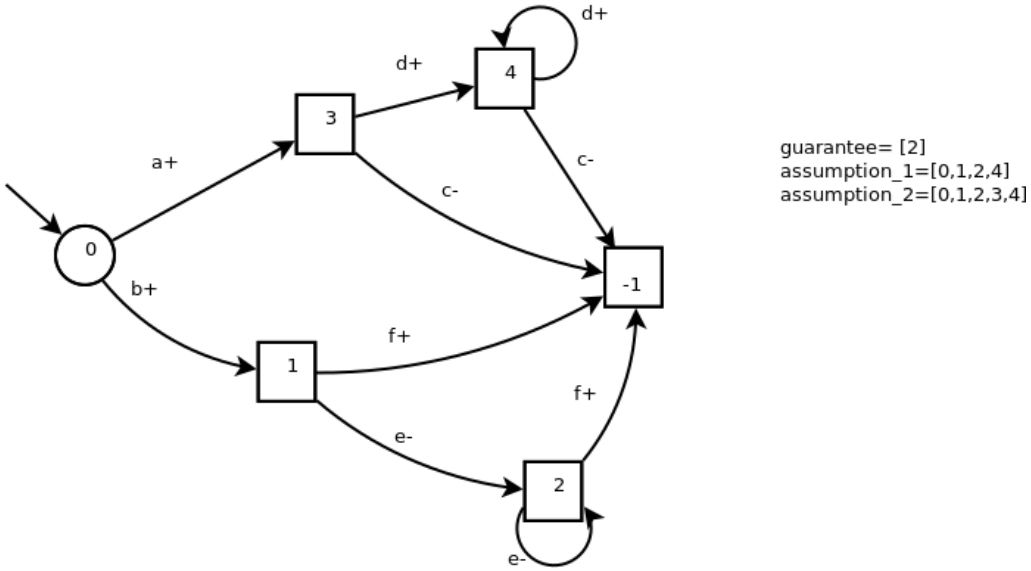

Fig. 2.5: Juego a sintetizar

En este ejemplo, al inicio los valores de ranking de los estados son (0,1) excepto el nodo trampa que inicia en ∞. Recordemos que en la definición 2.11 se muestra que (0,1) significa: desde el estado actual todos los caminos satisfacen la asunción 1 a lo sumo 0 veces. O lo que es sinónimo: nunca fue alcanzada la primera suposición.

La cola inicial tiene los elementos de predecesores (de transiciones no controlables) al estado final (trampa): $\{4, 3\}$; y los estados que pertenecen a la primera asunción y no pertenecen a las garantías: $\{0, 1, 4\}$. Luego la cola inicial esta compuesta por:$\{0, 1, 4, 3\}$.

Mientras haya elementos a procesar en la cola, se seguirá estabilizando los valores del ranking. La tabla 2.10.1 indica la actualización de los valores de ranking en cada paso. El paso inicial es el cero.

Al finalizar cada ciclo de actualización de un estado, se actualiza la cola de elementos pendientes. Se hace agregando los elementos predecesores al elemento actualizado. Sólo se agregan los elementos de las transiciones no controlables. Por ejemplo, en la tabla 2.10.1 en el paso tres (3) se ve que al actualizar el valor del estado cuatro (4), no se agrega ningún elemento nuevo a la cola de pendientes. Esto se debe a que las transiciones entrantes al estado cuatro (4), ambas son controlables (d+ : ver figura 2.5).

En el primer paso se consume el estado tres (3). Primero se evalúa si dicho estado alcanzó el valor infinito, como la respuesta es no y el estado aún no esta estable, se busca cuál es el mejor valor posible para dicho estado. Como el estado es no controlable, se busca el máximo valor de sus sucesores no controlables, el estado trampa: (5,1). Notar que como el estado tres no pertenece a la denominada primera asunción ($assumption_1$) no es necesario incrementar el valor obtenido. Luego, se actualiza el valor y se procede a agregar los predecesores a la cola de pendientes. Notar que la transición del estado cero al estado tres es controlable, luego no se agregará ningún elemento a la cola de pendientes.

En el segundo paso se consume el estado uno (1). Como el estado no alcanzó aún el valor infinito ni tampoco es estable, se busca el mejor posible valor para dicho estado. Siguiendo el mismo razonamiento del estado tres, el mejor posible valor pertenece al estado dos (2): (0,1). Como el estado uno (1) pertenece a la primera asunción debemos incrementar el

estado/paso	0	1(3)	2(1)	3(4)	4(0)
-1 (trampa)	(5,1)				
0	(0,1)				(1,1)
1	(0,1)		(0,2)		
2	(0,1)				
3	(0,1)	(5,1)			
4	(0,1)			(5,1)	

Tab. 2.1: Actualización por pasos del sistema de ranking

Paso	Poll	Cola
0		{3,1,4,0}
1	3	{1,4,0}
2	1	{4,0}
3	4	{0}
4	0	{}

Tab. 2.2: Actualización por pasos de la cola de pendientes

valor obtenido de su sucesor. Esto sirve para saber que estamos visitando un estado que pertenece a una asunción pero aún no hemos visitado ningún estado que pertenezca a la garantía que nuestro controlador debe asegurar. El valor actualizado queda en (0,2). En este caso tampoco se agrega ningún elemento a la cola de pendientes.

En el tercer paso se consume el estado cuatro (4). El valor de dicho estado en este momento es (0,1). Como aún el estado no esta estable respecto de sus sucesores se busca el mejor posible valor para actualizarlo. En este caso el sucesor no controlable tiene valor (5,1). El estado 4 pertenece a la primera asunción, pero ya alcanzó el máximo valor de caminos hasta la asunción 1. Luego, no hay más incremento posible. En cuanto a la cola de elementos pendientes, la transición entrante es controlable, luego no se agrega ningún elemento nuevo.

Por último en el paso 4 se consume el estado cero (0). El valor de dicho estado actualmente es (0,1). Aún no está estable y el estado cero es controlable, por lo que su mejor valor posible será el mínimo de sus sucesores: (0,2). Como dicho estado pertenece a la primera asunción notemos que el ranking (0,2) nos esta indicando que hemos visto la asunción 2 en el camino que nos lleva de vuelta al estado cero y estamos buscando ver infinitamente. Luego, como el estado cero pertenece a la asunción 1, debemos incrementar el valor para recordar que hemos visto todas las asunciones al menos una vez y volver a inicializar el l para volver a buscar todas las asunciones nuevamente. El ranking queda entonces en (1,1). El estado cero no tiene predecesores, luego la cola de pendientes queda vacía. Como no hay más elementos en la cola de pendientes el algoritmo termina.

3. ALGORITMO CONCURRENTE SGR(1)

En esta sección se plantean problemas vinculados con el uso de concurrencia en programación. Luego se muestra el pseudocódigo del algoritmo concurrente presentado para sintetizar problemas de control con objetivos de tipo *Generalized Reactivity (1)*. Además se muestra que dicho algoritmo es correcto y que esta correctitud no está ligada a detalles de configuración del juego o de la computadora.

3.1. Acceso concurrente a objetos compartidos

Para poder acceder a los objetos compartidos se pueden tomar distintas estrategias. En este capítulo se discuten las diferentes opciones de dichos accesos.

3.1.1. Repensar la Concurrencia

La ejecución clásica de un programa se entiende como: en una única unidad de procesamiento se ejecuta de forma secuencial una tarea tras otra. Al empezar a pensar en ejecución concurrente se piensa en separar el programa en subtareas y que cada una de ellas se ejecute en procesos separados, llamados *hilos*, que pueden o no ejecutar en la misma unidad de procesamiento. Los hilos (*threads* en inglés), pueden o no, compartir memoria o recursos.

Los modelos clásicos de programación concurrente asumen que en la ejecución del programa, las operaciones de cada hilo se van a ejecutar de a una a la vez y secuencialmente, y que el comportamiento del programa en general va a contemplar todos los posibles solapamientos de las operaciones.

Sin embargo, este modelo de concurrencia es insuficiente. Hoy en día las operaciones de hardware, a nivel procesador, por ejemplo: operaciones fuera de orden, y las optimizaciones de software, a nivel de compilación o de sistema operativo, por ejemplo: caches y buffers de almacenamiento, evidencian una deficiencia en la forma de programación concurrente de la actualidad.

Debido a la posibilidad de solapamiento de las operaciones en el mismo objeto y duplicación de instancias en distintos niveles de memoria (cache y memoria principal), debemos enfocar la programación concurrente para contemplar estas posibilidades. Además, no podemos asumir que dentro de un hilo de ejecución el procesador ejecutará dichas operaciones de forma secuencial. Luego, la definición clásica de concurrencia en un esquema de computadoras actuales (con múltiples núcleos), es insuficiente.

3.1.2. Sincrónico, concurrente y paralelo

La ejecución de múltiples tareas que interactúan entre sí durante un mismo lapso de tiempo se denomina ejecución concurrente. Las tareas pueden ser un conjunto de hilos de ejecución creados por el mismo programa y ejecutados en la misma unidad de proceso, en varios procesadores o en varias computadoras en red.

Al utilizar los mismos recursos se puede requerir una sincronización de acceso: un cierto orden de lectura o escritura sobre los objetos. En cualquier caso necesitamos que

la secuencia de interacciones y comunicaciones este coordinada para que los accesos a los recursos permitan un resultado correcto.

Debemos pensar los algoritmos, de forma general y no atada a un lenguaje de programación, por lo cual, hay que tener en cuenta diferentes aspectos de acceso a los colaboradores de los métodos. Podemos distinguir dentro de esos accesos los que son sólo de lectura y los que requieren escritura. Además tenemos objetos mutables e inmutables. Los últimos son los que tendremos que tener en cuenta ya que pueden derivar en errores.

En nuestro caso particular el algoritmo accede de forma concurrente a el sistema de valores de *ranking* y a la cola de elementos pendientes para analizar.

3.1.3. Administración de acceso a Memoria

Al hablar de concurrencia nos es relevante sobre todo los accesos de lectura y escritura a los datos que se encuentran en memoria. Dentro de estos accesos podemos diferenciar los *débiles* de los *fuertes*. En los primeros los datos no tienen ninguna restricción de acceso y el compilador o máquina virtual o el procesador son libres de reordenar y acceder como lo deseen. Mientras que en los segundos ocurre lo contrario, el programador suele poner alguna restricción que restringe el ordenamiento automático por parte de los mecanismos antes mencionados.

En el primer caso puede ocurrir que no se haya tenido en cuenta el multiprocesamiento o que realmente el código este preparado para ejecución concurrente y sea irrelevante el orden de acceso.

Las siguientes combinaciones de operaciones sobre un mismo objeto compartido suelen generar problemas de concurrencia:

loadstore leer y guardar

loadload leer y leer

storeload guardar y leer

storestore guardar y guardar

Ejemplos de acceso

A continuación cito un ejemplo implementado para mostrar los posibles problemas de esos 4 casos y cómo se tuvieron en cuenta.

En el algoritmo se hace uso de una estructura que es una `cola`, se asegura que las operaciones de agregado y quitado de elementos son atómicas para que puedan llamarse desde el esquema de trabajadores concurrentes sin necesidad de bloqueos.

La operación *Poll* representa la lectura. La operación *Add* representa el guardar.

Posibles problemas al usar dos trabajadores, T1 y T2, de manera concurrente sobre una `cola elementos sin repetidos`:

T1 Poll & T2 poll En caso que dos quieran tomar un elemento de la cola, queremos que no sea el mismo sino uno y luego otro consecutivo y distinto . Tener en cuenta que si la cola no está vacía se toma un elemento y se elimina de la cola. Esto podría ser implementado mediante operaciones atómicas 3.1.3 o bien usando algún tipo de bloqueo.

T1 Add & T2 Add En el caso que ambos hilos quieran agregar exactamente el mismo elemento, se busca que éste cambio se vea reflejado un única vez. El agregar puede tener bloqueo para que la operación de agregar sea usada por un único hilo a la vez. Sino también se puede utilizar alguna operación atómica de comparación y agregado. Si el orden de agregado fuera relevante sería necesario un bloqueo de escritura si o si, en caso contrario se puede asegurar que ambos hilos puedan agregar sin necesidad de bloqueo.

T1 Add & T2 Poll En el comportamiento secuencial, si la cola esta vacía, queremos que T2 pueda tomar el elemento que esta siendo agregado por T1. En cambio, si la cola contiene algún elemento, el agregar debe poder ejecutarse, igual que el Poll. Para asegurar este comportamiento se podría agregar un semáforo al agregar elementos a la cola que al liberarse es verificado por el Poll. Sin embargo, si no fuera necesario tener en cuenta el comportamiento estricto secuencial podría ocurrir que el Poll no toma ningún elemento y esto debería ser contemplado en algún otro lado del código.

T1 Poll & T2 Add Es un caso análogo al anterior. En el comportamiento secuencial queda claro que si la cola esta vacía T1 no puede tomar ningún elemento y luego T2 agrega, y la cola queda con 1 elemento. Ahora bien, en acceso concurrente no nos afectaría que primero se agregue y luego se realice el Poll dejando la cola nuevamente vacía. Supongamos que T1 toma el elemento E que corresponde a la misma representación de lo que T2 intenta agregar. En secuencial no hay problema, la cola queda con el mismo elemento en la cola. Ahora bien en concurrencia, para evitar que T2 tenga problemas en agregar el elemento, habría que restringir el agregado hasta terminada la operación de Poll.

Operaciones Atómicas

Una operación atómica es una operación en la que un procesador puede simultáneamente leer una ubicación y escribirla en la misma operación del bus. Esto previene que cualquier otro procesador o dispositivo de Entrada/Salida escriba o lea la memoria hasta que la operación se haya completado.

El término atómico implica la indivisibilidad e irreductibilidad del proceso, ya que éste debe realizarse en su totalidad o en caso de ser interrumpido poder deshacer sus acciones de modo que fuese como si no se hubiese realizado acción alguna.

Como ejemplos de éste tipo de operaciones encontramos:

AtomicAdd/Sub Suma/Resta dos números y escribe el resultado en el primero. Devuelve el valor original del primer número.

AtomicMin/Max Toma el valor de los dos parámetros, computa el mínimo/máximo y lo escribe en el primer parámetro. Luego devuelve el viejo valor del primer parámetro.

AtomicAnd Lee los dos parámetros de la memoria. Computa el y-lógico y almacena el resultado en ambas direcciones. Luego devuelve el viejo valor del primer parámetro.

AtomicAnd/Or/Xor Lee los dos parámetros de la memoria. Computa el y/o/xor-lógico y almacena el resultado en ambas direcciones. Luego devuelve el viejo valor del primer parámetro.

En cada ejemplo, las tres operaciones de lectura, análisis, escritura, se hacen de forma atómica.

Concurrencia sin bloqueo

Tal como se detalló en los problemas comunes, hay casos en los que no se requiere asegurar un acceso secuencial a los datos, o no hay una condición de ganada para las operaciones. Luego muchas veces con asegurar que los datos pueden ser accedidos de forma concurrente es suficiente y no se aseguran post condiciones fuertes sobre los resultados de las operaciones.

Para cada implementación de concurrencia hay que tener en cuenta este detalle, ya que podríamos estar sobre bloqueando sin necesidad de asegurar una secuencialidad de acceso.

Al decidir programar sin bloqueos debemos preguntarnos si hay muchos hilos que van a escribir. En caso afirmativo podemos utilizar métodos atómicos RMW (leer, modificar, escribir), si además hubiese un ciclo que haga una comparación y modificación de forma atómica habría que contemplar el problema ABA. Este problema se da cuando la estructura a la que se accede de forma concurrente no esta preparada para que todos sus hilos vean reflejados todos los cambios.

Además hay que analizar si se van a usar múltiples procesadores compartiendo una misma memoria (modelo UMA), éste sería el caso más común hoy en día. En caso afirmativo, si se requiere que los datos sean accedidos de forma secuencial hay que implementar mecanismos que aseguren que todos los hilos accederán a la información más actualizada. Por ejemplo en `JAVA` se puede usar la palabra clave *volatile*, o utilizar tipos atómicos. Si no se requiere el acceso secuencial, podemos simplemente usar semáforos o barreras para acceder en el orden semántico necesario y asegurar el orden de acceso a memoria.

Si en la concurrencia solo tenemos productores y consumidores es probable que sólo necesitemos usar alguna estrategia de adquisición y liberación de recursos. Si hubiera estructuras más complicadas o más acopladas, entonces necesitaríamos restricciones de acceso a memoria totales.

3.2. Pseudocódigo de la implementación

En la sección 2 se mostró la implementación secuencial del algoritmo para resolver el juego; mientras que el algoritmo 2 muestra el pseudocódigo de la versión concurrente del mismo.

Notar que la operación central de actualización *best(v)* es la que se encarga de chequear los valores de los sucesores y obtener un posible valor de actualización del estado actual. Se puede ver el uso en la linea 21 del algoritmo 1. Esta función requiere recorrer todos los sucesores del estado para conocer su nuevo valor sugerido. Para reducir los tiempos, primero se chequea si el nodo es estable y luego se procede a actualizar. Además si el juego es perdedor, el nodo inicial es perdedor, la estabilización se considera terminada y el juego resuelto.

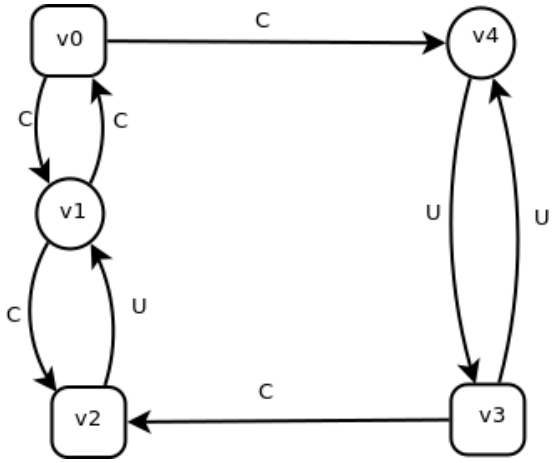

Fig. 3.1: Juego para mostrar actualizaciones

Veamos un ejemplo concreto de cómo pueden interactuar las actualizaciones, o sea, qué datos afectarán la estabilización de cada estado. Usamos para esto el ejemplo de la figura 3.1:

- Los estados que no tienen ejes en común se pueden actualizar en paralelo (por ejemplo v0 y v3).
- Los estados que pertenecen a asunciones del dominio serán mayor estricto que sus adyacentes (por ejemplo v0,v2 y v3).
- Los estados que cumplan los objetivos del controlador (por ejemplo v1 y v4) pueden tener igual valor que sus adyacentes.

La implementación de *wait* y *notify* entre cada consumidor puede hacerse utilizando un objeto en común para todos los trabajadores (ver algoritmo 3).

La fase se utiliza de forma dinámica, los trabajadores se pueden registrar y de-registrar tantas veces como sea necesario. Sin embargo el padre esperará que todos sus hijos se hayan de-registrado para continuar.

Notar además que cada trabajador consulta si puede continuar corriendo a través de un mensaje *canRun()*. El uso y costumbre es colocar un *while(true)* para el método *Run*. Sin embargo se decidió implementar para poder tener la opción de terminar los trabajadores antes de que la cola estuviese vacía y así tener la opción de agregar un *timeout* en el código. Al cortar anticipadamente no se asegura nada de los valores de ranking de cada estado.

Al igual que en el algoritmo original, cuando no se usa el *timeout*, el algoritmo concurrente cicla hasta que no quedan más elementos en la cola y cada trabajador completa y finaliza su trabajo.

3.3. Intuición de correctitud

3.3.1. Definiciones para mostrar correctitud

Teorema 3.1. *(Completitud del Algoritmo) Si hay una estrategia ganadora desde* $s_g \in G$*, entonces existe un sistema de ranking estable* $R_1, ..., R_m$ *donde para cada* $s_g \in S$ *y* $j \in \{1, ..., m\}$ *tenemos que* $R_j(s_g)$ *es o bien* ∞ *o bien* (k, l) *con* $k \leq max_l \mid \phi_l - (\psi_j) \mid$.

Algorithm 2 para resolver juegos SGR(1) de forma paralela

```
1: procedure SOLVEGAME(GAME=(STATES,TRANSITIONS),SAFE,GUARANTEES,ASSUMPTIONS)
2:     // Inicialización:
3:     for state : states do
4:         for g : guarantees do
5:             rank_g(state)← (0,1)
6:     for state : states do
7:         if state.isDeadEnd() then
8:             rank_g(state)← ∞
9:     // Encolar pendientes:
10:    for state : states do
11:        if ∃ g : guarantees / state ∉ g ∧ state ∈ assume_1 then
12:            pending.push(pair(state,g))
13:        if Γ⁻(state) = ∅ ∧ Γ⁺(state) = ∅ then
14:            for g : guarantees do
15:                rank_g(state) ← ∞
16:                pending.push(unstable_pred(state,g))
17:    // Estabilización:
18:    for n : maxWorkers do
19:        new Thread(rankBasedGameSolverWorker(pending, rankingSystem, phase))
20:    phase.arriveAndAwaitAdvance()
```

Algorithm 3 trabajador productor-consumidor para resolver juegos SGR(1)

```
1: procedure RUN()
2:     while canRun() do                          ▷ Puede reemplazarse por true
3:         if !pending.empty() then
4:             phase.register()
5:             consume()
6:             notify()
7:         else
8:             phase.arriveAndDeregister()
9:             wait()
       phase.arriveAndDeregister()
10: procedure CONSUME()
11:    Synchronized(pending)
12:        (state,g) ← pending.pop()
13:
14:    if rank_g(state) = ∞ then
15:        continue
16:    if isStable(rank_g(state)) then
17:        continue
18:    Synchronized(state)
19:        rank_g(state) ← increment(best(state,g),state,g)
20:
21:    pending.push(unstable_pred(state,g))
```

Demostración similar a [KPP05].

Definición 3.2. Para un juego G -con objetivos GR(1)- y conjunto de nodos V, sea un estado de G una función de ranking $\rho : V \to R_G$. Escribimos S_G para el conjunto de todos los estados de G.

3.3.2. Explicación de la intuición

Las actualizaciones del algoritmo concurrente sobre S_G se harán utilizando la misma ecuación de actualización que en el caso secuencial: 2.2. Dicha función calcula el mínimo o máximo valor dependiendo si el estado es controlable o no. Además tiene en cuenta si el estado pertenece al conjunto de asunciones.

En dicho cómputo lo que varía es del conjunto de estados sus valores de ranking a cada momento. En el algoritmo secuencial se puede notar que la actualización del ranking de un estado afectará en la futura actualización de un predecesor a dicho estado. Sin embargo, en el algoritmo concurrente, puede ocurrir que las actualizaciones de varios estados se hagan de forma simultanea. Por lo cual, intuitivamente podemos observar que si la ecuación *best* utiliza valores obsoletos podría no afectar el valor actual del nodo analizado y por lo tanto no modificar su valor en esa iteración.

Los casos que interesa ver son aquellos donde el estado analizado tiene sucesores que aún no fueron actualizados y poseen valores viejos u obsoletos. En el caso que todos sus sucesores ya fueron actualizados estamos en la misma situación que el algoritmo secuencial y la ecuación de *best* nos devuelve lo mismo.

En cada iteración se analiza un estado y su posible valor de actualización. Si logramos capturar todas las actualizaciones de dicho estado en algún orden que nos convenga, podríamos quedarnos con su última actualización y corroborar que dicho valor corresponda con el valor del algoritmo secuencial cuando el algoritmo de punto fijo dejó de iterar.

Para esto podemos construir una función que capture todas las actualizaciones que se hagan sobre todos los estados. Luego podemos analizar si dicha función para un estado en particular es siempre menor igual que el valor de ranking en el algoritmo secuencial y que si además siempre se respeta el orden parcial establecido para las actualizaciones, entonces podemos afirmar que el algoritmo concurrente converge y lo hace al mismo estado que el algoritmo secuencial.

Notar que al actualizar los valores, si para un estado se alcanza el valor infinito establecido para ese juego, dicho estado no será más actualizado y su valor de estabilización será el máximo ranking posible para dicho juego.

Además, es importante destacar que para estabilizar el valor de un estado se mira en orden todas las asunciones del objetivo. Al cumplirse que se observa infinitamente la asunción 1, se pasa a analizar la asunción 2. Luego, el orden necesario para nuestra función F será el orden de análisis de las asunciones.

3.4. Correctitud del Algoritmo Concurrente

El conjunto de estados S_G es una matriz finita, generada a partir del orden parcial $\rho_1 \leq \rho_2$ si y solo sí, para todo $v \in V$ tenemos $\rho_1(v) \leq_a \rho_2(v)$ en G, donde $a = max(asunciones)$.

$$(k_i, l_i) \leq_a (k_j, l_j) = \begin{cases} l_i \leq l_j & k_i = k_j \\ k_i \leq k_j & k_i \neq k_j \end{cases} \tag{3.1}$$

Ahora definimos la función $F : v \to R_G$ tal que $F_j(v)$ captura todas las actualizaciones que se hicieron sobre el estado v para el objetivo j, quedándose siempre con la última actualización realizada. Luego, se mostrará que F es monótona y que el mínimo punto fijo coincide con el computado por el algoritmo secuencial y que el concurrente computa el mismo estado.

Esta función computa el ranking del sucesor que luego va a ser usada para computar el sistema de ranking para ese estado en todo el juego. Para conseguir el valor final de ranking para todos los estados, la función de punto fijo debe converger y ser estable respecto de todas las garantías del objetivo.

$$F_j(v) = \begin{cases} \rho(v) & \text{v es estable} \\ increment(best(v,j)) & v \in Assumptions \\ best(v,j) & v \notin Assumptions \text{ y } v \notin \gamma_j \end{cases} \tag{3.2}$$

Un detalle de la ecuación 3.2 es que el caso $v \notin Assumptions$ y $v \in \gamma_j$, entonces su valor será (0,1), considerándose v estable.

Notar que al incrementar el valor de *best*, en el segundo caso de la ecuación, puede ocurrir un salto de la garantía analizada. Este salto ocurre cuando se cumple que la asunción en análisis es visitada infinitamente. Luego, el algoritmo debe continuar con la siguiente asunción para todas las garantías.

Se recomienda mirar la explicación teórica del uso de *increment* al final de la sección: 2.6.3.

Lema 1. Sea G un juego. Tenemos entonces:

1. Para todo ρ en S_G, $\rho \leq F(\rho)$
2. Para todo $\rho_1 \leq \rho_2$ en S_G, $F(\rho_1) \leq F(\rho_2)$

Ahora bien, usando éste lema, podemos formular la correctitud del algoritmo.

Lema 2. Sea un juego G con objetivos GR(1). Toda función ρ computada por el algoritmo concurrente, es el menor punto fijo de F para el juego G.

El cálculo de $F(\rho)$ con el lema 1 nos asegura que vamos a computar el menor punto fijo. y El lema 2 nos muestra que vamos a computar lo mismo que la función F. Luego tanto el algoritmo de punto fijo como la función que captura todas las actualizaciones finalizan con el mismo valor.

4. IMPLEMENTACIÓN

En esta sección se muestra como utilizar la herramienta *MTSA*, desde un punto de vista de usuario final y desarrollador. Además se muestran diagramas de la implementación desarrollada en ésta tesis, con detalles que especifican estructuras usadas y pruebas realizadas sobre las mismas.

4.1. Sobre *Modal Transition System Analyser*

De acuerdo a [DFCU08b], los sistemas de transición modales son modelos operacionales que distinguen entre acciones requeridas y prohibidas de comportamiento de un sistema a crear y comportamiento que aún no se conoce que el sistema tendrá que realizar. MTS, a diferencia de los modelos tradicionales de comportamiento, soporta razonamientos sobre comportamiento deseado del sistema en presencia de conocimiento incompleto. La herramienta MTSA soporta la construcción de dichos modelos, el análisis y elaboración de MTS.

El algoritmo fue implementado y probado en la herramienta MTSA [DL15].

Dado el soporte teórico y práctico de dicha implementación, se decidió continuar y ampliar la herramienta para tener suficientes datos para validar los resultados obtenidos con el nuevo algoritmo concurrente.

Dicha herramienta contaba con una implementación secuencial para el cálculo de estabilización de valores de *ranking*. Fue extendida para soportar ambas implementaciones a la vez para permitir comparar los tiempos y los resultados obtenidos con cada algoritmo. El pseudocódigo de la implementación original se puede ver en el algoritmo 1.

La herramienta MTSA se utilizó también para modelar los LTS y sintetizar los controladores.

MTSA está desarrollado actualmente en JAVA. El soporte académico de la herramienta y la confianza en sus resultados hizo que se optara por implementar el algoritmo en esa plataforma, con las herramientas de JAVA que estuviesen disponibles para mejorar los accesos concurrentes a las estructuras.

4.1.1. Descarga y Compilación

La herramienta tiene configurado Maven y GIT. Para descargar la última versión se sugiere buscar y clonar el repositorio GIT en la web de MTSA: `http://mtsa.dc.uba.ar/`.

Usando el comando `mvn exec:java` se puede compilar y ejecutar la aplicación para usar la última versión del repositorio descargado.

Los tests existentes se corren automáticamente al armar el paquete `mvn package`. Los archivos usados por los tests se encuentran en la carpeta de ejemplos de la aplicación, dentro de `/src/test/java/MTSATests`.

4.2. Modo de uso de MTSA

4.2.1. Interfaz Gráfica

Desde la interfaz gráfica de MTSA (ver figura 4.3)se puede ingresar manualmente el número de trabajadores deseados para el cálculo concurrente. El número sugerido en la interfaz será la cantidad máxima de cores disponibles según lo que le reporta la máquina virtual de JAVA. Por ejemplo en la figura 4.2 el número de cores virtuales es 4.

Se sugiere que una vez escrito el LTS se corrobore la sintaxis presionando el botón 4.1. Para compilar se puede utilizar el botón con la letra `C` mayúscula. Cuando se requiere componer el ambiente con los objetivos, por ejemplo: `controller ||C = (BISCOTTI) ~ G1`, se debe utilizar el botón ||.

Una vez compuesto o compilado el LTS se puede visualizar en las solapas: Draw o Layout.

Fig. 4.1: Botón Parse

Fig. 4.2: Composición

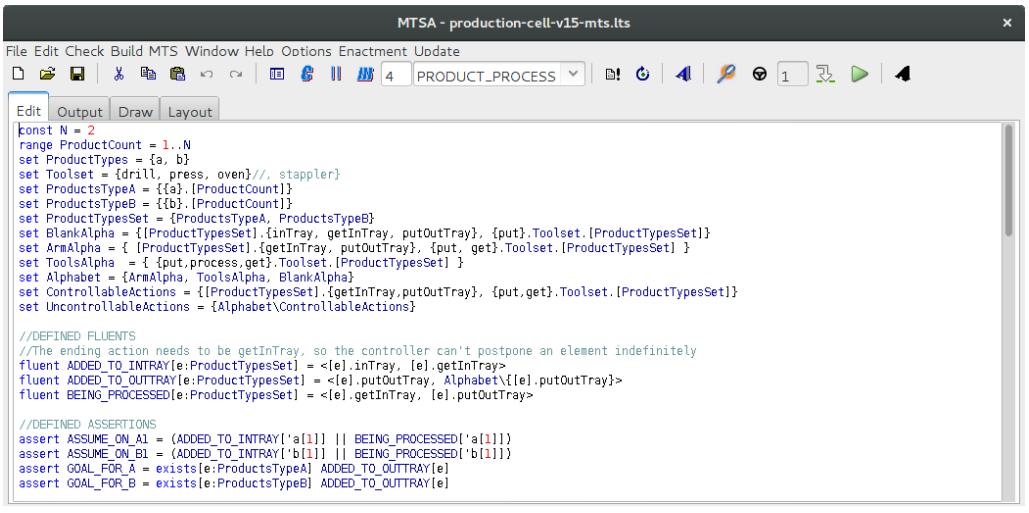

Fig. 4.3: Interfaz MTSA con 4 cores virtuales

4.2.2. Consola

Al correr desde una consola se puede agregar el parámetro `--threadCount` seguido del número máximo de hilos a usar.

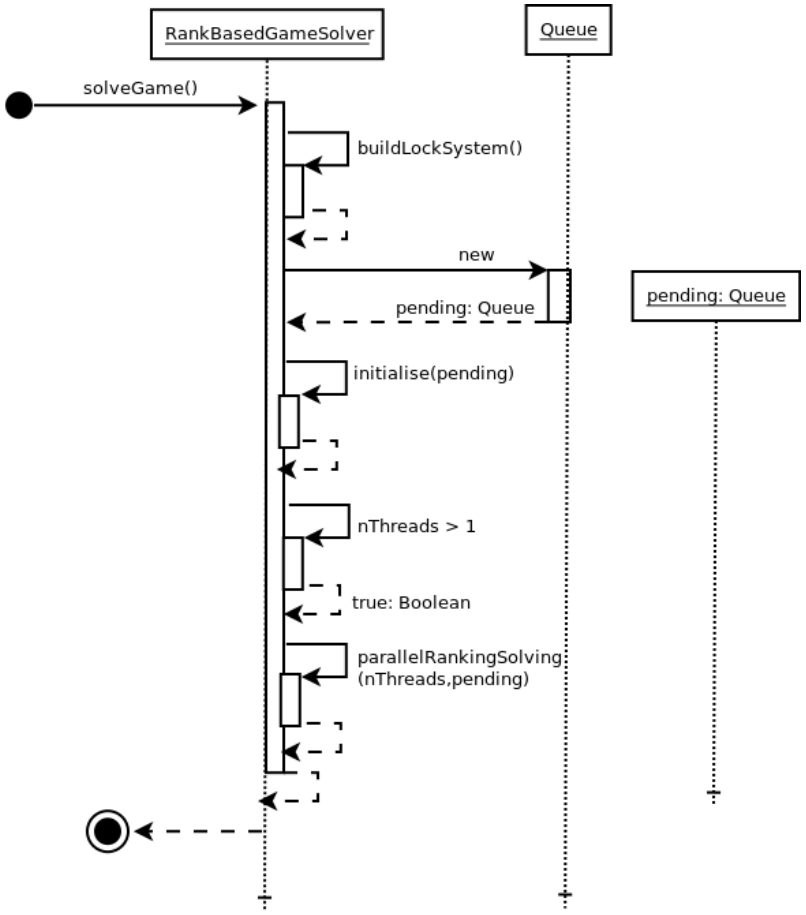

Fig. 4.4: SolveGame principal

Éste modo es el menos usado, ya que al sintetizar se desea poder visualizar el controlador resultante. Sin embargo, el uso de la terminal para correr los experimentos fue sumamente útil para poder usar computadoras remotas.

4.3. Diagramas de Secuencia Implementación Concurrente

En la siguiente sección se detalla en forma de diagramas de secuencia los mensajes enviados por cada objeto y las áreas sincrónicas utilizadas durante la estabilización del ranking.

4.3.1. Inicio de ejecución. Principal

A continuación se puede ver la secuencia principal (figura 4.4) que da inicio a la solución concurrente o bien secuencial.

En el diagrama se muestra el caso de uso un número de trabajadores mayor a 1, luego el algoritmo envía el mensaje *parallelRankingSolving*.

Se muestra también que la creación de la cola inicial de objetos a procesar esta a cargo del RankBasedGameSolver.

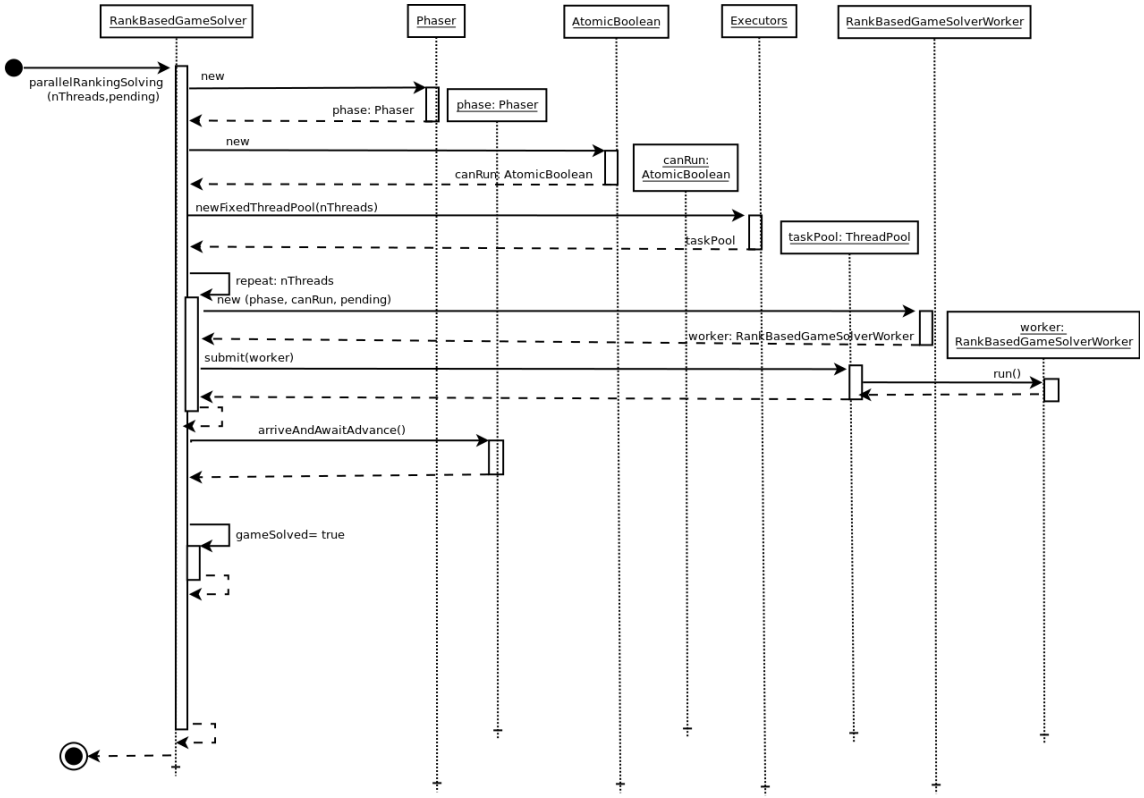

Fig. 4.5: Creación de workers y sincronización de fase

4.3.2. Creación de trabajadores, primera sincronización

En el diagrama 4.5 se puede observar el inicio de los trabajadores que harán el trabajo de forma concurrente. Antes de poder darle inicio a dichos trabajos, es necesario la creación de algunos objetos utilizados a modo de sincronización, por ejemplo: Phaser, AtomicBoolean.

Algo a tener en cuenta es que todos los trabajadores tienen acceso a los objetos de sincronización, incluida la cola de elementos pendientes. Todos estos objetos serán leidos y escritos en forma concurrente por los trabajadores.

El mecanismo de lectura y escritura concurrente de Phaser y AtomicBoolean viene dado en la implementación de JAVA. También es el caso de la cola utilizada: ConcurrentLinkedQueue.

4.3.3. Forma de consumo y escritura de cada trabajador

En esta sección se muestran dos diagramas. Por un lado se puede observar qué ocurre cuando aún hay elementos pendientes en la cola que necesitan ser procesados (ver figura 4.6), y por otro se muestra el mecanismo de fin al vaciarse la cola (ver figura 4.7).

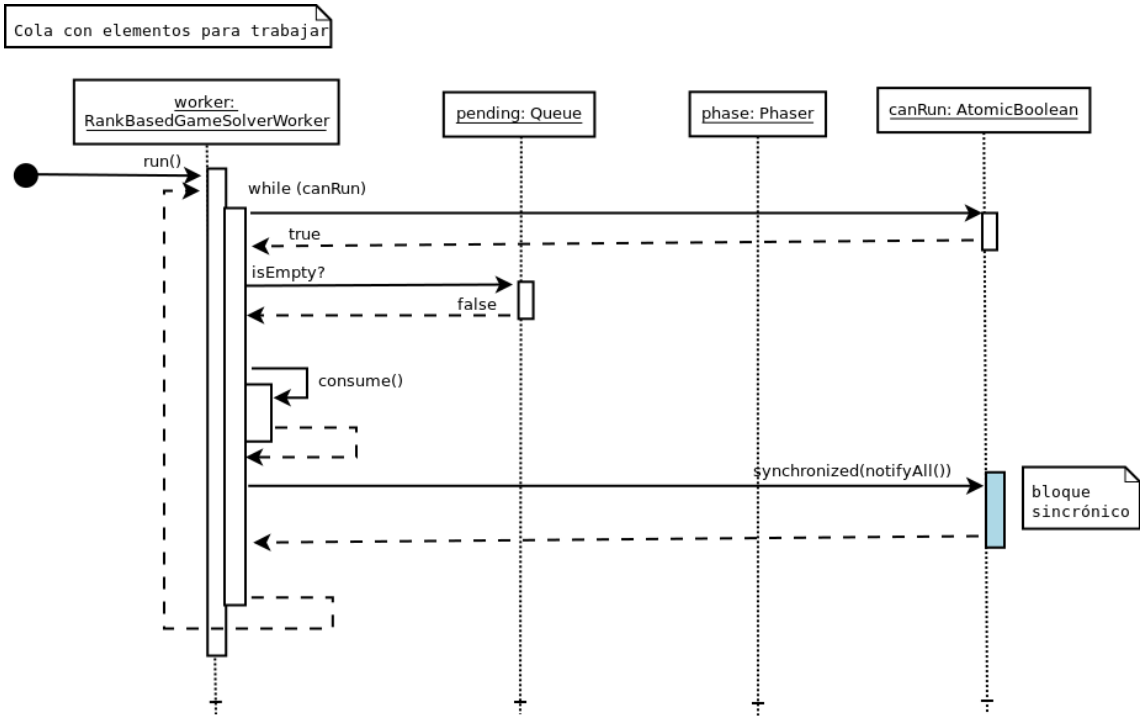

Fig. 4.6: Trabajador con elementos pendientes en la cola

4.3.4. Análisis de estados de la Cola

En la figura 4.8 se observa como se consume un estado de la cola de pendientes. Se muestra el ciclo en el cual se analiza el valor actual de dicho estado y el valor potencial de actualización. En el caso de ser necesario se actualiza el valor de dicho ranking y se agrega a sus estados predecesores a la cola de pendientes para revisión.

4.3.5. Actualización de ranking

Este método (ver figura 4.9) es muy importante ya que fue pensado para que el bloqueo de escritura de un determinado estado no afecte a todo el juego a la vez. Se buscó que muchos estados puedan ser actualizados a la vez y no haya escrituras espurias. No nos importa el caso que un trabajador lea un valor de ranking desactualizado, ya que de ser necesario se volverá a analizar en una futura iteración.

4.4. Detalle de clases implementadas

Esta sección complementa los diagramas mostrando la ubicación precisa de los métodos en la herramienta MTSA. Los detalles de implementación fueron importantes en este algoritmo, debido a que un cambio en la forma de sincronización o acceso afectaba los tiempos y los deadlocks en las ejecuciones.

Se entiende que el código puede estar sujeto a mejora (ver sugerencias en sección 7).

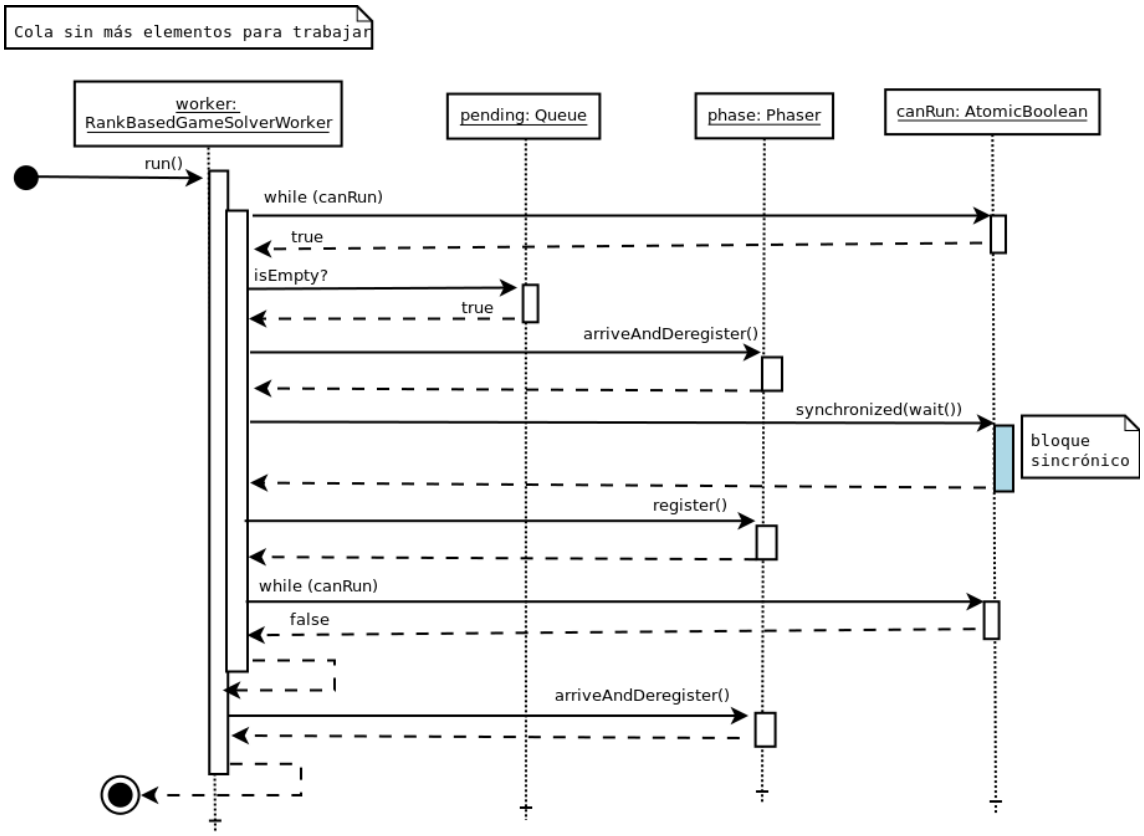

Fig. 4.7: Trabajador con la cola vacía

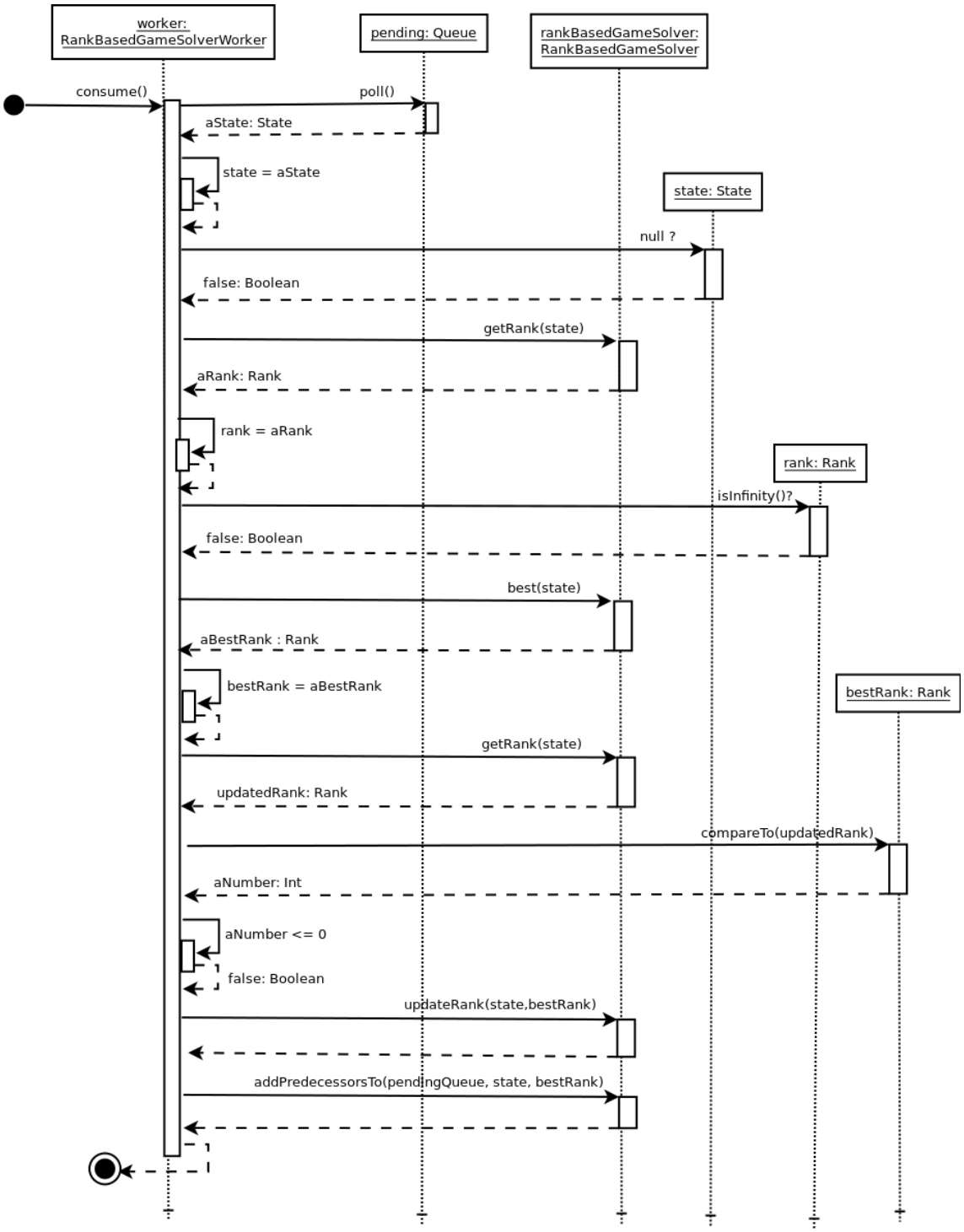

Fig. 4.8: Consumir estado no estable de la cola

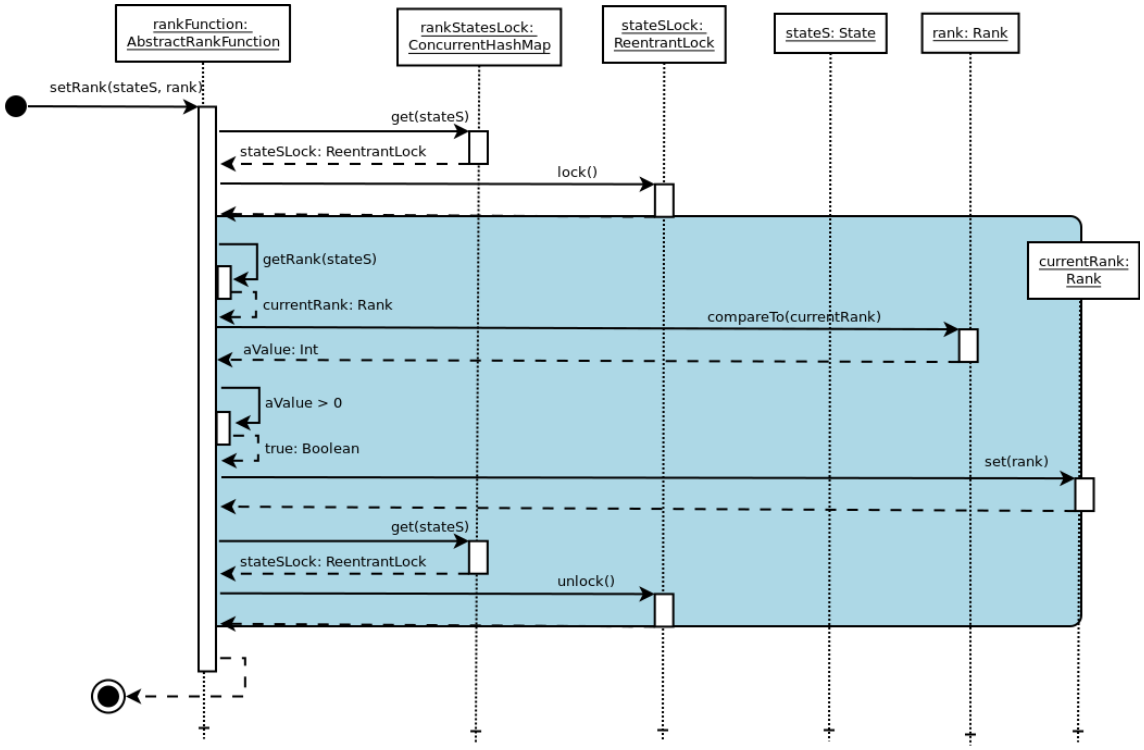

Fig. 4.9: Set Rank

4.4.1. package ltsa.MultiCore

Computer Options

Para poder comprobar los distintos tiempos variando la cantidad de trabajadores concurrentes se implementó una clase con las opciones de configuración que puede ser editada con una llamada pública. De esta manera, tanto desde consola como desde la ventana de la herramienta o desde los métodos para pruebas se podía variar la configuración.

Test MultiCore

Clase *main* creada para poder armar el JAR exclusivo para pruebas. La forma de ejecutarlo en consola es la siguiente:

```
TestMultiCore --worker NUMBER_OF_WORKERS --file LTSFile
TestMultiCore --worker NUMBER_OF_WORKERS --directory DIR
```

Los registros de tiempos son guardados en un directorio *log* donde se ejecute.

4.4.2. package MTSSynthesis.ar.dc.uba.util

Concurrent Set Queue

Debido a la necesidad de que los *add* y *pop* en la cola sea inmediato, se implementó una clase en `JAVA` con interfaz de cola pero colaborador interno un diccionario de acceso concurrente. El diccionario internamente es visto como un conjunto.

El agregado de elementos se hace simplemente agregando al conjunto. Dado que éste no tendrá elementos repetidos no hará falta agregar una verificación si el elemento existe o no. Esta operación es atómica.

Para poder quitar elementos de la cola necesitaremos primero verificar si éste existe y luego quitarlo. Esta operación podría traer problemas si hay dos intentando eliminar el mismo elemento, agregar y quitar o quitar y agregar. Dada la dependencia de datos que se da en esas operaciones es necesario utilizar un bloqueo para que sólo uno de los accesos sea exitoso.

Referirse a la sección 3.1.3 para más detalles de los problemas comunes. Notar que en los casos de Poll y Add, no fue necesario agregar bloqueos.

T1 Poll & T2 poll Se utilizó un iterador sobre un conjunto. Cada hilo tendrá su iterador cada vez que quiera realizar el Poll. Luego la implementación del iterador concurrente sobre el conjunto nos asegura que se obtendrán dos elementos *seguidos* y no el mismo.

T1 Add & T2 Add Lo único que buscamos es que no se pierdan las operaciones de agregado. No nos fue relevante el orden ya que a lo sumo habrá una iteración más para verificar los valores de los nodos. Para asegurar esto se usó un conjunto de acceso concurrente, donde para agregar simplemente se cambia un valor de verdad. El agregado es casi inmediato.

T1 Add & T2 Poll La implementación contempló el caso en que la cola esta vacía y el Poll se hace antes que el Add. Para que todos los hilos conozcan que un nuevo elemento esta disponible y no se queden dormidos se les avisa mediante una señal. Luego, si alguno estaba inactivo por no poder trabajar, retomará sus actividades.

T1 Poll & T2 Add No nos importo el orden de estas operaciones. En el caso de que T1 toma el elemento E que corresponde a la misma representación de lo que T2 intenta agregar, T1 tomaría en verdad los valores actualizados de E, ya que al agregar las modificaciones ya fueron grabadas en el elemento E.

En particular la implementación que se usó fue `ConcurrentHashMap`. Esta clase extiende de diccionario e implementa concurrencia y serializable. Es una tabla de *hash* que soporta concurrencia completa de pedidos y ajustable a el nivel de actualizaciones concurrentes esperado. En particular, se configuró que se adapte a la cantidad de hilos concurrentes de acuerdo a cada ejecución.

Sin embargo, a pesar de que las operaciones son *thread-safe*, las lecturas no usan bloqueo y no hay ningún soporte para bloquear la tabla entera que asegure un único acceso a la vez a dicha tabla. Se sugiere usar esta tabla cuando se requiere asegurar acceso de múltiples hilos pero no hay condiciones de sincronización en dichos accesos.

Las operaciones de lectura podrían solaparse, debido a la inexistencia de bloqueos, con las de actualización. Las lecturas reflejan los resultados de las operaciones completas más recientes. Los iteradores devuelven los elementos de acuerdo a su estado en la tabla de hash en cierto punto desde o en el momento de creación de dicho iterador. No arrojan la excepción *ConcurrentModificationException*. Sin embargo, los iteradores fueron diseñados para ser usados desde un único hilo a la vez.

La tabla de hash es internamente particionada para soportar el nivel de concurrencia deseado sin contingencias. Debido a que la ubicación de los elementos es esencialmente aleatoria en cada momento el nivel de concurrencia puede variar. Se sugiere indicar el

tamaño de toda la tabla al momento de su creación para evitar que haya que rearmarla durante la operación de uso. En la implementación se usó una tabla con nivel de concurrencia igual a la cantidad de hilos usados.

4.4.3. package MTSSynthesis.controller.game.model

RankBasedGameSolverWorker

En esta clase se implementa el funcionamiento de cada trabajador. Estos serán instanciados las veces que sea necesario para que el trabajo se haga concurrente.

Cada trabajador tiene la responsabilidad de consumir elementos de la cola de pendientes hasta que la cola esté vacía. Una vez que la cola esta vacía se pueden poner a esperar por más trabajo, y si todos están efectivamente esperando trabajo implica que el juego fue resuelto y se pueden cerrar.

La clase madre que instancio a los trabajadores será la encargada de liberar los hilos cuando detecte que todos sus hijos están ociosos.

En las implementaciones de evaluación se utilizó un bloqueo sobre la cola para poder cortar cuando se alcanzaba cierto tiempo. Fue implementado utilizando un bloque sincrónico sobre la cola. Pero no se requiere en la implementación definitiva. A continuación dejo el código de referencia:

```
synchronized (pendingQueue) {
    long time = (System.currentTimeMillis() - startTime);
    if (time > allowedTimeToRun) {
        System.out.println("Me dio timeout");
        break;
    }
}
```

Integrity Tests

Dada la complejidad del código, se implementaron algunos tests para asegurar que futuros cambios conservan la funcionalidad y se mantiene la concurrencia.

Todos los tests están en la carpeta /test/ y organizados en el siguiente paquete.

4.4.4. package MTSSynthesis.controller.game.model

1. Controlador resultante de corrida secuencial es bisimilar a correr con dos trabajadores
2. Controlador resultante de corrida con 2 y 4 trabajadores son bisimilares
3. El objetivo deseado para el controlador es cumplido en el controlador sintetizado
4. Si no existe controlador en la corrida secuencial, no existe tampoco al usar 2 trabajdores.

En los tests uno y dos, lo único que se busca es que el mismo controlador sea bisimilar. Para esto se asume que el controlador está etiquetado con `C`. Tener en cuenta que para poder verificar bisimilaridad es necesario componer las máquinas. Esto puede requerir

mucha memoria y no en casos de grafos muy grandes no poder realizarse. En esos casos dará un error: `java.lang.AssertionError: Too big to check bisimilarity: 8491`

En tercer se requiere que haya un assert llamado TESTGOAL. Esto fue agregado como alternativa a los primeros tests, para poder verificar casos donde hay muchos nodos en el controlador resultante.

El cuarto test verifica que si en la corrida secuencial no existía un controlador entonces tampoco debe ser posible hallarlo usando más trabajadores. Esto se hizo especialmente para los casos donde se usaban LTS no determinísticos. De todas maneras la intención del algoritmo es sólo usarse en juegos con objetivos GR(1) determinísticos.

5. EVALUACIÓN DE LA IMPLEMENTACIÓN

5.1. Metodología

Para poder evaluar la eficiencia del algoritmo concurrente se ejecutaron pruebas utilizando distintos casos de estudio. En todos los casos se buscó que los casos en una corrida secuencial tarden mas de un segundo. Se usaron casos que fuesen parametrizables en cantidad de estados y conexiones, para comparar los tiempos de acuerdo al crecimiento del juego.

Con el fin de evaluar el rendimiento del algoritmo, se tomaron casos de la literatura, generados por otros investigadores que también utilizan MTSA para sus pruebas, y casos generados artificialmente.

La ventaja de utilizar ejemplos artificiales, morfológicos, ha sido sustancial a la hora de analizar pros y contras de la paralelización.

Notar que el sólo hecho de calcular el promedio y el desvío estándar presupone una ejecución de tiempos que sigue una distribución normal. No se han realizado pruebas estadísticas de normalidad sobre los datos de entrada.

Se recomienda para futuras pruebas utilizar datos de estadística descriptiva, como la mediana, el mínimo y el máximo de cada muestra.

Para cada ejemplo se evaluó:

1. Generación de grafo que representa el ambiente
2. Traducción de objetivos a fluents
3. Diez iteraciones:
   a) Generación de juego a partir de los dos ítems anteriores
   b) Estabilización del ranking del juego
   c) Cálculo de tiempo total de estabilización
   d) Anotación de cantidad de elementos analizados hasta la estabilización
4. Cálculo de promedio de tiempo de estabilización
5. Cálculo de desviación estándar de los tiempos
6. Cálculo de promedio de cantidad de elementos analizados

Los tiempos de generación del juego fueron descartados para ésta evaluación ya que nos centramos en la estabilización de los valores del juego.

Se anotó la cantidad de estabilizaciones necesarias realizadas. Este dato fue relevante a la hora de analizar las diferencias de tiempos. Los valores de estabilización finales son iguales en cada juego, sin importar la cantidad de trabajadores -los controladores son bisimilares. La intuición nos dice que la cantidad de estabilizaciones debería ser similar en todos los casos, sin importar la cantidad de trabajadores, sin embargo esto no ocurrió así.

Se tuvo en cuenta que los mayores tiempos se obtienen cuando se fuerza el algoritmo al peor caso. En este sentido, se eligieron juegos donde no existe controlador, pero las

transiciones son controlables. El impacto en el incremento del tiempo en la estabilización se debe a que conocer si un estado es estable o no, requiere el mínimo valor de sus sucesores y si no existe controlador todos sus hijos serán eventualmente infinito. Luego, estaremos en el peor caso.

Al evaluar juegos donde sí existía controlador, la estabilización podía concluir antes que visitar todos los nodos, por lo tanto, no podíamos vincular la variación de tiempos con el tamaño o estructura del juego.

5.2. Análisis previo

Analizando el código original de la herramienta MTSA, se vio que las estructuras de datos utilizadas no eran las mejores respecto a la velocidad de accesos (lecturas y escrituras).

Luego, el primer análisis fue hecho sobre distintas implementaciones de "cola". Se puede ver la sección donde se detalla la implementación finalmente adoptada: 4.4.2. A modo de ejemplo en el código original los ejemplos de juegos con 40.000 nodos no terminaban en menos de 30 minutos. Se decidió omitir los resultados originales de MTSA por considerarlos outliers dentro de los nuevos resultados obtenidos.

Se compararon entonces distintas implementaciones de cola [1] usando el algoritmo secuencial, pero teniendo en cuenta que la estructura debía soportar accesos concurrentes. Se puede leer más detalle en la sección: 3.1.3.

Respecto a la nueva implementación, las preguntas a responder fueron:

1. ¿Qué implementación de cola conviene usar para asegurar las sincronizaciones de acceso deseadas?

2. Para cierta implementación de cola, ¿conviene agregar todos los predecesores o hacer el chequeo de estabilidad previo a agregarlo?

La primer pregunta se responde analizando cuánta responsabilidad sobre acceso concurrente se iba a delegar a la cola. En otras palabras, se puede usar una estructura que no garantice nada y que los accesos sean "cuidadosos", o bien que justamente la cola misma nos asegure el no solapamiento de escrituras y lecturas sobre ella.

Se probaron casos en los que la cola aceptaba lecturas y escrituras concurrentes pero no se aseguraba que las lecturas se hicieran de forma única por dos trabajadores distintos. También se analizaron casos donde todos los accesos de escritura o lectura estaban bloqueados, y podían ser realizados por un único *worker* a la vez.

Para la segunda pregunta, en el algoritmo original se notó que se agrega sólo los estados que aún requieren análisis y además se mantiene sólo una copia de cada estado en la cola. O sea, sólo agrega los estados que no son estables, para lo cuál hay que analizar su estabilidad respecto de todos sus sucesores al momento de agregarlos, y que no se encuentran en ese momento en la cola. Dado que, el hecho de analizar estabilidad requiere mucho cómputo en cada iteración y que agregar elementos no estables no afecta la finalización del algoritmo de punto fijo, se abre la posibilidad de evaluar el algoritmo agregando todos los elementos sin hacer el chequeo de estabilidad o realizándolo.

[1] Se probó con colas que tuviesen conjuntos, arreglos, vectores, e incluso que mantuviesen dos estructuras a la vez, una con repeticiones y otra sin, para analizar distintas velocidades de acceso.

Para poder optimizar los tiempos, lo que se buscaba entonces era que, para el caso donde hubiese que agregar los nodos se pudiese asegurar que la pertenencia fuese menor estricta a $O(n)$, para no tener que recorrer todos los elementos pendientes. A partir de esa opción se analizó el uso de un conjunto que tuviese interfaz de acceso de cola.

5.3. Casos de Estudio

Las pruebas se realizaron en diferentes casos de estudio, expuestos a continuación.

Por un lado se seleccionaron ejemplos con aplicaciones reales, tomados de referencia para regresión. Es el caso del Controlador de Tránsito y el Hospital. Estos casos no conocemos la morfología del juego, sólo podemos arrojar hipótesis teniendo en cuenta características del juego; por ejemplo: cantidad de asunciones, cantidad de garantías, tamaño inicial de la cola o cantidad de estados del juego.

Por otro lado se tomaron ejemplos donde la morfología del juego a sintetizar sí nos era conocida. Luego, podíamos arrojar hipótesis sobre los resultados que se obtuvimos basándonos en recorridos del grafo. Es el caso de: el árbol, la clique y la escalera.

A continuación se describe cada ejemplo con más detalle de su estructura y su elección.

5.3.1. Árbol

Por un lado se armaron juegos cuya estructura simulase un árbol, con un nodo inicial y ramas que no estaban conectadas entre sí. Dicho árbol finalizaba en un estado trampa, o sea con un ciclo a sí mismo. Se compararon dos configuraciones basadas en éste árbol:

- Pocas ramas saliendo del nodo raíz y cada rama con muchos nodos intermedios.
- Muchas ramas saliendo del nodo raíz y cada rama corta.

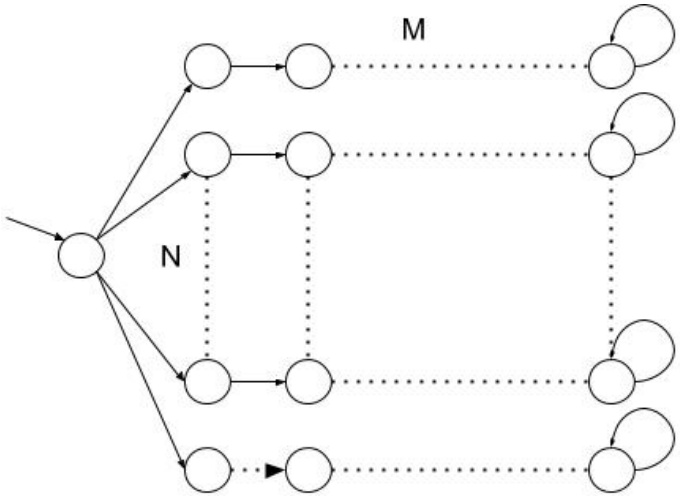

Fig. 5.1: Esquema de juego tipo árbol con estado trampa N x M

La idea de estos dos casos, pensando de manera secuencial, es la siguiente: las ramas cortas van a detectar que no son ramas ganadoras más rápido, mientras que cuántos más nodos tenga la rama más voy a tardar en saber que esa traza me hace perder. Por otro lado si la raíz tiene muchos hijos tiene que esperar a que todos sus hijos estén estables

para detectar si es perdedor o ganador. Además en el caso de los árboles la mayoría de los estados tienen un único sucesor y un único predecesor, haciendo que cada consumir y actualizar la cola sea casi inmediato.

5.3.2. Clique

El otro tipo de juego que se evaluó es la clique. La estructura fue usada ya que cada estado tenia la cantidad máxima de nodos sucesores y predecesores. Esto aumenta y dificulta el análisis de cada nodo, ya que los tiempos de evaluar el mejor valor para estabilizar implica conocer el valor de todos mis sucesores. El tiempo requerido para evaluar si un nodo esta estable, esta directamente ligado a la cantidad de sucesores.

Luego, el tiempo para saber si un nodo pierde esta vinculado con la longitud de la vuelta a sí mismo. Para poder saber que un nodo es perdedor el algoritmo debe dar toda la vuelta y volver a sí mismo sin haber visitado ningún estado que cumpla el objetivo.

Un detalle, es que se usó una clique no reflexiva, ya que es fácil notar que estos casos serán más fáciles de resolver que los que no tienen ciclos en cada nodo. Las transiciones que vuelvan al mismo nodo resultan en estrategias de ganada (o de pérdida) parciales que los otros nodos pueden usar para poder ganar (o perder) en la región.

5.3.3. Escalera

El espíritu de este ejemplo, tomado de la literatura [OF], es un intermedio entre la clique y el árbol. Cada nodo tiene 2 hijos: nodo i tiene sucesores: i+1 e i+2. En ambos casos se toma el módulo para que quede cíclico.

Los juegos de este tipo se eligen para evaluar los tiempos porque en juegos de n nodos, hay 2^n estrategias posibles de ganar, y sólo un jugador puede ganar el juego mientras el otro pierde. Este tipo de juegos son muy difíciles de resolver para heurísticas que adivinan la estrategia.

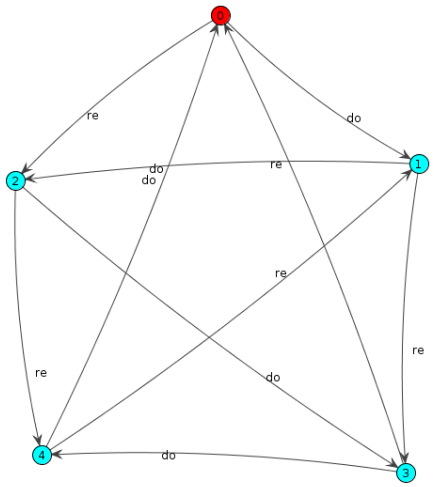

Fig. 5.2: Esquema de juego tipo escalera

De manera progresiva se pueden ir aumentando la cantidad de conexiones hijas, y llevarlo hasta el extremo de grafo clique. Ésto ayudó en la evaluación de nuestro algoritmo concurrente.

5.3.4. Centro médico

Éste ejemplo fue tomado de la literatura. La morfología del juego varía y se podía construir un controlador a partir del juego planteado. Recordemos que cuando sí existe un controlador los tiempos tienden a mejorar y acortarse ya que se consigue estabilizar todo el sistema sin recorrer todos los posibles caminos.

Los casos de la literatura, donde sí se puede obtener un controlador para los objetivos propuestos, fueron usados como testigos de que el algoritmo paralelizado continuaba generando un controlador bisimilar al hallado con el algoritmo secuencial. Y también se usaron como ejemplo de un caso real donde los tiempos mejoraron usando más trabajadores concurrentes para sintetizar el juego.

5.4. Implementación de Evaluación

5.4.1. Detalles de hardware

Una parte de los casos se ejecutaron en una instancia de *Amazon*: i2.2xlarge[2], con 8 cores virtuales y 61 GB de RAM.

La máquina virtual de JAVA fue limitada a usar 50 GB de RAM para poder manipular objetos grandes y así poder calcular el ranking en juegos con muchos estados y muchas conexiones.

Además se corrieron (pero no se exponen en este trabajo) más pruebas sobre una computadora de escritorio con 6 cores reales y 16GB de RAM. En este caso se limitó a 8GB el máximo uso de memoria de JAVA.

Para finalizar la tesis y armar las tablas comparativas se volvieron a correr todos los ejemplos utilizando el cluster del Centro de Cómputos de Alto Rendimiento [3].

Detalles de hardware utilizado:

- CPU: AMD Opteron 6320 x2
- RAM: 64GB ECC DDR3 1600MHz
- Mother: Supermicro H8DGi
- GPU: nVidia Tesla K20c x2
- HD: WDC Blue 500GB 7200RPM 16MB cache

5.4.2. Código de pruebas

Los tiempos se midieron dentro de la herramienta utilizando la hora de la computadora al inicio y fin del algoritmo. Para asegurar que los resultados no son azarosos y podían ser usados en éste trabajo, se buscó cuantificar la desviación estándar de los resultados usando el promedio de 10 corridas.

[2] https://aws.amazon.com/es/ec2/instance-types/

[3] `http://cecar.fcen.uba.ar/`

Se agregó un *timeout* al código original para poner un tope de media hora para la estabilización del ranking. Dicha mejora no se muestra en el pseudocódigo 3. Además en las pruebas de tiempos se excluyó la generación del controlador, siendo ésta una parte que se sugiere mejorar en un futuro (ver 7.2).

5.4.3. Profiling

Una vez implementado la primer versión del algoritmo concurrente, se decidió continuar el análisis un poco más profundo sobre la implementación especifica del algoritmo. Para esto se utilizó la herramienta JPROFILER[4].

Una técnica utilizada para conocer qué partes del algoritmo son las más utilizadas y cuántos recursos, tiempo y memoria utiliza cada método es el *profiling*. Se detectó que una de las llamadas más recurrentes era al método que actualizaba la cola de elementos pendientes. Dicho método se llama cada vez que se consume un elemento de la cola y puede ser llamado tantas veces como predecesores tenga el estado que esté siendo analizado.

El algoritmo en principio no sugiere si la cola puede o no tener elementos repetidos. La implementación y la bibliografía consultada [HPK11], sugerían mantener una cola con elementos únicos. La implementación concurrente de dicha cola, como es de suponer, no es trivial ya que se requiere por un lado un acceso secuencial a los datos y por otro que instantáneamente: sin bloquear el consumir nuevos elementos, se pueda saber si el objeto ya tiene su par en la cola o no. Dicho de otra manera, necesitamos que la cola acepte lecturas y escrituras concurrentes.

Para analizar las distintas implementaciones de la cola se comparó solamente los tiempos en una corrida secuencial. Se compararon los tiempos de dos implementaciones distintas de colas comparando con los tiempos de las corridas originales de MTSA. Las dos implementaciones analizadas fueron:

Concurrent Linked Queue Implementación de cola concurrente de Java con repetidos.

Unique Queue Implementación propia de cola utilizando como colaboradores una cola y un conjunto para evitar repetidos.

La implementación original sólo agrega elementos que no están estables. Lo que se verificó es que en todos los casos la cola que internamente no permite elementos repetidos era más rápida que la versión donde se recorre la cola y se verifica la existencia de dicho objeto antes de agregarlo. Esto es bastante obvio a partir de los ordenes de agregado en Colas y Conjuntos.

[4] http://www.ej-technologies.com/products/jprofiler/overview.html

6. RESULTADOS

El objetivo principal de este trabajo era implementar una solución del algoritmo de estabilización que pudiese correr en forma concurrente. Una de las consecuencias deseadas de esta implementación era que efectivamente el hecho de paralelizar disminuyese los tiempos de las ejecuciones de manera proporcional a la cantidad de procesadores usados. Obviamente, esto muchas veces no es posible, pero era lo deseable.

Teniendo en cuenta ese objetivo en primera instancia se buscó que efectivamente el algoritmo arrojase resultados correctos. Luego se pasó a optimizar la implementación para obtener menores tiempos de ejecución.

6.1. Comparación de tiempos

El análisis de tiempo (reloj) presentado a continuación se hizo teniendo en cuenta una implementación de cola sin contingencia y evaluando las dos opciones planteadas en la segunda pregunta: agregando todos vs. agregar sólo los no estables.

En esta sección se pueden observar las comparaciones promedio de tiempos entre 10 corridas seguidas, por cada uno de los casos de estudio evaluados.

En el apéndice, sección: A, se pueden ver los resultados de cada una de las corridas. Notar que los gráficos sólo muestran los valores promedio de todas las iteraciones. Las desviaciones estándar de los errores son en la mayoría de los casos menores al 1 % y en algunas excepciones menores al 20 %. Se entiende que esto no era deseado y que se debería haber seguido iterando hasta conseguir menor valor de desviación.

6.1.1. Árbol

En este ejemplo se ve que si bien en el caso de ejecución secuencial agregar todos los elementos es efectivamente más rápido que el caso de verificar la estabilidad y luego agregarlo, a la larga, cuando agregamos más *workers* la diferencia es mínima.

Notar que, a pesar de que la máquina usada tenía 16 núcleos, el caso de 16 trabajadores concurrentes, en promedio es peor que sólo 8. Podemos observar que la cantidad de iteraciones de punto fijo se incrementa en alrededor de cuatro millones en el algoritmo secuencial (agregando todos los elementos) a 7 millones en el caso de 16 hilos. Esto impacta negativamente en los tiempos.

Además se planteó una hipótesis sobre el orden de análisis de los nodos. Es decir, el hecho de evaluar los nodos secuencialmente en el orden agregado a la cola influye positivamente en el tiempo requerido para estabilizar todo el juego. Para esto basta ver que el tiempo de estabilización disminuye al incrementar la cantidad de hilos: $tiempoTotal/cantidadIteraciones$. Se deja como interrogante para un trabajo futuro, ver sección 7.2.

Notar que en los casos planteados justamente se invirtió la cantidad de ramas y su longitud. Se comparó el tiempo requerido para estabilizar cada caso y se ve que en el caso de agregar sólo los elementos no estables los tiempos son similares. En el caso de agregar todos los elementos el árbol de 20 x 100 empeora los tiempos.

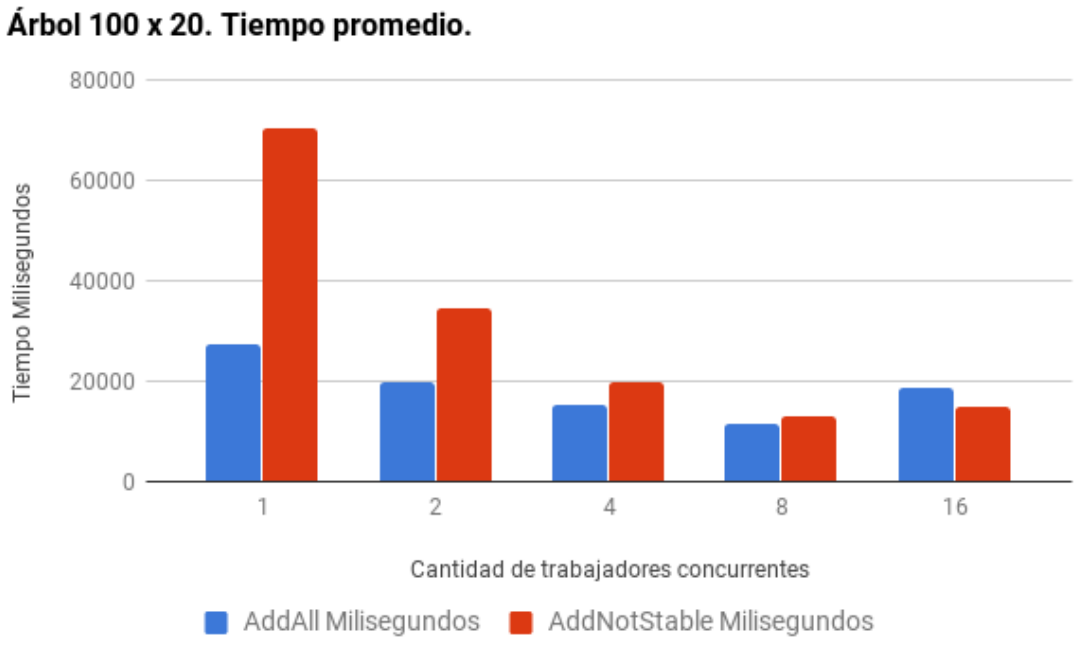

Fig. 6.1: Promedio de tiempo. Caso de estudio: Árbol 100 ramas de 20 nodos de longitud

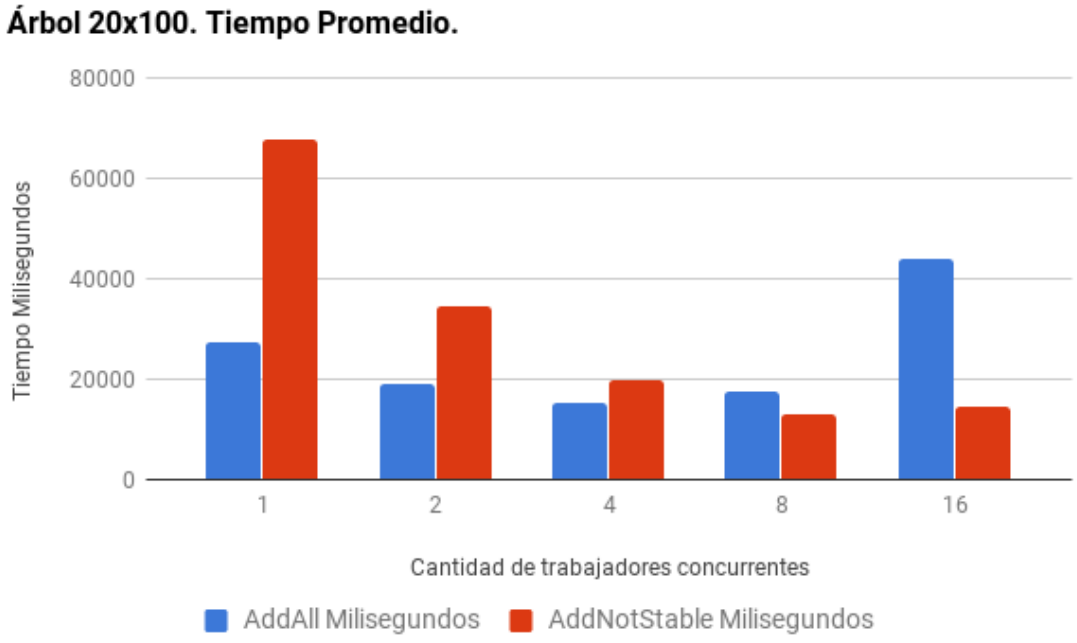

Fig. 6.2: Promedio de tiempo. Caso de estudio: Árbol 20 ramas de 100 nodos de longitud

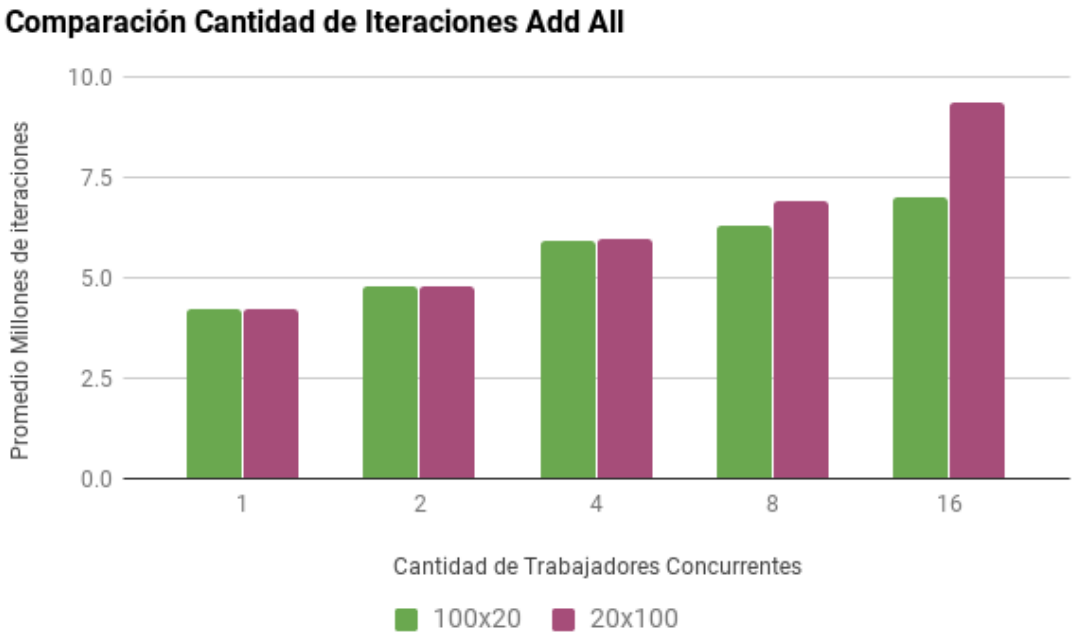

Fig. 6.3: Iteraciones promedio en Millones. Comparación de arboles usados agregando todos los predecesores a la cola

Para analizar y justificar porqué los tiempos del árbol de 20 ramas y 100 de longitud por rama empeoran se puede ver (en la figura 6.3) que la cantidad de iteraciones requerida para estabilizar aumenta cuando aumenta la cantidad de trabajadores concurrentes.

6.1.2. Clique

El caso de la clique es donde se observa que los tiempos siempre mejoran, sin importar la implementación de la cola usada.

Más aún se observa que el caso de agregar todos los elementos resulta más eficiente, a pesar de requerir más iteraciones de punto fijo para estabilizar todo el juego (ver figura 6.5). Notar que el caso de evaluar la estabilidad de cada elemento requiere conocer el valor de todos sus sucesores, luego cada uno de estas verificaciones es igual a la cantidad de nodos de la clique.

En el caso donde solo se agregan los elementos que aún no están estabilizados, basta con 1000 iteraciones para estabilizar todos los nodos. Que es simplemente terminar de recorrer todos los nodos inicialmente agregados a la cola hasta vaciarla.

6.1.3. Escalera

En éste caso se observa que no es recomendable utilizar la opción de agregar todos los predecesores a la cola sin evaluar previamente si ya se encuentra estabilizado. El tiempo crece hasta sobrepasar el tiempo de una corrida secuencial agregando solo los nodos estables.

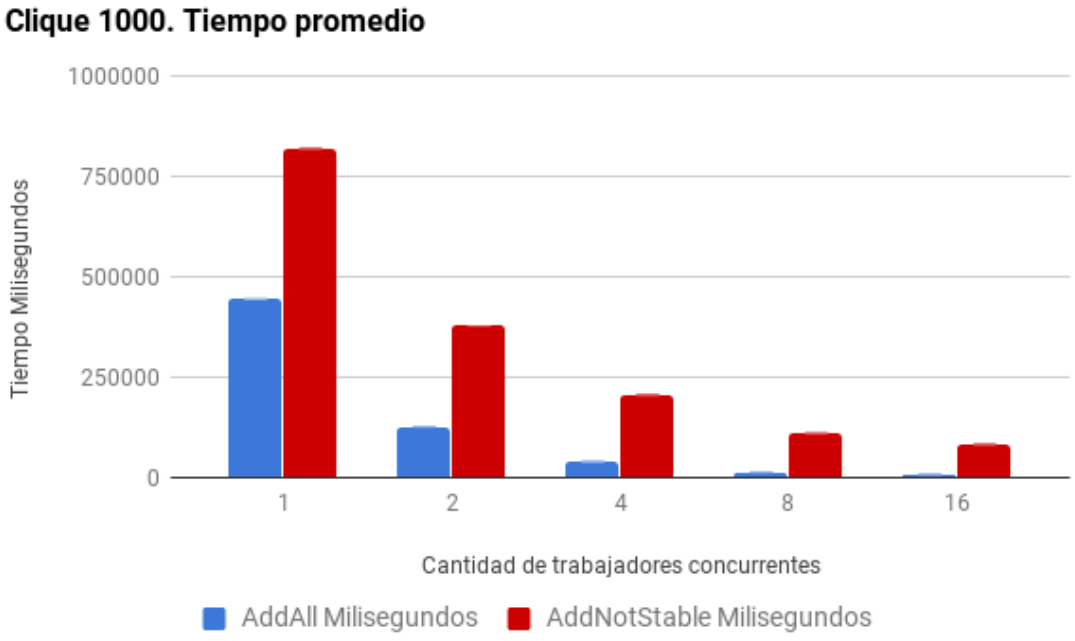

Fig. 6.4: Promedio de tiempo según cantidad de threads

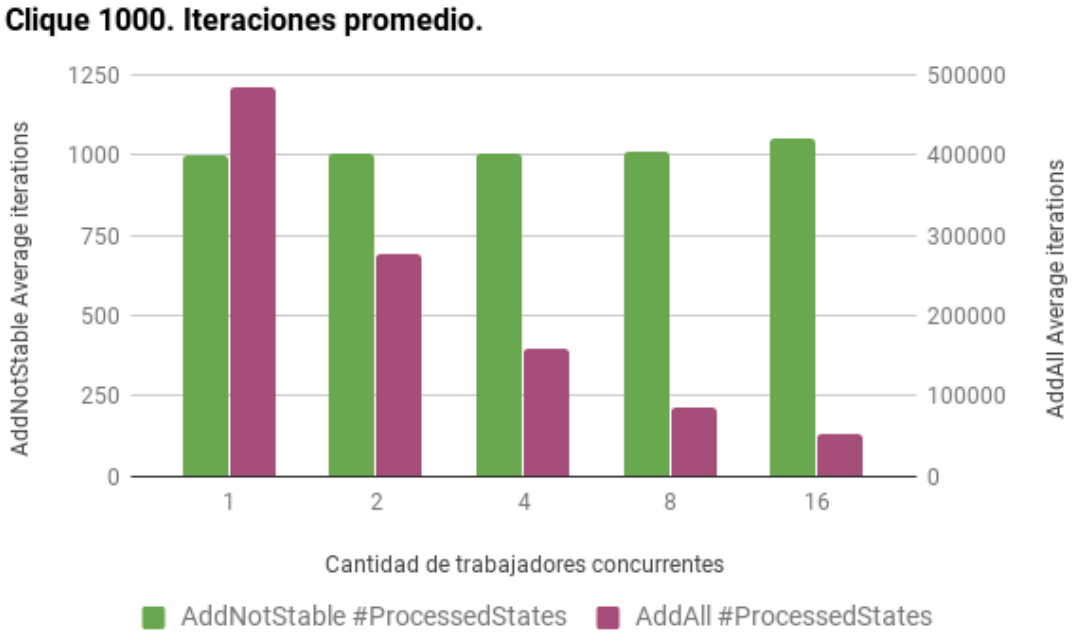

Fig. 6.5: Promedio de tiempo según cantidad de threads

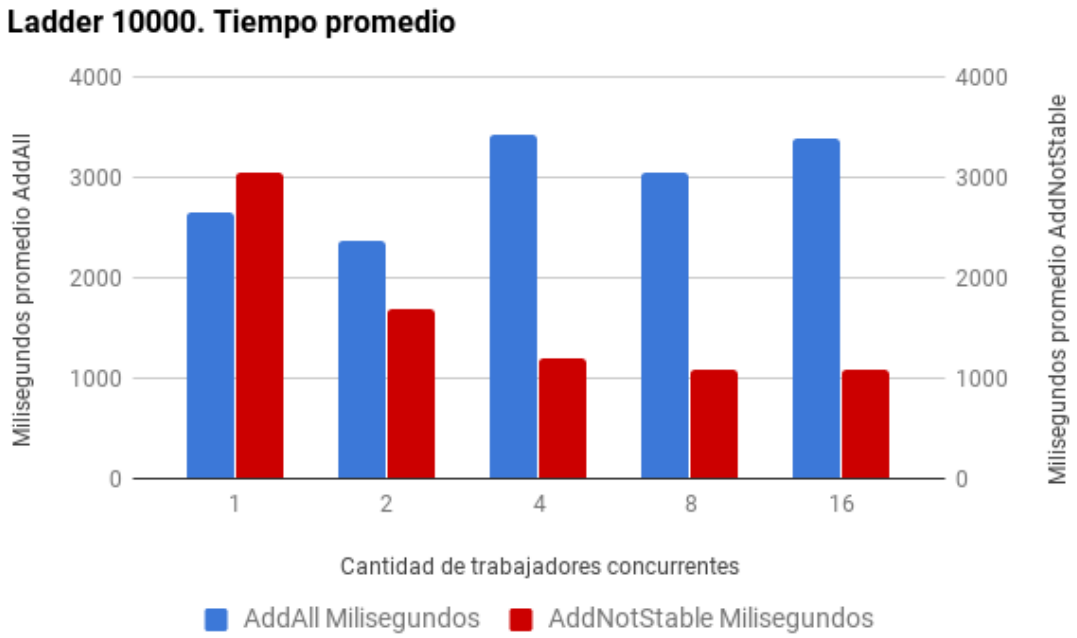

Fig. 6.6: Promedio de tiempo según cantidad de threads

Recordemos que el ejemplo de la escalera cada nodo tiene sólo dos sucesores y dos predecesores. Vuelve a impactar la hipótesis presentada para el árbol respecto al orden de evaluación de la cola.

6.1.4. Centro Médico y Ruteo

Estos casos fueron tomados de la literatura. Se observa que para un caso "real"donde la cantidad de sucesores y predecesores no se encuentra forzada, el algoritmo se comporta mejor agregando todos los predecesores sin evaluar la estabilidad previamente.

En el caso del ejemplo del caso clínico 6.7, se observa en el caso secuencial una diferencia de segundos versus minutos: 12 segundos el agregar todos, 2 minutos agregar verificando estabilidad.

En el caso de ruteo 6.8 se observa que los tiempos disminuyen también linealmente y que el caso de verificar la estabilidad previo a agregar los elementos a la cola supera al de agregar todos los predecesores.

6.2. Uso de CPU

A modo anecdótico se muestra a continuación un gráfico de uso de CPU al correr algunas pruebas. En el gráfico 6.9 se muestra el uso de CPU al correr las pruebas en una computadora con 8 cores. Los dos primeros picos son de 16 y 8 hilos respectivamente, y luego se ve como al correr usando solo 4 hilos el porcentaje de uso de CPU disminuye a la mitad. En el gráfico también se ve como al usar menos hilos la ejecución demora más.

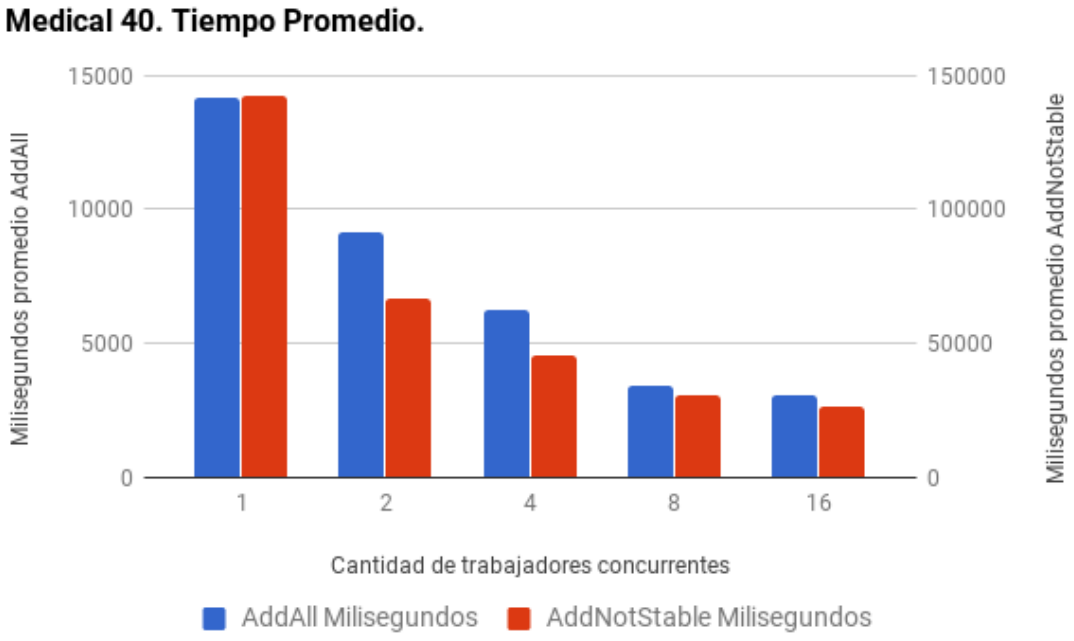

Fig. 6.7: Promedio de tiempo según cantidad de threads

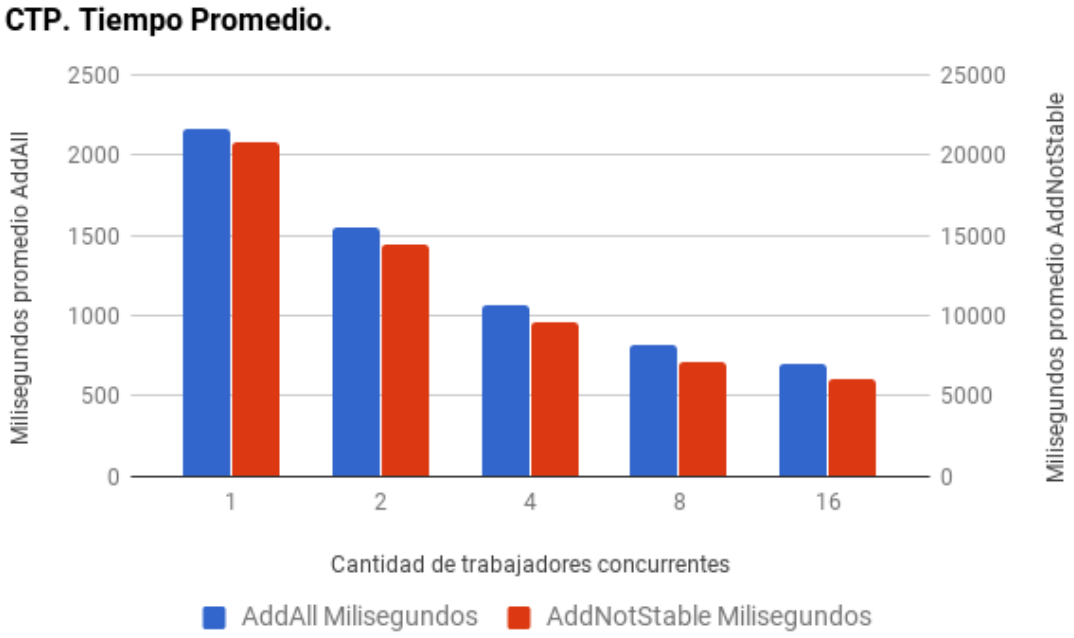

Fig. 6.8: Promedio de tiempo según cantidad de threads

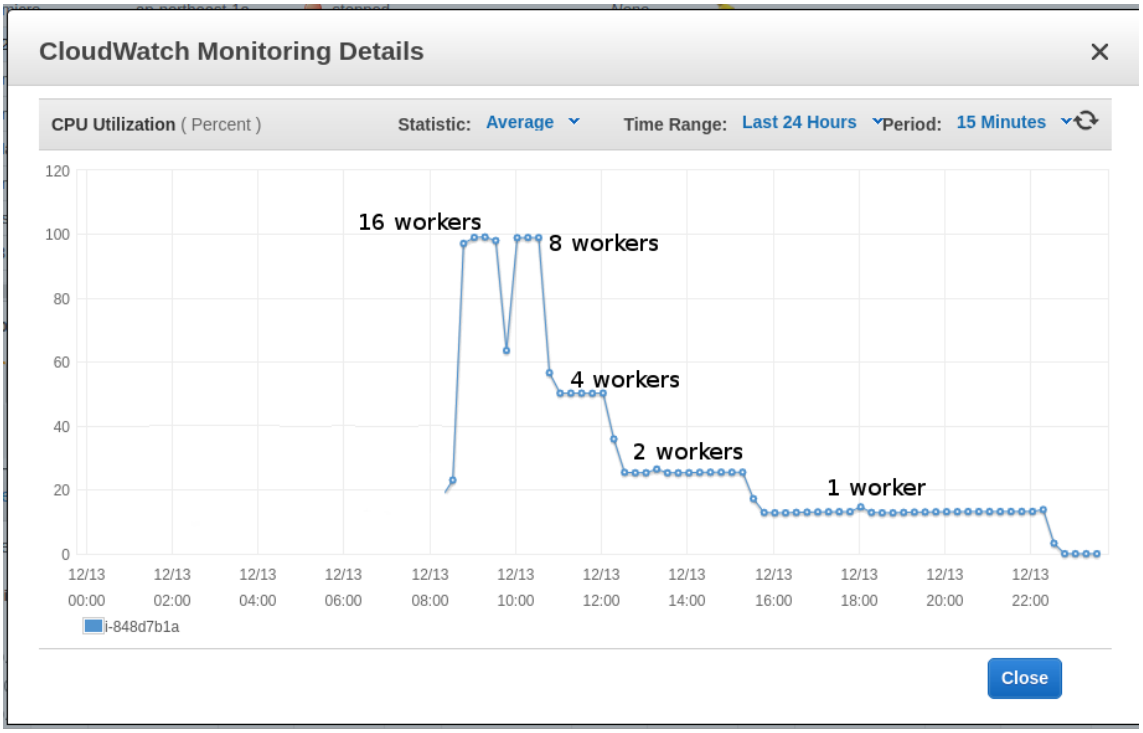

Fig. 6.9: Vista de utilización de CPU en el servidor con detalle de threads

El caso de 16 trabajadores concurrentes se usó para verificar que el nivel de paralelización de esa computadora era de 8 hilos.

6.3. Tiempos de procesamiento

Analicemos primero los casos morfológicos. Son los casos que se tomaron como referencia de estructuras conocidas para poder arrojar alguna hipótesis respecto a los tiempos y cantidad de iteraciones.

Como se puede ver en las figuras 6.7, 6.1, 6.2 y 6.4 las mejoras al agregar todos los estados mostraron una diferencia sustancial en los tiempos. En cambio, en el caso 6.6 se puede observar que de hecho el agregar todos los estados sin analizar la estabilidad termina afectando negativamente en todo el balance.

En los casos donde el juego tiene mayor número de conexiones y por lo tanto mayor computo por nodo para evaluar su posible nuevo valor, es donde el procesamiento paralelo saca mayor ventaja. Es el caso de la Clique donde los tiempos siempre decrementan linealmente.

Cuanto menor es el numero de conexiones de un nodo, menor será el tiempo que se dedique a analizar su mejor futuro valor de ranking, y si el nodo ya se encuentra o no estable. En el caso de los árboles con sólo 1 nodo sucesor (excepto en la raíz), se puede observar que si bien los tiempos siguen mejorando, no es lineal como en el caso de la clique. Incluso en los casos con demasiados hilos trabajando a la vez se ve que al contrario, los tiempos empeoran. La hipótesis de este caso es que hay un orden óptimo de análisis de los nodos. Si bien el algoritmo de punto fijo garantiza que va a terminar aún siendo

usado concurrentemente, vemos que el orden en el que se toman los elementos de la cola parece ser relevante. Más aún, suponemos que el máximo grado de paralelización estará determinado por la cantidad de ramas de dicho árbol.

Lo mismo se observa en el caso de la escalera donde hay sólo dos nodos sucesores por cada uno. Se ve que la cantidad de iteraciones de punto fijo se va incrementando al aumentar la cantidad de trabajadores concurrentes. Esto a priori debería empeorar los tiempos, sin embargo al realizar más trabajo a la par se contrarresta.

6.4. Iteraciones requeridas para alcanzar estabilidad

Lo que se observó al respecto es que la cantidad de iteraciones se duplicaba al duplicar la cantidad de trabajadores. El único caso donde esto disminuía, incluso inversamente, fue el de la clique.

Nuestra hipótesis al respecto es que en el caso de la clique no hace falta seguir un cierto orden de análisis de nodos ya que los cambios en uno van a ser vistos por todos los otros. En cambio en los casos con menor cantidad de conexiones ocurría que un nodo era tomado de la cola para su análisis y se devolvía sin cambio en su valor ya que su nodo anterior aún no había sido actualizado.

7. CONCLUSIONES

7.1. Análisis final

En éste trabajo se hizo la primera implementación del algoritmo concurrente para estabilización de juegos con objetivos GR(1). Fue necesario un profundo análisis del estado del arte y entendimiento del cómputo secuencial actual. Esto puede verse en las secciones dentro del capítulo 2.

En la sección 3.1 se hizo un análisis de acceso concurrente a datos. Esto sirvió para plantear una solución concurrente para nuestro algoritmo. Se pueden ver los pro y cons de implementar concurrencia sin bloqueo a datos compartidos, y sin secciones sincrónicas.

Los tiempos detallados en la sección 6 muestran que nuestra propuesta de algoritmo concurrente es una mejora al estado de arte en todos los casos analizados.

Para lograr una cobertura general de los tiempos de ejecución, se analizaron casos morfológicos. Dichos casos sirvieron de muestra que el algoritmo en casos extremos donde hay muchas conexiones o en casos donde hay pocas mejora los tiempos de la corrida secuencial.

Las excepciones de no monotonía decreciente fueron el árbol y la escalera. Ambos casos se pueden justificar por el poco nivel de paralelismo que tienen los grafos de su juego. Sin embargo ambos grafos se sugiere utilizar el algoritmo secuencial o 2 trabajadores concurrentes, agregando todos los predecesores a la cola sin analizar la estabilidad de dichos estados.

El caso de la clique y los casos de la literatura también se sugiere agregar todos los predecesores sin análisis de estabilidad. Sin embargo se observa que los tiempos siguen mejorando al agregar más trabajadores concurrentes. Es por esto que se deja sugerencia de trabajo a futuro para continuar la mejora de éste primer algoritmo, un análisis de la morfología del juego previo al computo de la estabilización.

7.2. Trabajo futuro

Al utilizar la técnica de *profiling* otro de los métodos más recurrentes y de mayor uso de CPU era la comparación de estados. Dado que el alcance de éste trabajo no era mejorar las implementaciones internas de la herramienta en cuestión dicha mejora quedó relegada.

Además se detectó que una parte paralelizable es la construcción del controlador a partir de los resultados del ranking obtenido. Al realizar ésta mejora se debería tener en cuenta la memoria utilizada y eliminar las llamadas recursivas.

Por otro lado, un área interesante para seguir investigando es el uso de distintos tipos de colas o conjuntos para organizar los estados del juego pendientes a estabilizar. Se expuso en la sección 4 que de acuerdo a la elección se mejoran los tiempos en determinados juegos.

Hubo otros casos no usados para éste análisis donde los juegos eran no determinísticos (MTS) y los tiempos empeoraban al agregar más trabajadores concurrentes. No fue objeto de este trabajo analizar éste tipo de juegos.

Además otra posible mejora podría ser: al momento de construcción de dicho juego tomar algunas referencias del mismo: como la cantidad de nodos adyacentes, cantidad de asunciones y garantías. No sería conveniente que cada usuario tenga que declarar la

cantidad de trabajadores (adivinando), lo ideal es que la herramienta realice ese análisis de forma automática y pueda sugerir un rango de posibles valores o el valor óptimo para la cantidad de trabajadores a usar.

En otros trabajos sobre paralelización de juegos de paridad se sugiere la división de los nodos para minimizar o eliminar la interacción de los procesos. En este trabajo se enfocó en no requerir del usuario datos para poder realizar la división de tareas en distintos hilos y creemos que obtuvimos buenos resultados. De todas maneras otra posible mejora sería trasladar la división de nodos intentar balancear las colas de trabajo a medida que se procesa el juego, de forma automática. Esto podría hacerse usando información de adyacencias de los nodos.

Por último como podemos observar en las tablas comparativas de resultados, los tiempos utilizando concurrencia mejoran los tiempos de cuando se utiliza el algoritmo tradicional secuencial. Resta analizar mejores usos de memoria para que pueda ser usado en controladores con mayor número de garantías y asunciones.

Apéndice

A. TABLAS DE RESULTADOS

A.1. Escalera 10.000 estados

Workers	Iteraciones AA	Milisegundos AA	Iteraciones ANS	Milisegundos ANS
1	201710	3383	112120	4262
1	201710	2671	112120	2950
1	201710	2721	112120	2975
1	201710	2546	112120	2920
1	201710	2541	112120	3105
1	201710	2693	112120	2826
1	201710	2621	112120	2864
1	201710	2494	112120	2835
1	201710	2532	112120	2827
1	201710	2500	112120	2927
1	201710	2513	112120	2987
2	249289	3034	132827	3031
2	244554	2176	124597	1724
2	244245	2195	123067	1512
2	245422	2105	129644	1692
2	276181	2471	123116	1472
2	246614	2181	122613	1531
2	275181	2336	123342	1476
2	273923	2396	123337	1630
2	274622	2394	123031	1490
2	270369	2440	123799	1503
2	272316	2341	123967	1504
4	355890	2597	169281	2371
4	361485	1919	158951	1157
4	356672	1715	163540	1058
4	334639	1640	165962	1109
4	342929	1609	165830	1057
4	358943	1662	162725	1041
4	390245	5155	163642	1080
4	391145	5353	168458	1061
4	385931	5261	165154	1035
4	384476	5215	163306	1052
4	393019	5499	163257	1069
8	471109	2316	230645	2320
8	536109	1527	254702	1193
8	519242	3295	251727	1008
8	523656	3169	264175	1003
8	545867	3285	252182	925
8	556034	3605	249326	848
8	500378	3301	246965	868
8	526311	3129	246862	1020
8	534346	3160	252810	913
8	529704	3258	256551	905
8	583778	3495	246729	855
16	579317	2544	262609	2614
16	762193	3395	314040	967
16	798002	3370	325179	958
16	892896	3724	368189	935
16	831725	3452	370662	884
16	798725	3291	361172	1002
16	848404	3602	371942	886
16	830741	3427	386936	899
16	865146	3593	371608	942
16	815566	3369	378079	879
16	817141	3439	385415	907

A.2. Clique 1000 estados

Workers	Iteraciones AA	Milisegundso AA	Iteraciones ANS	Milisegundos ANS
1	485624	457048	1001	864232
1	485624	452107	1001	845781
1	485624	451435	1001	840415
1	485624	424961	1001	838051
1	485624	422600	1001	807683
1	485624	424030	1001	784570
1	485624	441808	1001	807680
1	485624	453100	1001	838063
1	485624	449756	1001	784570
1	485624	450023	1001	799122
1	485624	454483	1001	795220
2	277300	131879	1003	391195
2	275939	124395	1001	392370
2	278081	126632	1002	374495
2	275898	124117	1002	375498
2	278963	126361	1001	375408
2	276965	124210	1002	370290
2	276722	124564	1001	371707
2	283033	126660	1002	370722
2	274681	121927	1002	372778
2	279825	124137	1002	372684
2	276889	123947	1004	372913
4	156984	44830	1005	206366
4	159099	38964	1009	202943
4	154129	37379	1006	203353
4	158568	38197	1002	203872
4	158168	38375	1003	203874
4	161323	38504	1004	204923
4	159362	37885	1004	204227
4	159811	38045	1004	204683
4	157434	37769	1006	204049
4	157999	37792	1012	204143
4	156873	37537	1004	203994
8	88131	20924	1035	115333
8	82744	11617	1010	108982
8	85400	11629	1002	108183
8	82425	10806	1005	108811
8	87348	11486	1008	108417
8	84919	10958	1010	108427
8	86950	11121	1007	108305
8	84600	10761	1006	107974
8	84835	10771	1011	108338
8	85861	10761	1009	108240
8	86079	10818	1007	108425
16	61282	16007	1033	94543
16	53938	6647	1067	80317
16	50514	5858	1097	80603
16	51708	5960	1042	78761
16	48876	5442	1101	80515
16	50302	5558	1064	79827
16	50349	5509	1033	79206
16	49232	5299	1059	79496
16	52037	5732	1025	78546
16	50086	5286	1026	78687
16	51316	5455	1025	78624

A.3. Árbol 100 ramas x 20 longitud

Workers	Iteraciones AA	Milisegundso AA	Iteraciones ANS	Milisegundos ANS
1	4228687	30204	3012079	68613
1	4228687	26741	3012079	70405
1	4228687	26600	3012079	70577
1	4228687	26998	3012079	70446
1	4228687	27729	3012079	70676
1	4228687	26742	3012079	70363
1	4228687	26616	3012079	70637
1	4228687	26710	3012079	70247
1	4228687	26801	3012079	70488
1	4228687	26737	3012079	70551
1	4228687	26858	3012079	70712
2	4785456	20370	3051339	34905
2	4809399	19464	3053122	33624
2	4802889	19498	3056294	34754
2	4810187	19782	3053072	33102
2	4802614	19595	3056251	34256
2	4788928	20156	3055356	34716
2	4789236	19551	3054924	34395
2	4770959	19542	3052838	33598
2	4811299	20068	3055016	34000
2	4801091	19929	3054735	34871
2	4804098	20246	3054246	34550
4	5832110	16196	3199849	22142
4	5882650	14382	3194675	19559
4	5906274	14569	3193362	19709
4	5867410	14586	3192523	19383
4	5940927	15342	3196085	19441
4	5948893	15297	3190375	19508
4	5939620	15441	3194050	19710
4	5972548	15611	3193385	19513
4	5860325	14223	3195910	19643
4	5922457	15011	3193820	19605
4	5875364	14729	3197330	19622
8	6345601	13520	3456485	14808
8	6312666	11590	3488061	12065
8	6264916	11042	3503268	12369
8	6318977	11433	3498020	12990
8	6292025	10651	3499662	12808
8	6207787	11433	3490369	12562
8	6092527	10738	3478503	12632
8	6400823	11593	3494181	12759
8	6161109	11144	3489217	12636
8	6313577	11395	3516729	13028
8	6308774	11219	3492694	12468
16	6777642	17896	4053271	16222
16	6834556	17668	4204584	14392
16	6984227	18166	4226268	14660
16	6918946	17724	4253554	14941
16	7093277	18934	4205904	14546
16	7072755	18964	4176657	14779
16	7180517	19974	4335402	15783
16	6909247	18476	4244216	14086
16	6994499	18955	4176741	14743
16	6957068	18551	4275788	15166
16	6984702	19113	4243610	15141

A.4. Árbol 20 ramas x 100 longitud

Workers	Iteraciones AA	Milisegundso AA	Iteraciones ANS	Milisegundos ANS
1	4228687	27590	3012079	67669
1	4228687	25888	3012079	66511
1	4228687	26133	3012079	70818
1	4228687	26518	3012079	71244
1	4228687	27433	3012079	70845
1	4228687	28102	3012079	71044
1	4228687	28449	3012079	70928
1	4228687	28364	3012079	66049
1	4228687	28430	3012079	63527
1	4228687	26521	3012079	63621
1	4228687	25808	3012079	63681
2	4777501	18764	3055821	36396
2	4746620	18207	3050049	33511
2	4730908	19204	3053988	35380
2	4755817	18666	3053441	33915
2	4753395	18874	3053114	34562
2	4821078	18801	3054845	34271
2	4806318	18696	3054432	34169
2	4786474	18906	3058994	34689
2	4759517	18709	3056011	34195
2	4800100	18923	3056537	35060
2	4807241	19211	3054765	34263
4	5915220	15462	3187923	21328
4	5976084	15453	3188305	19257
4	5932246	14905	3192998	19162
4	5919140	14967	3183944	19604
4	5944413	15558	3190481	19397
4	5973542	15290	3186545	19163
4	5917308	14928	3188351	19377
4	5876788	14632	3184370	19371
4	5905100	15180	3190738	19562
4	6008679	15913	3187506	19082
4	5948056	15354	3197347	19347
8	6789318	14940	3489167	14129
8	6908357	16687	3480005	12437
8	6816867	16833	3504079	12429
8	7143466	19241	3502684	12552
8	6842703	18955	3512134	12422
8	6891564	16650	3483480	12820
8	6893671	16779	3515163	12574
8	6801666	17016	3500574	12728
8	7121658	19019	3500970	12544
8	7112585	19436	3516465	12517
8	6765333	16905	3503971	12769
16	8644292	34615	4133649	16480
16	9569577	44272	4178582	13826
16	9603703	46130	4233739	14234
16	9410646	45469	4151203	14265
16	9315609	45629	4197396	14243
16	9538656	46304	4219448	14119
16	9507388	45679	4202710	14023
16	9359317	44480	4200443	14063
16	9224042	42831	4266002	14287
16	9281045	42621	4315978	15623
16	9311586	43645	4197830	14108

A.5. Medical 40 enfermedades

Workers	Iteraciones AA	Milisegundso AA	Iteraciones ANS	Milisegundos ANS
1	340630	14763	205746	13673
1	340630	14162	205746	13086
1	340630	14106	205746	12923
1	340630	14102	205746	12898
1	340630	14124	205746	12900
1	340630	14103	205746	12878
1	340630	14115	205746	12890
1	340630	14110	205746	12872
1	340630	14124	205746	12880
1	340630	14146	205746	12889
1	340630	14133	205746	12887
2	374842	10628	224414	6953
2	389191	9145	224215	6061
2	389579	9042	225124	6108
2	389286	8988	224730	5978
2	389719	8974	225107	5954
2	389706	8999	225033	5906
2	389902	9010	224868	5897
2	389064	8955	224843	5890
2	388214	8918	224747	5924
2	388232	8880	225116	5930
2	389471	8936	224689	5882
4	445575	6615	249965	4717
4	437837	5975	248494	4220
4	430385	6284	246539	4174
4	431268	6288	246819	4107
4	429930	6274	246243	4144
4	430692	6267	245723	4067
4	430403	6227	246332	4055
4	430725	6240	246731	4064
4	430051	6231	246657	4014
4	430466	6239	247115	4023
4	431003	6238	246895	4012
8	585903	4346	327843	3618
8	538114	3687	335969	2712
8	598512	3339	327827	2794
8	591875	3469	327009	2729
8	596708	3343	329637	2700
8	600212	3272	327766	2695
8	598627	3262	326361	2685
8	599581	3247	327828	2656
8	598678	3247	326083	2643
8	592800	3368	329605	2683
8	599699	3280	327491	2665
16	692843	4100	359261	3633
16	720443	3053	427543	2379
16	805739	3031	415307	2336
16	803374	2946	414506	2281
16	802531	2935	407161	2218
16	809310	2905	408455	2204
16	801634	2866	407058	2187
16	804998	2902	408672	2209
16	796360	2867	412967	2190
16	802494	2895	412856	2186
16	805412	2883	412428	2188

A.6. Transporte

Workers	Iteraciones AA	Milisegundso AA	Iteraciones ANS	Milisegundos ANS
1	105705	2567	58485	2362
1	105705	2056	58485	1927
1	105705	2106	58485	1941
1	105705	2097	58485	1775
1	105705	2200	58485	1849
1	105705	2072	58485	1932
1	105705	2116	58485	1888
1	105705	2081	58485	1774
1	105705	2149	58485	1767
1	105705	2189	58485	1809
1	105705	2090	58485	1766
2	129056	2171	67013	2033
2	125889	1508	65352	1310
2	126521	1551	65345	1256
2	125188	1552	65345	1236
2	125960	1445	65513	1211
2	125535	1515	64911	1265
2	125428	1453	65379	1193
2	124909	1488	65383	1200
2	125650	1491	65354	1203
2	125333	1444	65192	1210
2	125965	1456	64964	1251
4	161175	1654	79444	1532
4	159279	1119	78644	893
4	165719	1009	79091	843
4	164072	1068	80080	906
4	163956	1025	78088	771
4	164768	975	79182	791
4	162884	1015	79090	764
4	163763	956	78995	754
4	164322	968	78660	747
4	163746	978	78466	791
4	166339	956	79184	744
8	200366	1469	99619	1364
8	208824	914	98340	600
8	204428	792	100994	745
8	217674	785	102864	547
8	218304	712	101522	571
8	217489	741	103269	515
8	213192	709	102687	558
8	217819	709	102547	512
8	218869	697	101777	678
8	207957	725	103599	529
8	200738	772	102451	513
16	228376	1315	122264	1416
16	246250	770	120651	593
16	267763	669	134102	535
16	262650	695	129972	485
16	262246	607	130834	417
16	257001	593	129361	451
16	267051	633	129406	409
16	261760	587	129317	450
16	254145	601	130093	407
16	264354	595	129234	435
16	255231	599	132893	405

Bibliografía

[BVW13] Harald Bauer, Jan Veira, and Florian Weig. Moore's law: Repeal or renewal?, december 2013.

[DFCU08a] N. D'Ippolito, D. Fischbein, M. Chechik, and S. Uchitel. Mtsa: The modal transition system analyser. In *Automated Software Engineering, 2008. ASE 2008. 23rd IEEE/ACM International Conference on*, pages 475–476, Sept 2008.

[DFCU08b] Nicolas D'Ippolito, Dario Fischbein, Marsha Chechik, and Sebastian Uchitel. Mtsa: The modal transition system analyser. In *23rd IEEE ACM International Conference on Automated Software Engineering*, pages 475–476. IEEE Computer Society, 2008.

[DIBPU11] Nicolás D' Ippolito, Victor Braberman, Nir Piterman, and Sebastián Uchitel. Synthesis of live behaviour models for fallible domains. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 211–220, New York, NY, USA, 2011. ACM.

[DIBPU13] Nicolás D' Ippolito, Victor Braberman, Nir Piterman, and Sebastián Uchitel. Synthesizing nonanomalous event-based controllers for liveness goals. *ACM Trans. Softw. Eng. Methodol.*, 22(1):9:1–9:36, March 2013.

[DL15] Distributed Software Engineering (DSE) group at Imperial College London and Laboratory on Foundations and Tools for Software Engineering (LaFHIS) at University of Buenos Aires. Modal transition analizer. http://sourceforge.net/projects/mtsa/, 2015.

[EWS05] K. Etessami, T. Wilke, and R. Schuller. Fair simulation relations, parity games, and state space reduction for büchi automata. *SIAM Journal on Computing*, 34(5):1159–1175, 2005.

[GM03] Dimitra Giannakopoulou and Jeff Magee. Fluent model checking for event-based systems. In *Proceedings of the 9th European Software Engineering Conference Held Jointly with 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-11, pages 257–266, New York, NY, USA, 2003. ACM.

[GTW02] Erich Grädel, Wolfgang Thomas, and Thomas Wilke, editors. *Automata Logics, and Infinite Games: A Guide to Current Research*. Springer-Verlag New York, Inc., New York, NY, USA, 2002.

[GWT02] Erich Grädel, Thomas Wilke, and Wolfgang Thomas. *Automata, Logics, and Infinite Games: A Guide to Current Research*. Springer, 2002.

[HKP12] Michael Huth, Jim Huan-Pu Kuo, and Nir Piterman. Concurrent small progress measures. In *Proceedings of the 7th International Haifa Verification Conference on Hardware and Software: Verification and Testing*, HVC'11, pages 130–144, Berlin, Heidelberg, 2012. Springer-Verlag.

[HPK11] Michael Huth, Nir Piterman, and Jim Huan-Pu Kuo. Concurrent small progress measures. 7261:130–144, 2011.

[Jac95a] Michael Jackson. *Software Requirements &Amp; Specifications: A Lexicon of Practice, Principles and Prejudices.* ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1995.

[Jac95b] Michael Jackson. The world and the machine. In *Software Engineering, 1995. ICSE 1995. 17th International Conference on*, pages 283–283, April 1995.

[JP06a] Sudeep Juvekar and Nir Piterman. Minimizing generalized büchi automata. In Thomas Ball and RobertB. Jones, editors, *Computer Aided Verification*, volume 4144 of *Lecture Notes in Computer Science*, pages 45–58. Springer Berlin Heidelberg, 2006.

[JP06b] Sudeep Juvekar and Nir Piterman. Minimizing generalized büchi automata. In Thomas Ball and RobertB. Jones, editors, *Computer Aided Verification*, volume 4144 of *Lecture Notes in Computer Science*, pages 45–58. Springer Berlin Heidelberg, 2006.

[Jur98] Marcin Jurdziński. Deciding the winner in parity games is in up $\bigcap$ co-up. *Inf. Process. Lett.*, 68(3):119–124, November 1998.

[Jur00] Marcin Jurdziński. Small progress measures for solving parity games. In Horst Reichel and Sophie Tison, editors, *STACS 2000*, volume 1770 of *Lecture Notes in Computer Science*, pages 290–301. Springer Berlin Heidelberg, 2000.

[Kel76] Robert M. Keller. Formal verification of parallel programs. *Commun. ACM*, 19(7):371–384, July 1976.

[KGFP07] Hadas Kress-Gazit, Georgios E. Fainekos, and George J. Pappas. *Where's Waldo? Sensor-based temporal logic motion planning*, pages 3116–3121. 2007.

[KPP05] Yonit Kesten, Nir Piterman, and Amir Pnueli. Bridging the gap between fair simulation and trace inclusion. *Inf. Comput.*, 200(1):35–61, July 2005.

[KPR04] R. Kazhamiakin, M. Pistore, and M. Roveri. Formal verification of requirements using spin: a case study on web services. In *Software Engineering and Formal Methods, 2004. SEFM 2004. Proceedings of the Second International Conference on*, pages 406–415, Sept 2004.

[LvL02] Emmanuel Letier and Axel van Lamsweerde. Agent-based tactics for goal-oriented requirements elaboration. In *Proceedings of the 24th International Conference on Software Engineering*, ICSE '02, pages 83–93, New York, NY, USA, 2002. ACM.

[MK99] Jeff Magee and Jeff Kramer. *Concurrency: State Models & Java Programs.* John Wiley & Sons, Inc., New York, NY, USA, 1999.

[MKG97] J. Magee, J. Kramer, and D. Giannakopoulou. Analysing the behaviour of distributed software architectures: a case study. In *Distributed Computing Systems, 1997., Proceedings of the Sixth IEEE Computer Society Workshop on Future Trends of*, pages 240–245, Oct 1997.

[MPS95] Oded Maler, Amir Pnueli, and Joseph Sifakis. On the synthesis of discrete controllers for timed systems (an extended abstract). In *STACS*, pages 229–242, 1995.

[OF] Martin Lange Oliver Friedmann. *The pgsolver Collection of Parity Game Solvers*. Institut für Informatik.

[PM95] David Lorge Parnas and Jan Madey. Functional documents for computer systems. *Science of Computer Programming*, 25:41–61, 1995.

[PPS06a] Nir Piterman, Amir Pnueli, and Yaniv Sa'ar. Synthesis of reactive(1) designs. 01 2006.

[PPS06b] Nir Piterman, Amir Pnueli, and Yaniv Sa'ar. Synthesis of reactive(1) designs. volume 3855 of *Lecture Notes in Computer Science*, pages 364–380. Springer Berlin Heidelberg, 2006.

[PR89] A. Pnueli and R. Rosner. On the synthesis of a reactive module. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '89, pages 179–190, New York, NY, USA, 1989. ACM.

[RW89] P.J.G. Ramadge and W.M. Wonham. The control of discrete event systems. *Proceedings of the IEEE*, 77(1):81–98, Jan 1989.

[vdPW08a] Jaco van de Pol and Michael Weber. A multi-core solver for parity games. *Electr. Notes Theor. Comput. Sci.*, 220(2):19–34, 2008.

[vdPW08b] Jaco van de Pol and Michael Weber. A multi-core solver for parity games. *Electron. Notes Theor. Comput. Sci.*, 220(2):19–34, December 2008.

[VL01] Axel Van Lamsweerde. Goal-oriented requirements engineering: A guided tour. In *Proceedings of the Fifth IEEE International Symposium on Requirements Engineering*, RE '01, pages 249–, Washington, DC, USA, 2001. IEEE Computer Society.

[vL09] A. van Lamsweerde. *Requirements Engineering: From System Goals to UML Models to Software Specifications*. Wiley, 2009.

[VLL00] A Van Lamsweerde and E. Letier. Handling obstacles in goal-oriented requirements engineering. *Software Engineering, IEEE Transactions on*, 26(10):978–1005, Oct 2000.

[ZJ97] Pamela Zave and Michael Jackson. Four dark corners of requirements engineering. *ACM Transactions on Software Engineering and Methodology*, 6:1–30, 1997.