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

Implementación y estudio de un algoritmo para la comprobación de *General Multiparty Compatibility* entre *Communicating Finite State Machines* con datos

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

Axel J. Iglesias

Director: Ignacio Vissani
Codirector: Carlos Gustavo Lopez Pombo
Buenos Aires, 2016

IMPLEMENTACIÓN Y ESTUDIO DE UN ALGORITMO PARA LA COMPROBACIÓN DE GENERAL MULTIPARTY COMPATIBILITY ENTRE COMMUNICATING FINITE STATE MACHINES CON DATOS

En el paradigma de *Service Oriented Computing* la estructura de los sistemas es intrínsecamente dinámica. Las piezas de software ejecutan sobre infraestructuras de hardware globalmente distribuidas y utilizan servicios externos para alcanzar sus objetivos de negocio. La búsqueda (*discovery*) y la conexión (*binding*) con estos servicios externos debe llevarse a cabo en tiempo de ejecución. Esta tarea es llevada a cabo por un *middleware* dedicado.
Uno de los aspectos para poder determinar si un servicio es capaz de satisfacer los requerimientos del consumidor es el de interoperabilidad.
En este plano las *CFSMs* y las coreografías aparecen como formalismos adecuados para especificar los aspectos interoperacionales esperados/provistos por un servicio sobre un canal particular.
Si bien las *CFSMs* proveen un mecanismo adecuado para realizar el chequeo de interoperabilidad, hay otros aspectos funcionales que no son capturados en este formalismo. Para superar esta limitación estamos trabajando en una extensión a las *CFSMs* que contemple a los datos intercambiados y que permita expresar condiciones sobre los mismos. Estas condiciones deben ahora tenerse en cuenta a la hora de determinar si un conjunto de participantes son compatibles. Para ello está en desarrollo un algoritmo capaz de llevar a cabo este chequeo.
Un paso crucial del chequeo es la construcción del sistema sincrónico. Un LTS que representa el comportamiento ideal esperado de la ejecución paralela de los distintos participantes de una comunicación. El algoritmo de síntesis del sistema sincrónico depende del criterio de equivalencia entre estados dado. En este trabajo damos un criterio de equivalencia que, bajo ciertas condiciones, es correcto y hace que que el algoritmo de síntesis converja. Asimismo diseñamos e implementamos dicho algoritmo con el objetivo de sentar las bases para continuar el desarrollo e investigación futura.
Finalmente se analizaron casos de estudio y rendimiento de la implementación con resultados satisfactorios.

Palabras claves: Software Orientado a Servicios, Comunicación Asincrónica, Communicating Finite State Machines, Discovery, Contratos.

IMPLEMENTATION AND STUDY OF AN ALGORITHM FOR GENERAL MULTIPARTY COMPATIBILITY CHECK BETWEEN GUARDED COMMUNICATING FINITE STATE MACHINES

In the Service Oriented Computing paradigm systems have an intrinsically dynamic structure. Software runs over globally distributed hardware and uses external services to achieve business goals. Service discovery and binding have to be done at runtime. This task is performed by a dedicated middleware.
Interoperability is one aspect needed to be able to determine if a service consumer's requirements can be satisfied.
CFSMs and choreographies are appropriate formalisms to specify service channel's expected interoperable behaviour. However, other functional behaviours are not captured by these tools. To overcome this limitation, we propose a CFSM extension that take into account exchanged data and conditions over this data. A new algorithm that considers this new formalism to check the interoperability is under development.
One key step in this algorithm is the synthesis of the synchronous transition system that models the ideal synchronous behaviour of the communicating partners. This algorithm depends upon a criterion to stablish the equivalence between states. In this work we give an equivalence criterion that, under certain restrictions, is correct and makes the algorithm convergent. We alos design and implement that algorithm with the goal of setting a base for further research and development. Finally, we create study cases to analyze the behaviour and performance with satisfactory results.

Keywords: Service Oriented Software, Asynchronous Communication, Communicating Finite State Machines, Discovery, Contracts.

AGRADECIMIENTOS

Dado que para mi la presentación de la tesis no es simplemente la culminación de una carrera de grado sino también el evento que refleja el cierre de una etapa, aprovecho la ocasión para agradecer también a aquellas personas que en este período han tenido un rol importante para mi.

Comienzo por agradecer al jurado y a mis directores: Charlie y Nacho, no solo por abrirme la oficina a hacer esta tesis y poder recibirme sino también por las numerosas horas compartidas en el transcurso de la carrera, los empujones previos a los finales que dudaba en presentarme y las discusiones que tuvimos en momentos donde los proyectos colectivos necesitan que algunos maduremos y nos hagamos cargo.

A la sociedad en general y al Estado Nacional en particular, entendido como la administración de lo común, por brindarme muchas herramientas (aunque no todas) para poder estudiar una carrera universitaria y recibirme.

A los docentes del DC que le ponen infinitas ganas y buena onda para hacer de la cursada algo que se disfruta.

A mis compañeros de cursada, desde el CBC a las últimas materias. Principalmente aquellos con quienes compartimos grupos de trabajo y más de una materia. En particular a Agus, Laski y Partu por acompañarme en el último tramo de la cursada. Los 4 sabemos lo fundamentales que fueron en ese momento. Quedará en el recuerdo nuestro viaje en el tiempo y la épica anécdota de cómo vimos el 7-1 de Alemania a Brasil al mismo tiempo que rendíamos un parcial.

A Pau, no solo por haberme salvado algunas materias sino también porque me guardo muy lindos recuerdos suyos.

A mis docentes del secundario: Juanjo, Nati, Mati y Luk por sembrarme las preguntas indicadas, empezar a abrirme la cabeza y hacerme notar lo valiosa y transformadora que puede ser la docencia.

A Nati, Luk, Mati, Ani, Juli y Mery, por haber sido mis primerxs compañerxs de lucha, quienes me enseñaron que si mi camino no tiene corazón, no vale la pena.

A Nati, por decir las palabras necesarias en el momento adecuado para que yo hoy me esté recibiendo y no haya abandonado la carrera.

A Anita, por empujarme a dar los primeros pasos cuando entré a la facultad y sorprenderse de todo lo que fui logrando. Porque nos conocemos mucho y no importa el tiempo ni la distancia, sabemos que cada uno tiene un lugar especial en el corazón del otro.

A Guta, quien a pesar de los años, las idas y vueltas, los cuelgues, las desapariciones y tantas otras cosas que nos identifican e identifican nuestra relación, me acompañó todo este tiempo.

A Patria Grande, La Mella y al FEM! en particular, en todas sus generaciones, con quienes aprendí que a la facultad no había que venir sólo a estudiar, que los proyectos colectivos son los que mayor poder de transformación tienen y que el esfuerzo, la dedicación y el compromiso valen la pena. Como dijo Chavez: se puede cambiar el curso de la historia.

A todxs mis compañerxs de militancia en estos años, en especial aquellxs que hoy lxs considero mis amigxs.

A Nacho y Ali quienes nunca fueron indiferentes y por ser quienes me instaron a sentir como propio este proyecto.

A Agus y Lupi, por hacerme notar en cada charla que tenemos que todo ese tiempo, esa cabeza y ese compromiso entregado a la consolidación de un grupo, no fue mal realizado ni tiempo perdido. Hoy son 2 referentes que admiro y me enorgullese haber sido parte, aunque sea un poquito, de su crecimiento.
A Eze, Santi, Fiore, Cani y Florci quienes estoy seguro que entienden mejor que nadie por qué están en este parrafito. Por los días y noches compartidas. Por la inocencia del comienzo y el compromiso construido. Porque con ustedes aprendí que si nos corre en la sangre, hay que militar.
A Chizo, porque no se bien cuando dejó de ser una compañera y pasó a ser una gran amiga. Por las rancheadas, los bailes en administración, los viernes en el sum, los viajes a las Bahamas y todos los hermosos recuerdos que me regaló estos años.
A Manu por haberme acompañado prácticamente en primera fila todos estos años. Por las horas de charla acumuladas y las de diversión disfrutadas. Las palmadas en la espalda, las arengas y los baldazos de agua fría.
A mi vieja, por apoyarme siempre en lo que hago, le guste o no le guste, mientras yo sea feliz.
A mi viejo, de quien aprendí la importancia de estudiar, de formarme, de leer, de aprender. Por haber sido pieza fundamental de que yo haya seguido esta carrera y esté recibiéndome. Porque se bancó discusiones de todo tipo, reuniones eternas en la casa y por estar siempre dispuesto a charlar.
A mis hermanxs, Flor y Lau, por reaparecer en mi vida los últimos años, dándonos cuenta que fuimos tocados por la misma varita. Gracias Flor por no abandonar la pelea e instar a estar juntxs y a Lau por haber vuelto a compartir horas y horas de diversión asegurada.
A Flor por haber aparecido en el último tramo y haberme invitado a abrirme. Por bancarse mi alarmismo, mis pocas ganas de estudiar y todas mis locuras. Porque me encanta crecer con ella. Porque la elijo como compañera de vida hace 3 años y la elegiría muchos años más. Por su alegría inacabable, sus palabras sinceras y su amor constante.

Índice general

1. INTRODUCCIÓN

El software distribuido se está volviendo cada vez más dinámico de modo de soportar aplicaciones que sean capaces de responder y adaptarse a los cambios que se producen en su entorno de ejecución. Como ejemplo de esto, el paradigma de Computación Orientada a Servicios (*SOC* por sus siglas en inglés) ve a las aplicaciones como servicios que ejecutan sobre recursos computacionales que están disponibles globalmente.

En una arquitectura orientada a servicios (*SOA* por sus siglas en inglés) existen tres roles: el *broker* de servicios recibe las publicaciones de los *proveedores de servicios* y administra el repositorio de servicios conocidos. En tiempo de ejecución los *clientes* buscan (*discovery*) servicios a los cuales vincularse (*binding*), y los usan de modo de alcanzar los objetivos de negocio dados [2]. En un paradigma como el descrito, ni los arquitectos de software ni las programadoras tienen control sobre la naturaleza de los componentes con los que una aplicación puede vincularse debido a que el proceso de *discovery* y *binding* es llevado a cabo de manera transparente por una capa de software especializado, el *middleware*, que implementa el rol de *broker*. Esta situación se esquematiza en la Fig. 1.1.

Los dos principios de diseño básicos para el desarrollo de software distribuido son *coreografías* y *orquestación* (ver [12]). La orquestación representa la visión desde el punto de vista de uno de los participantes. En cambio, las coreografías modelan la interacción entre los participantes de una comunicación desde un punto de vista global [10]. En un modelo coreografiado, los componentes computacionales distribuidos se representan como vistas parciales de la comunicación global multi-agente. El comportamiento global se obtiene al ejecutar los procesos participantes en paralelo. En el contexto de *SOC*, los procesos se diseñan de manera distribuida, y la orquestación y las coreografías actúan como principios complementarios aunque con algún solapamiento.

Desde nuestro punto de vista, las coreografías representan una visión más natural para modelar los aspectos comunicacionales de los servicios. La mayor dificultad inherente al modelo de coreografías es poder determinar si un conjunto de participantes deseosos de entablar una comunicación, van a tener éxito a la hora de ejecutar en paralelo. En otras palabras, si el conjunto de participantes es capaz de interoperar o no. Las *Communicating Finite State Machines* [3] (CFSM) son un formalismo basado en autómatas que sirve para describir puntos de vista parciales de una coreografía. Éstas describen el rol de un participante dado, desde el punto de vista del intercambio de mensajes, en una comunicación

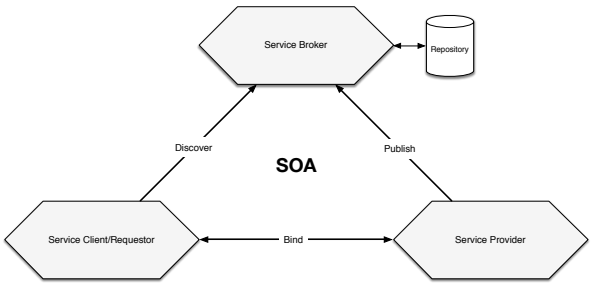

Fig. 1.1: El triángulo de SOA

multi-agente. En el caso de las *CFSMs*, el chequeo de interoperabilidad fue resuelto en [11] por medio de un algoritmo que permite sintetizar coreografías partiendo de las *CFSMs*. Si dicho algoritmo de síntesis logra producir una coreografía entonces la interoperabilidad entre los participantes queda garantizada por la propiedad llamada *general multiparty compatibility* (GMC) de modo que la composición está libre de errores de comunicación.

Como ya se mencionó, las *CFSMs* describen el rol de un participante en una comunicación desde el punto de vista del intercambio de mensajes. Las mismas no contemplan los datos intercambiados entre los distintos participantes y, por lo tanto, no pueden capturar la semántica asociada a los datos. Esto es un factor limitante a la hora de utilizar *CFSMs* como formalismo de especificación de servicios de software.

La necesidad de considerar valores asociados a modelos de comunicación no es nueva. En [5], Delzanno y Bultan utilizan guardas sobre un conjunto fijo y global de variables booleanas. Estas guardas están asociadas a las transiciones y actúan como pre y post-condiciones que predican sobre las variables posibilitando una estructura de control más rica.

En [8], Fu, Bultan y Su proponen una clase de *communicating machines* con guardas en las que las guardas predican sobre los valores asociados a los mensajes. Esta nueva clase de *communicating machines* está basada en los *conversation protocols* presentados en [7]. Las guardas introducidas en [8] pueden interpretarse como una especificación de la transformación sufrida por los valores al ejecutarse la transición a la que están asociadas.

Si bien las *CFSMs* y los *conversation protocols* son similares, existen algunas diferencias entre estos dos formalismos. Principalmente, en los *conversation protocols*, tal como son presentados en [8, 7], cada máquina posee un único *buffer* desde el cual recibir mensajes. Por lo tanto este modelo no resulta adecuado para razonar sobre sistemas de comunicación donde cada participante participa de varias sesiones de comunicación. Es decir que este modelo no permite representar programas que se comunican a través de enlaces punto a punto, como ser enlaces TCP. Por otra parte, la visión global de la comunicación provista por los *conversation protocols* carece de construcciones explícitas para describir el paralelismo entre acciones de una coreografía. Al contrario, la visión global de la comunicación en *CFSMs* es provista por un *global graph*. Los *global graphs* [6] son *work-flow graphs* que representan la relación de causalidad existente entre las interacciones así como los *choices* distribuidos y los *fork/join* de los *threads*.

En [13], Vissani et al. proponen extender las *CFSMs* agregando valores a los mensajes y guardas que predican sobre estos valores. Las guardas cumplen el rol de garantías en el caso de transiciones de envío, y de requerimientos en el caso de las transiciones de recepción. Esta extensión da lugar a una nueva clase de *CFSMs*, las *Guarded Communicating Machines* (GCM).

De manera análoga al mecanismo de chequeo presentado en [11], en [13] se presenta un algoritmo para el chequeo de compatibilidad para la nueva clase de *communicating machines*. El algoritmo presentado en [11] consiste en construir el *sistema de transición sincrónico* determinado por las *communicating machines* dadas y chequear si ese sistema cumple con dos propiedades: 1) Propiedad de Representablidad y 2) Propiedad de saltos (*branching property*) . El chequeo de estas dos propiedades es de orden exponencial, sin embargo, en la práctica, el algoritmo se comporta eficientemente. La cantidad de estados del sistema sincrónico construido en [11] está acotada por el producto cartesiano de los conjuntos de estados de cada CFSM. En el caso de [13], en cambio, la cantidad de estados del sistema sincrónico puede ser mucho mayor. Esto se debe a que la información sobre los

valores intercambiados que fue acumulada a lo largo de dos caminos distintos que arriban a un estado particular puede variar. De este modo, lo que en el sistema sincrónico construido en [11] era un único estado, en el sistema sincrónico construido en [13] puede desdoblarse en tantos estados como caminos *distintos* arriban a ese estado.

La cantidad de estados generados al construir el sistema sincrónico dependerá del criterio elegido para determinar la equivalencia entre los mismos. El objetivo de esta tesis consiste entonces en desarrollar una **implementación del algoritmo de síntesis del sistema sincrónico** para un conjunto de *GCMs* que sea paramétrico en el lenguaje utilizado para describir las condiciones sobre los valores y que sea eficiente. Para ello, comenzamos por dar un criterio de equivalencia entre estados que, bajo ciertas condiciones (esencialmente la inexistencia de dependencias transitivas en los datos) resulta correcto y eficiente. Como parte de la evaluación del algoritmo proponemos el diseño y la implementación de una serie de casos de estudio que nos permitan evaluar el desempeño del mismo.

El presente trabajo se organiza como sigue: En el Cap. 2 presentamos las definiciones y conceptos preliminares, así como el algoritmo de síntensis propiamente dicho y la primera contribución, el criterio de equivalencia entre estados. En el Cap. 3 se explican los detalles de diseño e implementación de la herramienta y se explica la característica de modificabilidad que presenta la misma. Luego, en el Cap. 4 se presentan y explican los casos de estudio desarrollados así como los resultados obtenidos con los mismos. En el Cap. 5 recapitulamos y analizamos los resultados obtenidos, planteamos las conclusiones y posibles líneas para continuar este trabajo.

2. PRELIMINARES

En la Sec. 2.1 introducimos las definiciones sobre las cuales se basa el trabajo [11] ya que son necesarios para entender los cambios propuestos por Vissani et al. en [13] que pueden encontrarse en la Sec. 2.2. Se debe tener en cuenta que, por una cuestión de coherencia, el criterio de equivalencia entre estados se presenta en la Sec. 2.2.4.

Por último, en la Sec. 2.3 se encuentra el pseudocódigo del algoritmo propuesto en el trabajo [13].

2.1. Communicating Finite State Machines y chequeo de compatibilidad

En esta sección desarrollamos las definiciones de Communicating Finite State Machine y Synchronous Transition System, que nos servirán para explicar el chequeo de compatibilidad, que se muestra en la Sec. 2.3. El lector familiarizado con estas definiciones puede saltar a la Sec. 2.2 donde se presentan cambios sobre estas definiciones.

Vamos a representar procesos como *communicating finite state machines* (CFSMs). Las CFSMs son máquinas de estados finitas que se comunican asincrónicamente a través de un *buffer* unidireccional, no acotado y con preservación del orden. Por lo tanto, una CFSM es una máquina de estados finita cuyas transiciones están etiquetadas por acciones denotando operaciones de lectura o escritura sobre los *buffers*. Consideramos un conjunto de nombres de participantes $\mathcal{P}$: $\mathsf{p}, \mathsf{q}, \ldots$, y un conjunto de nombres de acciones Σ: $\mathsf{a}, \mathsf{b}, \ldots$. Luego, el conjunto de etiquetas está definido por la siguiente gramática:

$$Act_\Sigma ::= \mathsf{pq!a} \mid \mathsf{pq?a}$$

Una acción $\ell \in Act$ es una acción de *envío* $\ell = \mathsf{pq!a}$, entendida como p escribe el mensaje $\mathsf{a} \in \Sigma$ en el *buffer* para q, o bien una acción de *recepción* $\ell = \mathsf{pq?a}$, entendida como q tiene como input un mensaje a en el buffer en el que p le escribe.

Definición 1. Communicating Finite State Machines, CFSMs

Una *communicating finite state machine* es un sistema de transición finito dado por una 4-upla $M = (Q, q_0, \Sigma, \rightarrow)$ donde

- Q es un conjunto finito de *estados*,
- $q_0 \in Q$ es el estado inicial, y
- $\rightarrow \subseteq Q \times Act_\Sigma \times Q$ es un conjunto de *transiciones*; escribimos $q \xrightarrow{\ell} q'$ para $(q, \ell, q') \in \rightarrow$ (y $q \rightarrow q'$ cuando ℓ es inmaterial).

Dado un conjunto $\mathsf{P} \subset \mathcal{P}$ y una CFSM $M_\mathsf{p} = (Q_\mathsf{p}, q_{0\mathsf{p}}, \Sigma, \rightarrow_\mathsf{p})$ para cada $\mathsf{p} \in \mathsf{P}$, la tupla $\mathbf{S} := (M_\mathsf{p})_{\mathsf{p} \in \mathsf{P}}$ es un *communicating system*.

La semántica de un *communicating system* está dada por un sistema de transiciones usando *configuraciones*, que mantienen el estado de cada máquina y el contenido de cada *buffer* del sistema. Dado un *communicating system* $\mathbf{S} := (M_\mathsf{p})_{\mathsf{p} \in \mathsf{P}}$, los *buffers* de $\mathbf{S}$ son

$C = \{\mathsf{pq} \mid \mathsf{p}, \mathsf{q} \in \mathsf{P} \ \wedge \ \mathsf{p} \neq \mathsf{q}\}$. El contenido w_{pq} del canal pq está dado por una secuencia de nombres de acciones, i.e., $w_{\mathsf{pq}} \in \Sigma^*$.

Definición 2 (Semántica de *communicating system*)**.** Sea $\mathbf{S} = (M_{\mathsf{p}})_{\mathsf{p}\in\mathsf{P}}$ un *communicating system*. Una *configuración* de $\mathbf{S}$ es un par $s = \langle \mathbf{q} \ ; \ \mathbf{w} \rangle$ donde $\mathbf{q} \in (Q_{\mathsf{p}})_{\mathsf{p}\in\mathsf{P}}$ y $\mathbf{w} = (w_{\mathsf{pq}})_{\mathsf{pq}\in C}$ con $w_{\mathsf{pq}} \in \Sigma^*$. Escribimos $\mathsf{C}_{\mathbf{S}}$ para el conjunto de todas las posibles configuraciones de $\mathbf{S}$. La semántica de $\mathbf{S}$ está dada por el LTS $\mathcal{A}(\mathbf{S}) = (\mathsf{C}_{\mathbf{S}}, Act_\Sigma, \rightarrow)$ con $\rightarrow \subseteq \ \mathsf{C}_{\mathbf{S}} \times Act_\Sigma \times \mathsf{C}_{\mathbf{S}}$ definido por las siguientes reglas:

$$\frac{\mathsf{s} \in \mathsf{P} \qquad q_{\mathsf{s}} \xrightarrow{\mathsf{sr!a}} q'_{\mathsf{s}} \qquad w'_{\mathsf{sr}} = w_{\mathsf{sr}}.a \qquad (\forall \mathsf{p} \neq \mathsf{s}) q'_{\mathsf{p}} = q_{\mathsf{p}} \qquad (\forall c \neq \mathsf{sr}) w'_c = w'_c}{\langle (q_{\mathsf{p}})_{\mathsf{p}\in\mathsf{P}} \ ; \ (w_c)_{c\in C} \rangle \xrightarrow{\mathsf{sr!a}} \langle (q'_{\mathsf{p}})_{\mathsf{p}\in\mathsf{P}} \ ; \ (w'_c)_{c\in C} \rangle} \textsc{Send}$$

$$\frac{\mathsf{r} \in \mathsf{P} \qquad q_{\mathsf{r}} \xrightarrow{\mathsf{sr?a}} q'_{\mathsf{r}} \qquad w_{\mathsf{sr}} = a.w'_{\mathsf{sr}} \qquad (\forall \mathsf{p} \neq \mathsf{r}) q'_{\mathsf{p}} = q_{\mathsf{p}} \qquad (\forall c \neq \mathsf{sr}) w'_c = w'_c}{\langle (q_{\mathsf{p}})_{\mathsf{p}\in\mathsf{P}} \ ; \ (w_c)_{c\in C} \rangle \xrightarrow{\mathsf{sr?a}} \langle (q'_{\mathsf{p}})_{\mathsf{p}\in\mathsf{P}} \ ; \ (w'_c)_{c\in C} \rangle} \textsc{Rcv}$$

El conjunto de *configuraciones alcanzables de* $\mathcal{S}(\mathbf{S})$ es

$$\mathbf{RS}(\mathbf{S}) = \{s \mid \langle (q_{0\mathsf{p}})_{\mathsf{p}\in\mathsf{P}} \ ; \ (\epsilon)_{c\in C} \rangle \Rightarrow s\}$$

donde dadas s, s' dos configuraciones de $\mathbf{S}$ decimos que $s \Rightarrow s'$ si y solo si existen $s_1, \ldots, s_n$ tales que $s \rightarrow s_1 \rightarrow \ldots \rightarrow s_n \rightarrow s'$.

La regla Send define el comportamiento del participante s poniendo el mensaje a en el *buffer* sr, mientras la regla Rcv modela al participante r consumiendo el mensaje a del *buffer* sr. Notar que un participante s no puede escribir mensajes en el *buffer* w_{pq} cuando $\mathsf{p} \neq \mathsf{s}$. Del mismo modo, r no puede leer mensajes desde w_{pq} cuando $\mathsf{q} \neq \mathsf{r}$.

En principio, dado un *communicating system*, no hay garantía de que los participantes vayan a interoperar correctamente (i.e. libres de errores de comunicación). La noción de correctitud que nosotros adoptamos requiere que los sistemas sean *deadlock-free* y no tengan configuraciones con *recepciones no especificadas* o *mensajes huérfanos*.

La condición de estar libre de *deadlock* es estándar, así que sólo comentaremos las otras dos condiciones. La configuración de un sistema es una configuración con *mensajes huérfanos* cuando tiene un *buffer* no vacío y cada máquina está en un estado sin transiciones de salida. La configuración de un sistema es una configuración con *recepciones no especificadas* cuando tiene una máquina en un estado solo con transición de recepción, y ninguna de ellas puede ejecutarse y el *buffer* correspondiente a al menos una de esas transiciones no está vacío (esto es que la máquina recibió algún mensaje pero era inesperado).

Definición 3 (Sistema de transición sincrónico)**.** Sea $\mathbf{S} = (M_{\mathsf{p}})_{\mathsf{p}\in\mathsf{P}}$ un *communicating system*. La semántica sincrónica de $\mathbf{S}$ está dada por el LTS $\mathcal{S}(\mathbf{S}) = ((Q_{\mathsf{p}})_{\mathsf{p}\in\mathsf{P}}, Int_\Sigma, \rightarrow)$ con $\rightarrow \subseteq \ (Q_{\mathsf{p}})_{\mathsf{p}\in\mathsf{P}} \times Int_\Sigma \times (Q_{\mathsf{p}})_{\mathsf{p}\in\mathsf{P}}$ definido por la siguiente regla:

$$\frac{\mathsf{s}, \mathsf{r} \in \mathsf{P} \qquad q_{\mathsf{s}} \xrightarrow{\mathsf{sr!a}} q'_{\mathsf{s}} \qquad q_{\mathsf{r}} \xrightarrow{\mathsf{sr?a}} q'_{\mathsf{r}} \qquad (\forall \mathsf{p} \notin \{\mathsf{s}, \mathsf{r}\}) \ q'_{\mathsf{p}} = q_{\mathsf{p}}}{(q_{\mathsf{p}})_{\mathsf{p}\in\mathsf{P}} \xrightarrow{\mathsf{s}\twoheadrightarrow\mathsf{r}:a} (q'_{\mathsf{p}})_{\mathsf{p}\in\mathsf{P}}} \textsc{Int}$$

2.1.1. Generalised Multiparty Compatibility

En esta sección presentamos las definiciones necesarias para poder dar la condición de *Generalised Multiparty Compatibility* (GMC). Esta condición nos permitirá sintetizar una coreografía (*Global Graph*) a partir de un sistema sincrónico. La misma depende de dos condiciones: a) *representabilidad* (Def. 5) que establece que para cada máquina del sistema, cada traza y cada *choice* están representados en el sistema sincrónico; y b) *branching property* (Def. 6) cuando hay un *choice* en el sistema de transición sincrónico, una única máquina toma la decisión y cada uno de los otros participantes es, o bien puesto al tanto de la decisión o no está involucrado en la misma. La propiedad de representabilidad asegura que el sistema de transición sincrónico tiene suficiente información para poder determinar cualquier propiedad de *safety* de cualquier ejecución asincrónica mientras que, la propiedad de *branching* garantiza que, si una ramificación en el sistema sincrónico representa un *choice* entonces este *choice* está "bien formado".

Para poder definir estar propiedades necesitamos primero algunas definiciones auxiliares:

Definición 4 (Proyecciones)**.** Dado un *communicating system* $\mathbf{S} = (M_\mathsf{p})_{\mathsf{p}\in\mathsf{P}}$ y su respectivo sistema de transición sincrónico $\mathcal{S}(\mathbf{S}) = ((Q_\mathsf{p})_{\mathsf{p}\in\mathsf{P}}, Int_\Sigma, \rightarrow)$ definimos la proyección de un evento $e = (q_\mathsf{p})_{\mathsf{p}\in\mathsf{P}} \xrightarrow{\mathsf{s}\twoheadrightarrow\mathsf{r}:a} (q'_\mathsf{p})_{\mathsf{p}\in\mathsf{P}}$ al participante p (y lo notamos $e\upharpoonright_\mathsf{p}$) como sigue:

$$e\upharpoonright_\mathsf{p} = \begin{cases} q_\mathsf{p} \xrightarrow{\mathsf{pr}!a} q'_\mathsf{p} & (\mathsf{p} = \mathsf{s}) \\ q_\mathsf{p} \xrightarrow{\mathsf{sp}?a} q'_\mathsf{p} & (\mathsf{p} = \mathsf{r}) \\ q_\mathsf{p} \xrightarrow{\epsilon} q_\mathsf{p} & (\mathsf{p} \neq \mathsf{s} \wedge \mathsf{p} \neq \mathsf{r}) \end{cases}$$

Las proyecciones sobre secuencias de eventos se definen de la manera obvia. De la misma manera se puede definir $\mathcal{S}(\mathbf{S})\upharpoonright_\mathsf{p}$ (la proyección del sistema sincrónico $\mathcal{S}(\mathbf{S})$ al participante p) de forma directa. Se debe notar sin embargo que será necesario agregar transiciones ϵ a las acciones de p para que la proyección sea coherente con la Def. 4.

Definición 5 (Representabilidad)**.** Decimos que un sistema $\mathbf{S} = (M_\mathsf{p})_{\mathsf{p}\in\mathsf{P}}$ es representable si para todo $\mathsf{p} \in \mathsf{P}$:

1. El lenguaje generado por la máquina M_p (denotado como $\mathcal{L}(M_\mathsf{p})$) es igual al lenguaje generado por la proyección del sistema sincrónico $\mathcal{S}(\mathbf{S})\upharpoonright_\mathsf{p}$ y

2. para todo estado $q \in Q_\mathsf{p}$ existe un estado n del sistema sincrónico $\mathcal{S}(\mathbf{S})$ tal que la p-ésima componente de n es q (es decir, el estado correspondiente a la máquina p en n es q) y todas las acciones que salen del estado q son parte de alguna palabra del lenguaje generado por la proyección $\mathcal{S}(\mathbf{S})\upharpoonright_\mathsf{p}$ a partir del estado $n\upharpoonright_p$.

Definición 6 (Branching property)**.** Decimos que un sistema $\mathbf{S} = (M_\mathsf{p})_{\mathsf{p}\in\mathsf{P}}$ tiene la propiedad de *branching* si el sistema sincrónico generado respeta lo siguiente:

1) O bien para todo nodo n del cual se pueda salir por dos ejes distintos etiquetados con e_1 y e_2, entonces los nodos de llegada de esos ejes se juntan en un tercer nodo n' a partir de la transición complementaria e_2 y e_1 respectivamente. Es decir, si todo *branch* es la ejecución concurrente de un par de acciones,

2) o bien

   (a) Para todo participante p

      I. la primera acción de p en cada rama es distinta o,

      II. p no está involucrado en ninguna de las ramas. Es decir que las ramas se juntan antes de que p realice cualquier acción.

   (b) un único participante es el responsable de realizar la elección de la rama y,

   (c) para todo participante r involucrado en la elección no hay *reace conditions* entre los mensajes que puede recibir.

Para una definición formal de las Defs. 5 y 6 referimos al lector a [11]. El chequeo de la propiedad de representabilidad puede realizarse en tiempo exponencial en la cantidad de estados mientras que el chequeo de la propiedad de *branching* es de orden factorial en la cantidad de transiciones. Sin embargo, en la práctica, estos chequeos se realizan en un tiempo razonable.

Definición 7 (Generalised Multiparty Compatibility)**.** Un *communicating system* $\mathbf{S} = (M_{\mathsf{p}})_{\mathsf{p}\in\mathsf{P}}$ es GMC si es representable y tiene la propiedad de *branching*.

Teorema 1 (Soundness)**.** *Si* $\mathbf{S} = (M_{\mathsf{p}})_{\mathsf{p}\in\mathsf{P}}$ *es GMC entonces es* seguro*. Esto significa que está libre de deadlock, no tiene mensajes huérfanos ni recepciones no especificadas.*

2.2. Guarded Communicating Machines

En esta sección desarrollamos los cambios propuestos en [13] sobre las CFSM para incorporar condiciones sobre los datos. Se definen también los Sistemas de Transición Sincrónicos para GCFSM y su semántica.

2.2.1. GCMs

Para este trabajo se van a adoptar predicados sobre los datos asociados a los mensajes que envían los participantes del sistema. La explicación formal sería: cuando una acción $\mathsf{pq!m}\langle x\rangle$ tiene asociada una condición φ, entonces la transición es entendida como *"El participante* p *envía el mensaje* m *al participante* q *tal que el valor* x *asociado a* m *satisface* φ*"* siendo una suerte de garantía, y cuando una acción $\mathsf{pq?m}(x)$ tiene asociada una condición φ, entonces la transición es entendida como *"El participante* q *recibe el mensaje* m *del participante* p *tal que el valor* x *asociado a* m *satisface* φ*"* siendo una suerte de asunción.

El algoritmo presentado es paramétrico en el lenguaje específico utilizado para definir las condiciones. Sólo requerimos que la lógica subyacente sea monótona (i.e. si $\Gamma \vdash \varphi$ entonces $\Gamma \cup \{\alpha\} \vdash \varphi$) y que el lenguaje tenga conjunción ($\wedge$), una relación de demostrabilidad $\Gamma \vdash \varphi$ con $\Gamma \subseteq \mathbb{G}$ y $\varphi \in \mathbb{G}$ ($\mathbb{G}$ el conjunto de fórmulas), *top* ($\top$) y *bottom* ($\bot$). En lo que resta del trabajo asumiremos que estamos trabajando en un fragmento decidible de modo que la relación $\vdash$ puede ser computada.

Extendemos la sintaxis de etiquetas de acción (ℓ) para contar con los datos intercambiados durante la comunicación:

$$Act_{\Sigma} ::= \mathsf{pq!a}\langle\mathbf{x}\rangle \mid \mathsf{pq?a}(\mathbf{x})$$

Denotamos al conjunto de *variables de interacción libres* de ℓ con $\mathrm{fv}(\ell)$. Asumimos, sin pérdida de generalidad, que la misma variable no es utilizada en dos transiciones distintas para evitar colapsos de nombres. Eso significa que, dadas $q \xrightarrow[\alpha]{\ell} q'$ y $s \xrightarrow[\alpha']{\ell'} s'$ entonces $\mathrm{fv}(\ell) \cap \mathrm{fv}(\ell') = \varnothing$. Se debe notar que no imponemos la misma restricción a las condiciones α y α' de modo que es posible establecer ciertas relaciones entre las variables intercambiadas en una interacción y las variables intercambiadas en otra interacción.

En la Sec. 2.2.2 damos la semántica para las GCMs. La misma se da en términos de *encarnaciones* de las variables. Esto quiere decir que, cada vez que se ejecuta una transición, consideramos una nueva encarnación para cada una de las variables que son enviadas y/o recibidas en esa transición. En este contexto, las sentencias se evalúan sobre la última encarnación de cada variable. Es importante aclarar que prohibimos las dependencias transitivas en los datos. Es decir que no admitimos sistemas en los que las condiciones sobre la encarnación de una variable dependan de una encarnación anterior de la misma variable. Esta imposición se establece para evitar que el sistema sincrónico generado sea infinito. Esencialmente se puede pensar que estamos evitando que exista la posibilidad de "sumar", es decir de establecer condiciones del tipo $x_{i+1} = x_i + 1$. Como una condición extra, vamos a pedir que la validez de una sentencia cualquiera dependa, a lo sumo, de una encarnación de cada variable. Estas dos condiciones nos van a permitir establecer la correctitud del criterio de equivalencia y la convergencia del algoritmo.

Por último, finalizamos esta sección dando la definición de GCM:

Definición 8. Guarded Communicating Finite State Machine

Una CFSM con condiciones (GCM) es una 5-upla $M = (Q, q_0, \Sigma, \rightarrow, \mathcal{G})$ donde $(Q, q_0, \Sigma, \rightarrow)$ es una CFSM y $\mathcal{G}$ es una función que asigna una condición a cada transición.

Escribimos $q \xrightarrow[\alpha]{\ell} q'$ cuando $q \xrightarrow{\ell} q'$ y $\mathcal{G}(q \xrightarrow{\ell} q') = \alpha$. Llamamos $\mathrm{fv}(M)$ al conjunto de variables libres de M, i.e., $\mathrm{fv}(M) = \bigcup_{q \xrightarrow[\alpha]{\ell} q'} (\mathrm{fv}(\ell) \cup \mathrm{fv}(\alpha))$.

2.2.2. Semántica asincrónica para GCMs

A continuación damos un conjunto de definiciones auxiliares (Def. 9–14) que nos servirán para definir formalmente la semántica de las GCMs. La primera definición nos permite encarnar (renombar) las variables libres en una sentencia. Las siguientes dos definiciones formalizan la noción de conocimiento asociado a una variable y la clausura del mismo (Def. 10) y la noción de conocimiento asociado a un vector de variables y la clausura del mismo (Def. 11).

Definición 9. La función de *instanciación* $inst : \mathbb{G} \times (\mathcal{V} \to \mathbb{N}) \to \mathbb{G}$ es una función que, dada una sentencia $\alpha \in \mathbb{G}$ y una función de *encarnación* $\mathcal{I} : \mathcal{V} \to \mathbb{N}$ devuelve la reescritura de la sentencia α que se obtiene a partir de reemplazar cada variable libre x por la variable x_i donde i es la encarnación correspondiente a la variable x de acuerdo a $\mathcal{I}$:

$$inst(\alpha, \mathcal{I}) = \alpha[x_{\mathcal{I}(x)}/x]_{x \in \mathrm{fv}(\alpha)}$$

Definición 10. Dada una función $G : \mathcal{V} \to \mathbb{G}$ definimos $G^* : \mathcal{V} \to 2^{\mathbb{G}}$ como

$$G^*(x) = \bigcup_{i \in \mathbb{N}} G_i(x)$$

donde

$$\begin{aligned} G_0(x) =& \{G(x)\} \\ G_i(x) =& \{G(x') \mid x' \in fv(G_{i-1}(x))\} \end{aligned}$$

Definición 11. La función $\mathcal{K} : 2^{\mathcal{V}} \times (\mathcal{V} \to \mathbb{N}) \times (\mathcal{V} \to \mathbb{G}) \to (\mathcal{V} \to \mathbb{G})$ representa la función de clausura del conocimiento. Lo que hace es, dado un conjunto de variables $\vec{\mathbf{x}}$ y un diccionario $G : \mathcal{V} \to \mathbb{G}$ es retornar las sentencias que están, directa o indirectamente, involucradas en la definición de $\vec{\mathbf{x}}$:

$$\mathcal{K}(\vec{\mathbf{x}}, \mathcal{I}, G) = \kappa : \mathcal{V} \to \mathbb{G}$$

tal que

$$(\forall v \in \mathcal{V})\kappa(v) = \begin{cases} \top & v \notin \{x_{\mathcal{I}(x)} \mid x \in \vec{\mathbf{x}}\} \\ \bigwedge G^*(v) & v \in \{x_{\mathcal{I}(x)} \mid x \in \vec{\mathbf{x}}\} \end{cases}$$

Observación 12. *Dado un vector $\vec{\mathbf{x}}$ de variables en $\mathcal{V}$ denotamos con $len(\vec{\mathbf{x}})$ a la longitud del vector $\vec{\mathbf{x}}$ y denotamos con $\vec{\mathbf{x}}[i]$ (con $0 \leqslant i < len(\vec{\mathbf{x}})$) a la i-ésima variable en $\vec{\mathbf{x}}$.*

Las siguiente dos definiciones formalizan la noción de *map* (diccionario) de variable perteneciente a $\mathcal{V}$ a sentencia en $\mathbb{G}$ y de la operación de actualización por conjunción mediante el agregado de una nueva sentencia (Def. 13) o mediante la "unión" de dos *maps* distintos (Def. 14). Estas definiciones nos serán de utilidad para expresar cómo evolucionan los contextos a medida que las GCMs van ejecutando.

Definición 13. Dada una función $G : \mathcal{V} \to \mathbb{G}$ y una sentencia $\gamma \in \mathbb{G}$, definimos la función $\wedge : (\mathcal{V} \to \mathbb{G}) \times \mathbb{G} \to (\mathcal{V} \to \mathbb{G})$ como sigue:

$$\wedge(G, \gamma) = G'$$

tal que

$$(\forall v \in \mathcal{V})G'(v) = \begin{cases} G(v) & v \notin \mathrm{fv}(\gamma) \\ G(v) \wedge \gamma & v \in \mathrm{fv}(\gamma) \end{cases}$$

Definición 14. Dada una función $G : \mathcal{V} \to \mathbb{G}$ y una función $\kappa : \mathcal{V} \to \mathbb{G}$, definimos la función $\wedge : (\mathcal{V} \to \mathbb{G}) \times (\mathcal{V} \to \mathbb{G}) \to (\mathcal{V} \to \mathbb{G})$ como sigue:

$$\wedge(G, \kappa) = G'$$

tal que

$$(\forall v \in \mathcal{V})G'(v) = G(v) \wedge \kappa(v)$$

Ahora sí estamos en condiciones de dar la semántica de un *communicating system* para GCMs.

Fácilmente extendemos la definición de *communicating system* para considerar GCMs en vez de sólo CFSMs, pero requerimos que cualquier GCM se defina usando sólo variables locales, i.e., para $\mathbf{S} := (M_{\mathsf{p}})_{\mathsf{p}\in\mathsf{P}}$ se mantiene que $\mathsf{p} \neq \mathsf{q}$ implica $\mathrm{fv}(M_{\mathsf{p}}) \neq \mathrm{fv}(M_{\mathsf{q}})$.

El comportamiento asincrónico de una GCM se obtiene al considerar las configuraciones con dos componentes extra. Uno de ellos describe las condiciones de las transiciones ya ejecutadas para alcanzar un estado particular. Este componente extra lo llamamos *context*. El otro componente mantiene la última encarnación utilizada de cada variable. Notar que, dado que una variable solo puede ser enviada/recibida en una única transición, la cantidad de encarnaciones de una variable denota también la cantidad de veces que se transitó dicha transición.

Definición 15 (Semántica de *communicating system* para GCMs)**.** Sea $\mathbf{S} = (M_{\mathsf{p}})_{\mathsf{p}\in\mathsf{P}}$ un *communicating system*. Una *configuración* de $\mathbf{S}$ es una 4-upla $s = \langle \mathbf{G} \ ; \ \mathbf{q} \ ; \ \mathbf{w} \ ; \ I\rangle$ donde $\mathbf{G} = (G_{\mathsf{p}})_{\mathsf{p}\in\mathsf{P}}$ con $G_{\mathsf{p}} : \mathcal{V} \to \mathbb{G}$, $\mathbf{q} \in (Q_{\mathsf{p}})_{\mathsf{p}\in\mathsf{P}}$, $\mathbf{w} = (w_{\mathsf{pq}})_{\mathsf{pq}\in C}$ con $w_{\mathsf{pq}} \in \Sigma^*$ y $\mathcal{I} : \mathcal{V} \to \mathbb{N}$. Escribimos $\mathsf{C}_{\mathbf{S}}$ para denotar al conjunto de todas las posibles configuraciones de $\mathbf{S}$. La semántica de $\mathbf{S}$ está dada por el LTS $\mathcal{A}(\mathbf{S}) = (\mathsf{C}_{\mathbf{S}}, Act_{\Sigma}, \to)$ con $\to \subseteq \ \mathsf{C}_{\mathbf{S}} \times Act_{\Sigma} \times \mathsf{C}_{\mathbf{S}}$ definido por las siguientes reglas:

$$\frac{\begin{array}{c}\mathsf{s} \in \mathsf{P} \quad q_{\mathsf{s}} \xrightarrow[\alpha]{\mathsf{sr!a}\langle\vec{\mathbf{x}}\rangle} q'_{\mathsf{s}} \qquad w'_{\mathsf{sr}} = w_{\mathsf{sr}}.\langle a\langle\vec{\mathbf{x}}\rangle, \mathcal{K}(\vec{\mathbf{x}}, \mathcal{I}', G'_{\mathsf{s}})\rangle \\ G'_{\mathsf{s}} = G_{\mathsf{s}} \wedge \iota_{\alpha} \qquad (\forall v \in \mathcal{V}) G'_{\mathsf{s}}(v) \not\models \bot \qquad \mathcal{I}' = \mathcal{I}[x \mapsto \mathrm{máx}(\mathcal{I}) + 1]_{x\in\vec{\mathbf{x}}} \\ (\forall \mathsf{p} \neq \mathsf{s}) q'_{\mathsf{p}} = q_{\mathsf{p}} \qquad (\forall \mathsf{p} \neq \mathsf{s}) G'_{\mathsf{p}} = G_{\mathsf{p}} \qquad (\forall c \neq \mathsf{sr}) w'_c = w_c\end{array}}{\langle (\mathbf{G}_{\mathsf{p}})_{\mathsf{p}\in\mathsf{P}} \ ; \ (\mathbf{q}_{\mathsf{p}})_{\mathsf{p}\in\mathsf{P}} \ ; \ (\mathbf{w_c})_{\mathbf{c}\in\mathbf{C}} \ ; \ I\rangle \xrightarrow{\mathsf{sr!a}\langle\vec{\mathbf{x}}\rangle} \langle (\mathbf{G}'_{\mathsf{p}})_{\mathsf{p}\in\mathsf{P}} \ ; \ (\mathbf{q}'_{\mathsf{p}})_{\mathsf{p}\in\mathsf{P}} \ ; \ (\mathbf{w}'_{\mathbf{c}})_{\mathbf{c}\in\mathbf{C}} \ ; \ I'\rangle}\ \textsc{Send}$$

$$\frac{\begin{array}{c}\mathsf{r} \in \mathsf{P} \quad q_{\mathsf{r}} \xrightarrow[\beta]{\mathsf{sr?a}(\vec{\mathbf{y}})} q'_{\mathsf{r}} \qquad w_{\mathsf{sr}} = \langle a\langle\vec{\mathbf{x}}\rangle, \kappa\rangle.w'_{\mathsf{sr}} \qquad \mathcal{I}' = \mathcal{I}[y \mapsto \mathrm{máx}(\mathcal{I}) + 1]_{y\in\vec{\mathbf{y}}} \\ G'_{\mathsf{r}} = (G_{\mathsf{r}} \wedge \kappa) \wedge \left(\vec{\mathbf{x}} = \vec{\mathbf{y}}\right) \qquad (\forall v \in \mathcal{V}) G'_{\mathsf{r}}(v) \not\models \bot \qquad \gamma \models \iota_{\beta} \\ (\forall \mathsf{p} \neq \mathsf{r}) q'_{\mathsf{p}} = q_{\mathsf{p}} \qquad (\forall \mathsf{p} \neq \mathsf{r}) G'_{\mathsf{p}} = G_{\mathsf{p}} \qquad (\forall c \neq \mathsf{sr}) w'_c = w_c\end{array}}{\langle (\mathbf{G}_{\mathsf{p}})_{\mathsf{p}\in\mathsf{P}} \ ; \ (\mathbf{q}_{\mathsf{p}})_{\mathsf{p}\in\mathsf{P}} \ ; \ (\mathbf{w_c})_{\mathbf{c}\in\mathbf{C}} \ ; \ I\rangle \xrightarrow{\mathsf{sr?a}(\mathbf{x})} \langle (\mathbf{G}'_{\mathsf{p}})_{\mathsf{p}\in\mathsf{P}} \ ; \ (\mathbf{q}'_{\mathsf{p}})_{\mathsf{p}\in\mathsf{P}} \ ; \ (\mathbf{w}'_{\mathbf{c}})_{\mathbf{c}\in\mathbf{C}} \ ; \ I'\rangle}\ \textsc{Rcv}$$

donde

$$\begin{aligned}\iota_{\alpha} &= \mathit{inst}(\alpha, \mathcal{I}') \\ \iota_{\beta} &= \mathit{inst}(\beta, \mathcal{I}') \\ \gamma &= \bigwedge_{v\in\mathrm{fv}(\iota_{\beta})} G'^{*}_{\mathsf{r}}(v)\end{aligned}$$

Las reglas de inferencia definidas arriba son análogas a aquellas en Def. 2 pero ahora: (1) chequeamos la validez de las guardas sobre el estado actual (5ta y 6ta premisas de la regla Rcv) y (2) acumulamos las guardas durante la ejecución (3era premisa de la regla

Send y 4ta premisa de la regla Rcv). La diferencia principal entre las reglas Send y Rcv radica en la forma en la que se manejan los parámetros de los mensajes. Si bien en ambos casos las variables son reencarnadas ocultando las restricciones previamente acumuladas sobre las mismas (lo que es particularmente útil a la hora de manejar máquinas con ciclos), la utilización de las guardas es lo que las distingue. En el caso de los envíos, la guarda sólo se chequea para evitar que la información acumulada tras la ejecución de esa transición genere que el contexto resultante se transforme en una contradicción. Pero fuera de este requisito, la máquina siempre tiene habilitados los envíos. Por el otro lado, en el caso de las recepciones, la máquina utiliza la información transferida en el *buffer*, junto con el contexto, para verificar que la guarda de una transición se deduce de esa información. Sólo en este caso dicha transición de recepción se encuentra habilitada.

2.2.3. Semántica sincrónica para GCMs

Tal como en el caso de las CFSMs, introducimos la noción de comportamiento sincrónico (o ideal) de un *communicating system* para GCMs. Adoptamos como notación que $\{G\}$ es el conjunto de todos los contextos y que $\{\mathcal{I}\}$ es el conjunto de todas las funciones de encarnación.

Definición 16. Sistema de transición sincrónico para GCMs

Sea $\mathbf{S} = (M_{\mathsf{p}})_{\mathsf{p}\in\mathsf{P}}$ un *communicating system* para guarded CFSMs. La semántica sincrónica de $\mathbf{S}$ está dada por el LTS $\mathcal{S}(\mathbf{S}) = ((\{G\}\times\{\mathcal{I}\})\times(Q_{\mathsf{p}})_{\mathsf{p}\in\mathsf{P}}, Int_\Sigma, \rightarrow)$ con $\rightarrow \subseteq (\{G\}\times\{\mathcal{I}\})\times(Q_{\mathsf{p}})_{\mathsf{p}\in\mathsf{P}} \times Int_\Sigma \times (\{G\}\times\{\mathcal{I}\})\times(Q_{\mathsf{p}})_{\mathsf{p}\in\mathsf{P}}$ definida por la siguiente regla:

$$\frac{\begin{array}{c}\mathsf{s},\mathsf{r}\in\mathsf{P} \qquad q_{\mathsf{s}} \xrightarrow[\alpha]{\mathsf{sr!a}\langle\vec{\mathbf{x}}\rangle} q'_{\mathsf{s}} \qquad q_{\mathsf{r}} \xrightarrow[\beta]{\mathsf{sr?a}(\vec{\mathbf{y}})} q'_{\mathsf{r}} \qquad (\forall \mathsf{p}\notin\{\mathsf{s},\mathsf{r}\})\ q'_{\mathsf{p}} = q_{\mathsf{p}} \\ \mathcal{I}' = \mathcal{I}[x\mapsto \text{máx}(\mathcal{I})+1]_{x\in\vec{\mathbf{x}}} \\ G' = G\wedge\iota_\alpha \qquad \gamma \not\models \bot \qquad \gamma\models\iota_\beta\end{array}}{\langle[(G,\mathcal{I})]\ ;\ (q_{\mathsf{p}})_{\mathsf{p}\in\mathsf{P}}\rangle \xrightarrow{\mathsf{s}\to\mathsf{r}:\mathsf{a}} \langle[(G',\mathcal{I}')]\ ;\ (q'_{\mathsf{p}})_{\mathsf{p}\in\mathsf{P}}\rangle}\ \textsc{Int}$$

donde

$$\begin{aligned}\iota_\alpha &= inst(\alpha,\mathcal{I}')\\ \iota_\beta &= inst(\beta,\mathcal{I}')\\ \gamma &= \bigwedge_{v\in\text{fv}(\iota_\beta)} G'^{*}(v)\end{aligned}$$

y donde $[(G',\mathcal{I}')]$ es la clase de equivalencia dada por la noción de equivalencia de la Def. 17.

2.2.4. Criterio de equivalencia y convergencia del algoritmo

En esta sección vamos a presentar el criterio de equivalencia entre estados utilizado. Argumentaremos que el mismo es correcto respecto de la semántica dada a las GCMs, es decir que si dos estados q y q' son equivalentes de acuerdo al criterio dado, entonces los caminos generados a partir de ellos son los mismos. También argumentaremos que, bajo las restricciones impuestas a las GCMs, el cardinal de las clases de equivalencia generadas por la relación de equivalencia dada es finito, de modo que el algoritmo converge.

Definición del criterio de equivalencia y correctitud

Definición 17. Dadas $G : \mathcal{V} \to \mathbb{G}$, $G' : \mathcal{V} \to \mathbb{G}$, $\mathcal{I} : \mathcal{V} \to \mathbb{N}$ y $\mathcal{I}' : \mathcal{V} \to \mathbb{N}$ decimos que

$$(G, \mathcal{I}) \equiv (G', \mathcal{I}')$$

si y solo si

$$\left(\bigwedge \bigcup_{v \in \mathcal{V}} \{ \text{“new\_”} u_i = \text{“old\_”} u_j \mid u_i \in \mathrm{fv}\big(G'^*(v_{\mathcal{I}'(v)})\big) \text{ and } u_j \in \mathrm{fv}\big(G^*(v_{\mathcal{I}(v)})\big) \} \right) \implies \left(\bigwedge G'^*(v_{\mathcal{I}'(v)})[u \mapsto \text{“new\_”} u] \Leftrightarrow G^*(v_{\mathcal{I}(v)})[u \mapsto \text{“old\_”} u] \right)$$

En esencia, el criterio de equivalencia entre contextos establece que el conocimiento acumulado sobre la última encarnación de cada variable debe ser equivalente en ambos. La hipótesis previa a la implicación, que a partir de ahora llamaremos HEq, iguala la última encarnación de cada variable en cada uno de los contextos entre sí **y además** iguala la encarnación de una variable u que aparece en el conocimiento en un contexto con la encarnación de esa misma variable u que aparece en el otro contexto. Notar que, como efecto de las condiciones pedidas (no dependencia transitiva de los datos, etc.) en el conocimiento de una variable en un contexto dado puede aparecer, a lo sumo, una encarnación de cada variable. Por otra parte, al evaluar una fórmula en un contexto, esta evaluación se realiza reemplazando cada variable por su última encarnación. Esto es lo que nos da el argumento para igualarlas. Para simplificar la notación del criterio hemos asumido que toda variable $v_i \in \mathrm{fv}\big(G'^*(v_i)\big)$, es decir que toda v_i aparece mencionada (y libre) en el conocimiento asociado a ella misma.

Proposición 1. El criterio definido en Def. 17 es correcto.

Demostración. Que el criterio sea correcto quiere decir que si $(G, \mathcal{I}) \equiv (G', \mathcal{I}')$ entonces para toda sentencia $\alpha, \beta \in \mathbb{G}$, $(\gamma \implies \iota_\beta) \Leftrightarrow (\gamma' \implies \iota'_\beta)$ donde

$$\begin{aligned} \iota_\alpha &= \mathit{inst}(\alpha, \mathcal{I}) \\ \iota_\beta &= \mathit{inst}(\beta, \mathcal{I}) \\ \gamma &= \bigwedge_{v \in \mathrm{fv}(\iota_\beta)} (G \wedge \iota_\alpha)^*(v) \end{aligned}$$

y

$$\begin{aligned} \iota'_\alpha &= \mathit{inst}(\alpha, \mathcal{I}') \\ \iota'_\beta &= \mathit{inst}(\beta, \mathcal{I}') \\ \gamma' &= \bigwedge_{v \in \mathrm{fv}(\iota'_\beta)} (G' \wedge \iota'_\alpha)^*(v) \end{aligned}$$

Para probar que esto es así vamos a considerar modelos $\mathcal{M}$ sobre el conjunto de variables $\{\text{“new\_”} v \mid v \in \mathcal{V}\} \cup \{\text{“old\_”} v \mid v \in \mathcal{V}\}$ y renombramos las variables de las sentencias apropiadamente. Esto es, las variables de γ' y β' son prefijadas con “new_” y las variables de γ y β con “old_”. Además para todo modelo $\mathcal{M}$ de $(\gamma \implies \iota_\beta)$ (recordar que las

consideramos renombradas apropiadamante) existe un modelo equivalente $\mathcal{M}'$ que respeta HEq. Como $\mathcal{M}'$ respeta HEq entonces, en $\mathcal{M}'$, γ y γ' son equivalentes (como resultado de que los contextos son equivalentes). ι_β y ι'_β son directamente la misma fórmula pero con las variables renombradas, variables que ya dijimos que son iguales, y por lo tanto son equivalentes también. Luego se deduce que $(\gamma \implies \iota_\beta) \Leftrightarrow (\gamma' \implies \iota'_\beta)$. □

Proposición 2. El sistema de transición definido en Def. 16 considerado a partir del estado $\langle(\lambda v.\top, \lambda v.0)\ ;\ (q_{0\mathsf{p}})_{\mathsf{p}\in\mathsf{P}}\rangle$ es finito.

La demostración de la Prop. 2 se reduce a demostrar que recorrer dos veces el mismo camino lleva a dos configuraciones equivalentes en el sentido de la Def. 17. Luego, si llamamos camino simple a una configuración a aquellas sucesiones de transiciones que nunca recorre más de una vez cada ciclo, como la cantidad de caminos sin ciclos desde el estado inicial hacia otro estado es finita, la cantidad de contextos distintos también lo es. Si bien no daremos una demostración de este punto (queda como trabajo futuro) argumentaremos que cada vez que se recorre un camino simple hasta un estado particular, debido a la ausencia de dependencias transitivas, la información acumulada sobre cada una de las variables "vivas" (la última encarnación de cada variable) es la misma. Dado que cualquier sentencia que se quiera verificar será instanciada sobre las variables vivas, entonces el contexto que se obtiene es equivalente a uno anteriormente generado.

2.3. Algoritmo

A continuación se encuentra el pseudocódigo del algoritmo propuesto en [13] para la construcción del Synchronous Transition System.

Definición:

Sea $\mathcal{V}$ un conjunto de variables, el *contexto* C una estructura $\langle inc_{\mathsf{C}} : \mathcal{V} \mapsto \mathbb{N}, K_{\mathsf{C}} : \mathcal{V} \times \mathbb{N} \mapsto \mathcal{F}orm^{\mathrm{FOL}}\rangle$. Donde inc_{C} mapea cada variable con el numero de *encarnación* (veces que ha sido instanciada) en el contexto actual y K_{C} mapea cada variable y su encarnación a un conjunto de predicados.

```
1  function BuildSTS(input {M_p}_{p∈P})
2    (*  Donde M_p = (Q_p, q_0p, Σ_p, →_p, G_p) es un GCFSM para cada p.
3      Por simplicidad requerimos que las variables usadas
4      en los mensajes sea disjuntas incluso entre
5      diferentes maquinas.
6      Notar que esta limitacion es facil de superar. *)
7    initialCtx ← ⟨0 : V ↦ {0}, ⊤ : V × N_0 ↦ {⊤}⟩
8    (* Creamos el contexto inicial, es decir, el contexto del
9      primer estado del STS.
10     Notar que el uso de la funcion constante 0 como
11     map de encarnacion para el estado inicial significa
12     que ninguna variable fue instanciada aun. Del mismo modo,
13     el uso de la funcion constante ⊤ como el
14     map de predicados significa que, hasta el momento,
15     no hay condiciones sobre las variables. *)
16   q_0 ← ⟨(q_0p)_{p∈P}, initialCtx⟩
17   (* definimos el primer STS State como los estados
```

```
18    iniciales de cada una de las maquinas y el initialCtx *)
19  Q_sts ← {q_0}
20  (* Agregamos el estado inicial al conjunto de STS States *)
21  →_sts ← {}
22  (* Inicializamos el conjunto de transiciones*)
23  while Q_sts changes
24    forall ⟨q, C⟩ in Q_sts
25    (* Mientras cambie el conjunto de STS States,
26        lo iteramos para explorar cada estado *)
27      forall p ∈ P forall r ∈ P
28      (* por cada STS State recorremos las maquinas
29        de a pares *)
30        forall q_p --pr!a⟨x⃗⟩ / X(x⃗,χ⃗)--> q'_p in →_p forall q_r --pr?a(y⃗) / Y(y⃗,ι⃗)--> q'_r in →_r
31          (* Tomamos pares de transiciones de igual mensaje
32          correspondientes al mismo canal
33            Requerimos:
34            • fv(X) = x⃗ ∪ χ⃗  y  fv(Y) = y⃗ ∪ ι⃗ .
35            • x⃗ ∩ χ⃗ = ∅  y  y⃗ ∩ ι⃗ = ∅ .
36          *)
37          if ∃χ_i ∈ χ⃗ | inc_C(χ_i) = 0 or ∃ι_i ∈ ι⃗ | inc_C(ι_i) = 0 then
38            (* Si hay alguna variable libre entonces ambas
39            transiciones no son compatibles *)
40            continue
41          end if
42
43          (* Cada vez que una variable es instanciada creamos
44          una nueva generacion. Las variables que estan
45          siendo instanciadas en este paso pertenecen a esta
46          nueva generacion (i.e. x⃗ y
47          y⃗). El numero de generacion se obtiene sumando
48          uno al valor previo. Es decir, sumar uno al
49          maximo de la imagen de la funcion inc_C. *)
50          inc'(v) ← { max(Img(inc_C)) + 1   if v ∈ x⃗ ∪ y⃗
                      { inc_C(v)              oterwhise
51
52          (* en el predicado, reemplazo las variables libres
53          con la encarnacion apropiada de cada una *)
54          X' ← X(x⃗, χ⃗)|^{inc'(v)}_{v∈x⃗∪χ⃗}
55          Y' ← Y(y⃗, ι⃗)|^{inc'(v)}_{v∈y⃗∪ι⃗}
56
57          (* agrego la equivalencia entre x⃗ y y⃗
58            a la base de conocimiento *)
```

59 $$K'(v,i) \leftarrow \begin{cases} x_n^{inc'(x_n)} = y_n^{inc'(y_n)} \wedge X' & \text{if } v = x_n \wedge i = inc'(v) \\ x_n^{inc'(x_n)} = y_n^{inc'(y_n)} & \text{if } v = y_n \wedge i = inc'(v) \\ K_{\mathsf{C}}(v,i) \wedge X' & \text{if } v \in \vec{\chi} \wedge i = inc'(v) \\ K_{\mathsf{C}}(v,i) & \text{oterwhise} \end{cases}$$

60

```
61  (* ahora podemos construir la hipotesis *)
62  hyp ← ⊤
63  forall v ∈ x⃗ ∪ χ⃗ ∪ ι⃗
64    hyp ← hyp ∧ BuildHypothesis(v, K', inc', ∅)
65  end forall
66
67  if hyp ⊢ ⊥ then
68    (* si llegamos a una contradiccion,
69       estas dos transiciones no son compatibles *)
70    continue
71  end if
72
73  if hyp ⊢ Y' then
74    q' ← q|^{q'_p}_{q_p} |^{q'_r}_{q_r}
75
76    (* busco si existe algun estado equivalente *)
77    forall ⟨q'', ⟨inc'', K''⟩⟩ ∈ Q_sts
78      if q' = q'' then
79        (* Ahora chequeo si la informacion sobre
80           la ultima encarnacion de cada variable
81           es equivalente en ambos estados *)
82        sameVars ← (∀v ∈ 𝒱) inc'(v) > 0 ⇔ inc''(v) > 0
83        if ¬sameVars then
84          (* Si estos dos estados no comparten
85          las variables instanciadas entonces no son
86          equivalentes *)
87          continue
88        end if
89
90        (* Ahora reviso si los contextos son equivalentes *)
91        equiv ← ⟨inc', K'⟩ ≡ ⟨inc'', K''⟩
92        if equiv then
93          ⟨q', ⟨inc', K'⟩⟩ ← ⟨q'', ⟨inc'', K''⟩⟩
94          break
95        end if
96      end if
97    end forall
98    Q_sts ← Q_sts ∪ {⟨q', ⟨inc', K'⟩⟩}
99    →_sts ←→_sts ∪ {⟨q, C⟩ --p→r:a⟨x⃗⟩--> ⟨q', ⟨inc', K'⟩⟩}
100 end if
```

```
101         end forall
102       end forall
103     end forall
104   end while
```

```
1  function BuildHypothesis(input v, input K, input inc, input seen)
2      (* Comienzo con el conocimiento que
3      tengo en el contexto sobre la variable
4      en cuestion *)
5      knowledge ← K(v, inc(v))
6      seen ← seen ∪ {v}
7      (* Recorro todas las variables libres
8      involucradas en el conocimiento y
9      agrego recursivamente al la hipotesis
10     para de todas las variables libres que
11     forman parte del conocimiento *)
12     do
13         forall v' ∈ fv(knowledge)\seen
14             knowledge ←BuildHypothesis(v', K, inc, seen)
15         end forall
16     while knowledge changes
17     return knowledge
```

3. IMPLEMENTACIÓN

En este capítulo desarrollaremos los aspectos relevantes de la implementación del algoritmo introducido en la Sec. 2.3. El capítulo se divide en 7 secciones: 1) Visión General: se da un primer pantallazo de la herramienta desarrollada. 2) Bibliotecas Utilizadas: se presentan y explican las bibliotecas externas utilizadas durante el desarrollo, incluyendo aquellas que fueron desestimadas luego de cambios en la implementación. 3) Interfaz Gráfica: se describe el objetivo de la interfaz gráfica implementada. 4) Parser: se define la gramática utilizada para la codificación del *input*, el uso y su extensibilidad. 5) Lenguaje de Sentencias: esta implementación es paramétrica en el lenguaje de las condiciones sobre los datos. Esta sección explica la implementación de las sentencias, cómo se desarrolló el lenguaje utilizado para los casos de estudios y cómo extenderlo. 6) Estructuras de Datos: aquí se explican las estructuras de datos implementadas para representar los objetos involucrados en el algoritmo. 7) Synchronous Transition System: finalmente se describe la implementación del algoritmo presentado en la Sec. 2.3.

Para los diagramas de clase se va a utilizar UML[1] como formalismo para su representación.

3.1. Visión General

Se va a explicar a continuación una primer idea del funcionamiento de la herramienta desarrollada. Para esto, en la figura 3.1 podemos encontrar un diagrama de alto nivel de todo lo detallado en esta sección.

Para hacer uso de esta herramienta el usuario debe empezar por codificar los participantes involucrados en un archivo de texto plano. El detalle respecto a esta codificación puede encontrarse en las secciones 3.4 y 3.5. Luego, haciendo uso de la interfaz gráfica, carga el archivo y comienza la ejecución.

El primer paso será *parsear* la entrada. Para esto, el `Parser` generado con `CUP` lee el archivo y construye las estructuras de datos correspondientes a los objetos necesarios para la creación de las GCMs.

Una vez construidas todas las máquinas, el `SynchronousTransitionSystem` las utiliza, junto al `SentenceFactory`, para armar el sistema. En este proceso se van creando, a medida que se exploran las máquinas, los `SynchronousTransitionSystemState` con sus respectivos `StateContext`.

Para la construcción de los estados hará falta verificar las condiciones sobre los datos. Para esto en el lenguaje elegido para los casos de estudios se hace uso de un `SMT Solver`: Z3[1].

Al finalizar el proceso se habrán construido tanto las máquinas *parseadas* desde la codificación como el sistema sincrónico. Sus respectivos grafos son representados y visualizados por la interfaz gráfica usando `JGraphT`.

En las próximas secciones se desarrollará en mayor profundidad cada parte de este proceso.

[1] https://es.wikipedia.org/wiki/Lenguaje_unificado_de_modelado

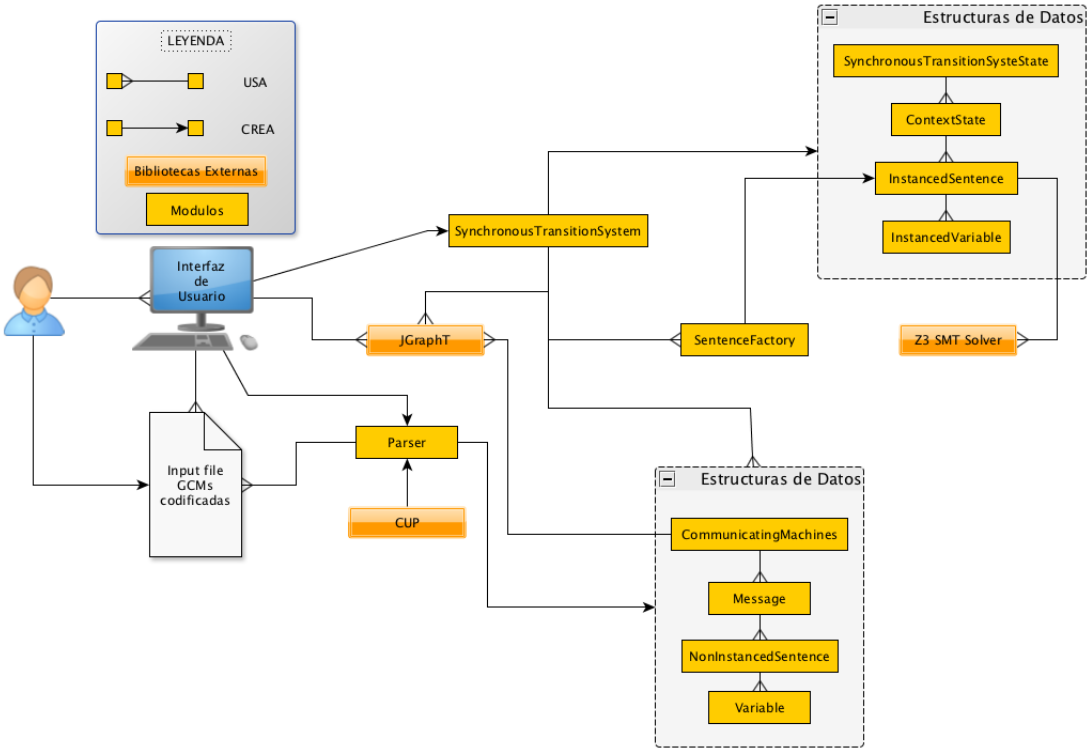

Fig. 3.1: Arquitectura de la herramienta desarrollada

3.2. Bibliotecas Utilizadas

3.2.1. JGraphT y JGraphX / JGraph

Tanto las GCMs como el Sistema Sincrónico son grafos dirigidos y por lo tanto las bibliotecas buscadas tenían que garantizar la correcta implementación de estructuras de datos que permitan manejar grafos y multigrafos direccionales suficientemente grandes, como los obtenidos al armar el Sistema, así como la existencia de métodos que permitan recorrerlos de manera eficiente.

Por otro lado, tuvimos en cuenta la extensibilidad de la biblioteca buscada para incorporar ejes personalizados en la GCMs. Éstos son los mensajes, datos y condiciones enviados en la comunicación.

Por último pero no menos importante, era necesario que los grafos puedan visualizarse para facilitar la implementación de la interfaz gráfica.

Como resultado de esta búsqueda utilizamos la biblioteca **JGraphT** [2] que incorpora la biblioteca **JGraphX/JGraph** [3].

JGraphT: Esta biblioteca está enfocada en la estructura de datos y los algortimos de grafos.

JGraph/JGraphX: Esta biblioteca está enfocada en el renderizado y visualización (modificable) de grafos. Conocida como JGraph, la última versión modifico su nombre a JGraphX ya que su código fue reescrito completamente.

[2] http://jgrapht.org/
[3] https://github.com/jgraph/jgraphx

3.2.2. CUP

Para implementar el parser y la gramática del *input* del algoritmo (i.e. la codificación de GCMs) elegimos una biblioteca que nos facilitará la construcción de un parser para Java partiendo de la definición de una gramática de manera sencilla.

CUP [4] (Construction of Useful Parser) es un generador de parsers LALR [5] para Java. Definiendo la gramática y su funcionamiento en un archivo `.cup` la biblioteca se encarga de generar la clases `Parser` y `sym`, siendo la primera el parser (que requiere la implementación de un `scanner`) y la segunda la definición de los símbolos terminales.

3.2.3. Z3

En nuestros casos de estudio usamos como lenguaje lógica proposicional con aritmética para expresar las condiciones sobre los datos de las comunicaciones entre las GCMs. Como se vio en el algoritmo presentado anteriormente, es necesario garantizar un mecanismo de decisión de compatibilidad entre contratos. Para este caso utilizamos un SMT Solver [6].

En un comienzo hicimos el desarrollo usando un *SAT Solver*[4], **SAT4j** [7], pero al incorporar aritmética en el lenguaje la opción natural paso a ser la utilización de un *SMT Solver* haciéndose necesario buscar otra biblioteca.

Z3 es el demostrador de teoremas elegido.

3.2.4. Cloning y Objenesis

La complejidad y la necesidad de construir por copia algunas de las estructuras nos llevó a usar **Cloning** y **Objenesis** [8] para clonar objetos.

Por cuestiones de rendimiento finalmente no se usan en la versión final. En el capítulo de casos de estudio y resultados puede encontrarse más detalle sobre esto.

3.3. Interfaz Gráfica

La interfaz gráfica desarrollada para esta tesis cumple el objetivo de sentar pisos de necesidades que debe cumplir una interfaz para el análisis del algoritmo. Estas son:

- Input de GCMs codificadas usando archivos de texto.
- Selección del lenguaje utilizado para expresar las condiciones sobre los datos de las comunicaciones de las GCMs.
- Visualización de las GCMs parseadas.
- Visualización de las condiciones de las comunicaciones de las GCMs.
- Visualización del sistema sincrónico generado.
- Visualización del contexto de cada estado del sistema sincrónico.

[4] http://www2.cs.tum.edu/projects/cup/
[5] https://en.wikipedia.org/wiki/LALR_parser
[6] https://en.wikipedia.org/wiki/Satisfiability_modulo_theories
[7] http://www.sat4j.org/
[8] https://github.com/kostaskougios/cloning

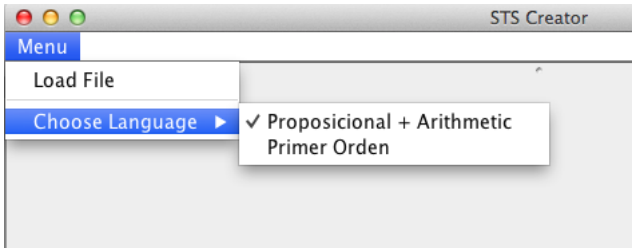

Fig. 3.2: GUI: Selección de lenguaje e input

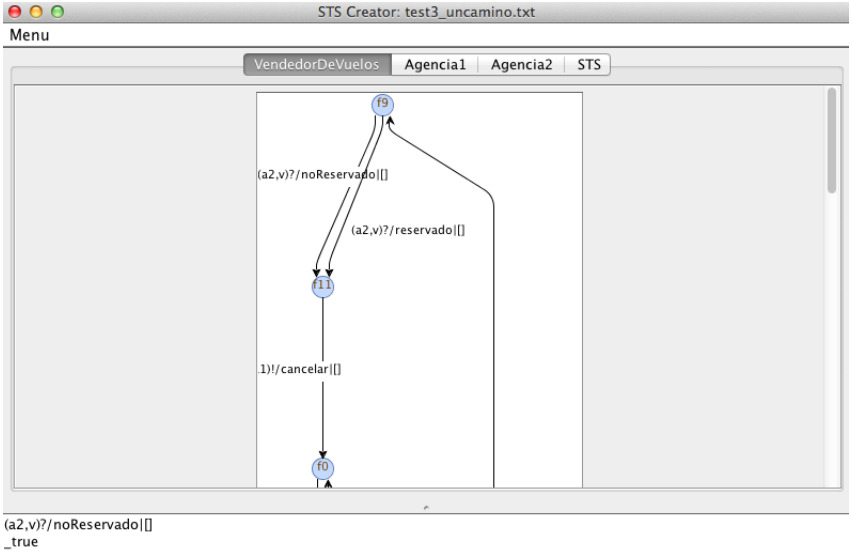

Fig. 3.3: GUI: Visualización de GCM y condiciones de la comunicación

La versión de la herramienta que se presenta en este trabajo se implementó usando la biblioteca Swing de Java que comprende los componentes básicos multiplataforma para crear y manipular ventanas. De esta forma se pudo aprovechar la buena integración existente entre JGraphX y Swing.

En las imagenes 3.2, 3.3 y 3.4 puede verse la interfaz de la aplicación en ejecución cumpliendo los objetivos propuestos.

El desarrollo de una interfaz más agradable, flexible y útil excede los alcances de esta tesis, sin embargo cumple satisfactoriamente con los objetivos propuestos. Las mejoras pensadas para la interfaz gráfica pueden encontrarse en la sección de trabajo futuro.

3.4. Parser

Como se mencionó anteriormente, el *input* para la decisión de compatibilidad y la consecuente construcción del sistema sincrónico es un archivo de texto con las GCMs codificadas. A continuación se encuentra la gramática y el uso del parser.

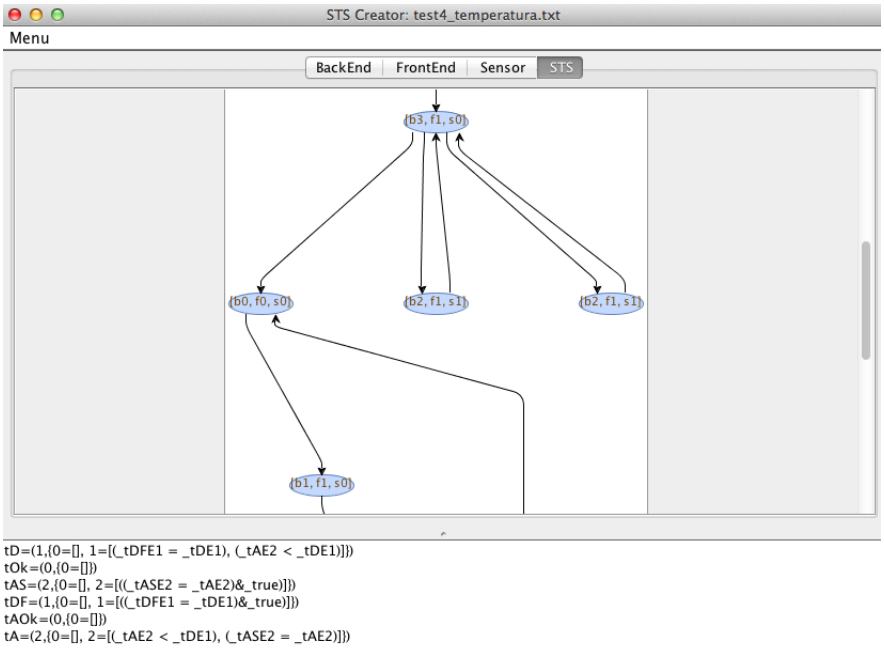

Fig. 3.4: GUI: Visualización del sistema sincrónico y el contexto de un estado

3.4.1. Gramática

$$
\begin{aligned}
\langle \text{machine} \rangle &\models \langle \text{machine} \rangle \langle \text{id} \rangle \, , \langle \text{id} \rangle \, (\, \langle \text{id} \rangle \,) \, \{ \, \langle \text{trans} \rangle \, \} \;\mid\; \lambda \\
\langle \text{trans} \rangle &\models \langle \text{id} \rangle \, \langle \text{msg} \rangle \, \langle \text{id} \rangle \, ; \langle \text{trans} \rangle \;\mid\; \lambda \\
\langle \text{msg} \rangle &\models \langle \text{id} \rangle : \langle \text{id} \rangle \, ! \, \langle \text{id} \rangle < \langle \text{vars} \rangle > \{ \, \langle \text{sentence} \rangle \, \} \\
&\quad \mid \langle \text{id} \rangle : \langle \text{id} \rangle \, ? \, \langle \text{id} \rangle \, (\, \langle \text{vars} \rangle \,) \, \{ \, \langle \text{sentence} \rangle \, \} \\
\langle \text{vars} \rangle &\models \langle \text{id} \rangle : \langle \text{id} \rangle \;\mid\; \langle \text{id} \rangle : \langle \text{id} \rangle \, , \langle \text{vars} \rangle \;\mid\; \lambda \\
\langle \text{id} \rangle &\models \text{a} \ldots \text{z} \;\mid\; \text{A} \ldots \text{Z} \;\mid\; \text{a} \ldots \text{z} \langle \text{id} \rangle \;\mid\; \text{A} \ldots \text{Z} \langle \text{id} \rangle
\end{aligned}
$$

La gramática presentada permite codificar GCMs comenzando por definir un nombre, un identificador y el estado inicial. Luego se definen los estados y sus trancisiones.

Cada línea de las transiciones define un estado origen, la transición y el estado destino.

Las transiciones se definen con el nombre del mensaje, las máquinas involucradas, si es de envío (!) o de recepción (?), las variables que se envían o reciben y, por último, las condiciones sobre los datos.

En la figura 3.5 puede verse un ejemplo de *input* de dos máquinas.

La gramática no establece *a priori* un lenguaje para definir las condiciones de los contratos (*sentence*). Estas producciones deben redefinirse para cada lenguaje que decidamos usar. Es decir que debemos construir un *parser* específico para cada lenguaje.

3.4.2. Uso

Una vez definida la gramática, sus producciones y sus símbolos terminales, es necesario establecer el funcionamiento del *parser*. Para ello es necesaria la implementación

```
maquinaA,A(a0) {
    a0 msg1:A!B<x_1,..,x_n>{sentence} a1;
    a1 msg2:B?A(y_1,..,y_n){sentence} a0;
}
maquinaB,B(b0) {
    b0 msg1:A?B(x_1,..,x_n){sentence} b1;
    b1 msg2:B!A<y_1,..,y_n>{sentence} b0;
}
```

Fig. 3.5: Ejemplo de codificación de GCMs

de un `Scanner` para la lectura de los símbolos. La biblioteca utilizada produce una clase `sym.class` con la definición de todos los símbolos terminales que hayamos definido.

Luego, es nuestra responsabilidad definir, para cada producción, los objetos y estructuras resultantes del *parseo*.

$$
\begin{aligned}
\langle \text{machine} \rangle &\models \textit{List<CommunicatingMachine>} \\
\langle \text{trans} \rangle &\models \textit{List<Truple<String, Message, String>>} \\
\langle \text{msg} \rangle &\models \textit{Message} \\
\langle \text{vars} \rangle &\models \textit{Vector<Variable>} \\
\langle \text{id} \rangle &\models \textit{String} \\
\langle \text{sentence} \rangle &\models \textit{NonInstancedSentence}
\end{aligned}
$$

3.4.3. Lógica Proposicional con Aritmética

Para nuestros casos de estudio vamos a utilizar lógica proposicional con aritmética como lenguaje para las condiciones sobre los datos (sentencias). Definimos entonces las siguientes producciones para las sentencias.

$$
\begin{aligned}
\langle \text{sentence} \rangle &\models \langle \text{form} \rangle \mid \lambda \\
\langle \text{form} \rangle &\models \langle \text{id} \rangle \mid (\langle \text{form} \rangle \mid \langle \text{form} \rangle) \mid (\langle \text{form} \rangle \,\&\, \langle \text{form} \rangle) \\
&\quad \mid (\langle \text{form} \rangle <-> \langle \text{form} \rangle) \\
&\quad \mid (\sim \langle \text{form} \rangle) \mid \langle \text{aritmeticform} \rangle \\
\langle \text{aritmeticform} \rangle &\models (\langle \text{aritmetic} \rangle < \langle \text{aritmetic} \rangle) \\
&\quad \mid (\langle \text{aritmetic} \rangle > \langle \text{aritmetic} \rangle) \\
&\quad \mid (\langle \text{aritmetic} \rangle = \langle \text{aritmetic} \rangle) \\
&\quad \mid (\langle \text{aritmetic} \rangle >= \langle \text{aritmetic} \rangle) \\
&\quad \mid (\langle \text{aritmetic} \rangle <= \langle \text{aritmetic} \rangle) \\
\langle \text{aritmetic} \rangle &\models \langle \text{id} \rangle \mid (\langle \text{aritmetic} \rangle + \langle \text{aritmetic} \rangle) \\
&\quad \mid (\langle \text{aritmetic} \rangle - \langle \text{aritmetic} \rangle) \\
&\quad \mid (\langle \text{aritmetic} \rangle * \langle \text{aritmetic} \rangle)
\end{aligned}
$$

Y sus respectivos objetos y estructuras resultantes.

$$\begin{aligned} \langle \text{sentence} \rangle &\models \textit{NonInstancedSentence} \\ \langle \text{form} \rangle &\models \textit{PropSentence} \\ \langle \text{aritmeticform} \rangle &\models \textit{PropSentence} \\ \langle \text{aritmetic} \rangle &\models \textit{AritmeticSentence} \end{aligned}$$

En la Fig. 3.6 puede verse un ejemplo de estas sentencias.

```
((3+5)<(2+n)) | ((~x) & (y <-> z))

Con n : int
Con x, y, z : bool
```

Fig. 3.6: Ejemplo de sentencia proposicional con aritmética

3.5. Lenguaje de Sentencias

En esta sección vamos a ver cómo se representan los lenguajes de sentencias (i.e. el lenguaje usado para las condiciones sobre los datos), qué estructuras intervienen y cuáles son necesarias implementar para extender los lenguajes aceptados.

Por último, se explica cómo se implementó el lenguaje de lógica proposicional con aritmética que fue el elegido para los estudios, a modo de ejemplificar cómo llevar a cabo la extensibilidad.

3.5.1. Variables y Sentencias

El envío y recepción de datos en los mensajes de las GCMs está representado por la clase `Variable`. Estas variables se instancian al momento de construir el sistema sincrónico, representando las distintas encarnaciones y generando el conocimiento de cada estado. Su estructura puede encontrarse en la figura 3.7.

El valor que tomarán estas variables será definido por el lenguaje que se elija.

La implementación del algoritmo para construir el sistema desde un comienzo iba a ser paramétrico en el lenguaje usado para expresar las condiciones sobre los datos. Para esto se diseñaron e implementaron las interfaces correspondientes a las Sentencias No Instanciadas y las Sentencias Instanciadas.

Las No Instanciadas son la representación de las condiciones de la GCMs, cuyas variables no se encuentran instanciadas, mientras que las Sentencias Instanciadas se crean a partir de las primeras, encarnando las variables, es decir instanciándolas.

Ambas interfaces pueden encontrarse en la figura 3.8.

Hay 3 métodos que ambas interfaces comparten: `freeVariables()`, `freeVariablesNames()` y `conjuntion(other)`. Los dos primeros retornan las variables libres o sus nombres respectivamente. `conjuntion` retorna la conjunción entre dos sentencias.

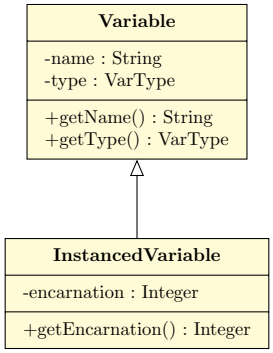

Fig. 3.7: Diagrama de clases de Variable

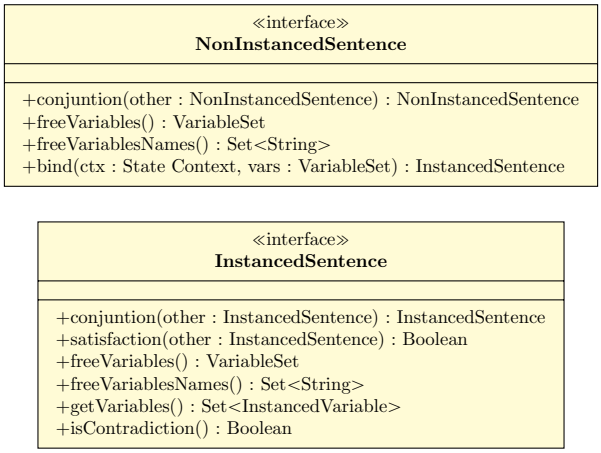

Fig. 3.8: Interfaces de Non Instanced Sentence y Instanced Sentence

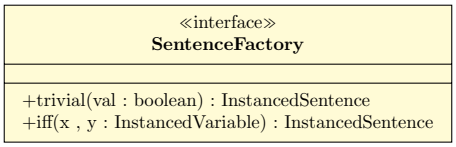

Fig. 3.9: Interfaz de Sentence Factory

Las sentencias no instanciadas cuentan con el método `bind` que retorna una sentencia instanciada usando el contexto pasado como parámetro para instanciar las variables libres. Los contextos (serán vistos más adelante) representan el conocimiento adquirido hasta el momento. Por otra parte, las sentencias instanciadas tienen 3 nuevos métodos: `copy`, `satisfaction` y `isContradiction`. `copy` devuelve una copia de la sentencia. Veamos los otros 2.

`satisfaction` será el encargado de demostrar si una sentencia (hipótesis) infiere otra sentencia. `isContradiction`, como bien podemos suponer, verificará si la sentencia es una contradicción o no.

Cualquier lenguaje que decidamos incorporar al sistema debe contar con todos estos métodos.

3.5.2. Sentence Factory

La construcción de sentencias se realiza mayoritariamente al momento del *parseo* de las GCMs, esto hace que se delegue la responsabilidad a la gramática y el parser. Sin embargo, se hace necesaria la construcción de sentencias triviales correspondientes al lenguaje elegido para las condiciones de los datos. Para esto se implementó la interfaz `SentenceFactory`.

La sentencias triviales son 2:

- True/False: Necesaria para la construcción de sentencias a partir de hipótesis nulas.
- X $\leftrightarrow$ Y: Utilizada para igualar variables de envío y recepción.

En la figura 3.9 se presenta dicha interfaz.

3.5.3. Extensibilidad

Lo visto hasta ahora concluye los requisitos para poder extender el algoritmo a nuevos lenguajes. Enumerando, las clases y estructuras a implementar son:

- *Parser*: Extender la gramática de sentencias para el nuevo lenguaje, teniendo en cuenta las estructuras que el mismo irá construyendo.
- Implementar `NonInstancedSentence`: Será la implementación del lenguaje de condiciones para las GCMs.
- Implementar `InstancedSentence`: Será la implementación del lenguaje que servirá para generar el conocimiento (*knowledge*) del sistema sincrónico.
- Implementar `SentenceFactory`: Clase encargada de construir las sentencias triviales del lenguaje elegido.

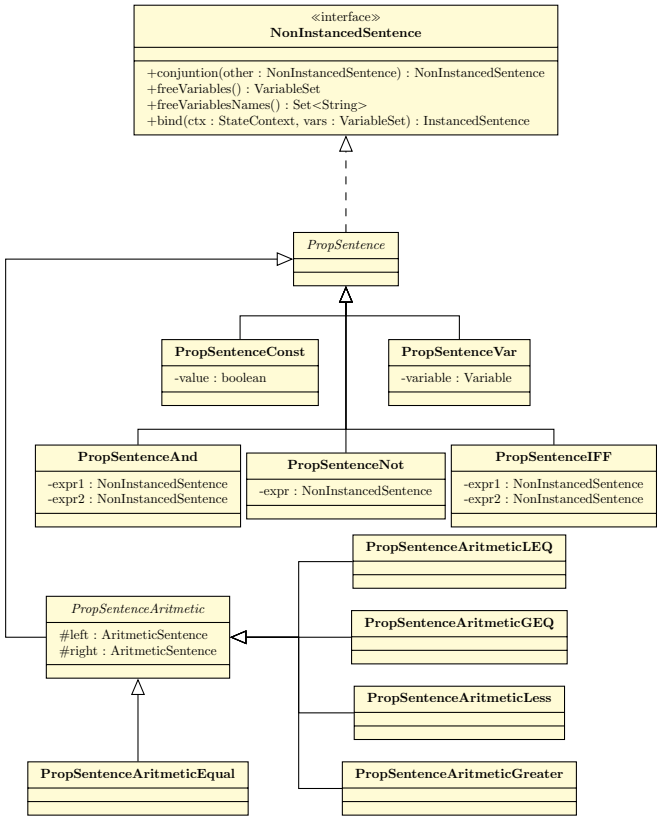

Fig. 3.10: Diagrama de clases de Non Instanced Sentences para proposicional + aritmética

- Extensión del `VarType`: En caso de ser necesario, se deberá extender el tipo de variables permitidas.

En la siguiente sección se puede encontrar el ejemplo del lenguaje elegido para los casos de estudio.

3.5.4. Lógica Proposicional con Aritmética y Solver

En un comienzo, el lenguaje elegido fue lógica proposicional. Luego de algunas pruebas y para aumentar la potencialidad del lenguaje se decidió agregarle aritmética. Anteriormente se vio la gramática pensada para este lenguaje.

En las figuras 3.10 y 3.11 puede encontrarse el diagrama de clases correspondiente a las sentencias booleanas mientras que en las figuras 3.12 y 3.13 se muestran los diagramas de clases de las sentencias aritméticas implementadas.

Para estas implementaciones, dada la composición recursiva y estructura de árbol de las sentencias, se utilizó el patrón *Composite*[9].

También, como se enumeró en los pasos necesarios para incorporar nuevos lenguajes, se implementó la `SentenceFactory` de proposicional. En la figura 3.14 puede verse el diagrama de clases correspondiente.

Para el mecanismo de decisión de validez de las sentencias se utilizaron bibliotecas externas (SAT4j en un comienzo y Z3 finalmente). Para poder hacer uso de estas bibliotecas fue necesario hacer una conversión de nuestras estructuras de lógica proposicional con

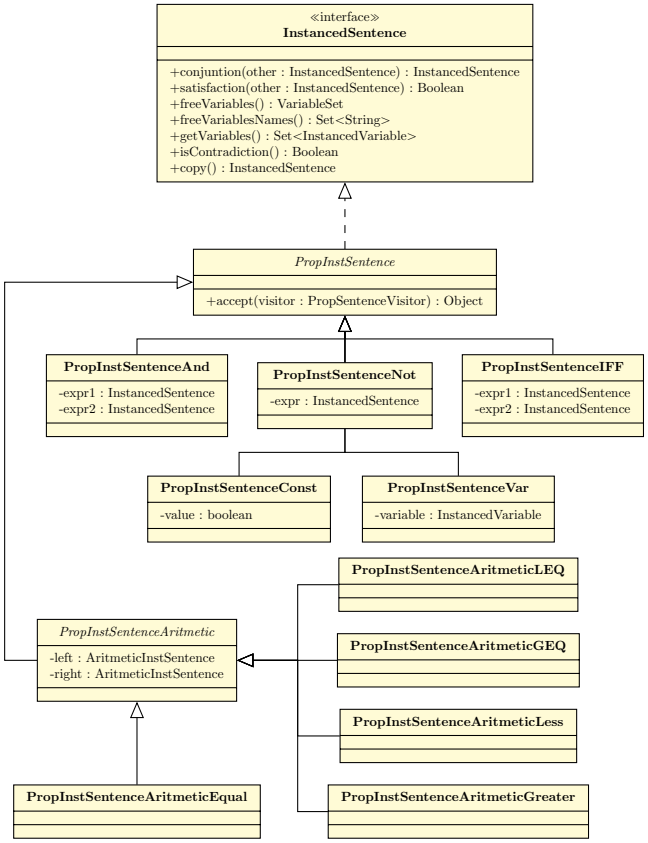

Fig. 3.11: Diagrama de clases de Instanced Sentences para proposicional + aritmética

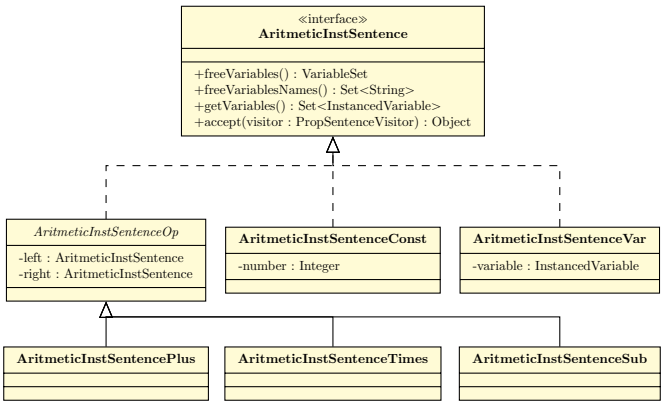

Fig. 3.12: Diagrama de clases de Instanced Aritmetic Sentence

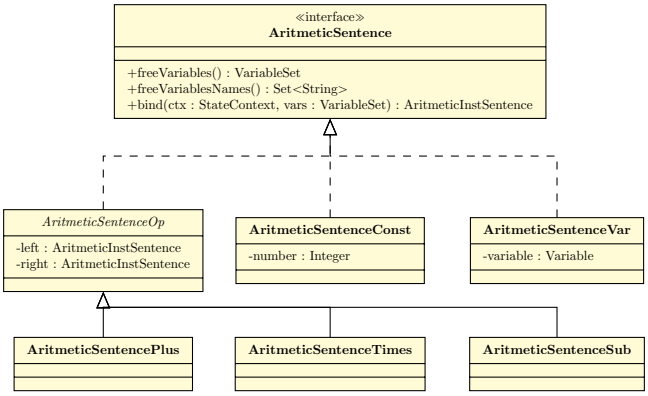

Fig. 3.13: Diagrama de clases de Aritmetic Sentence

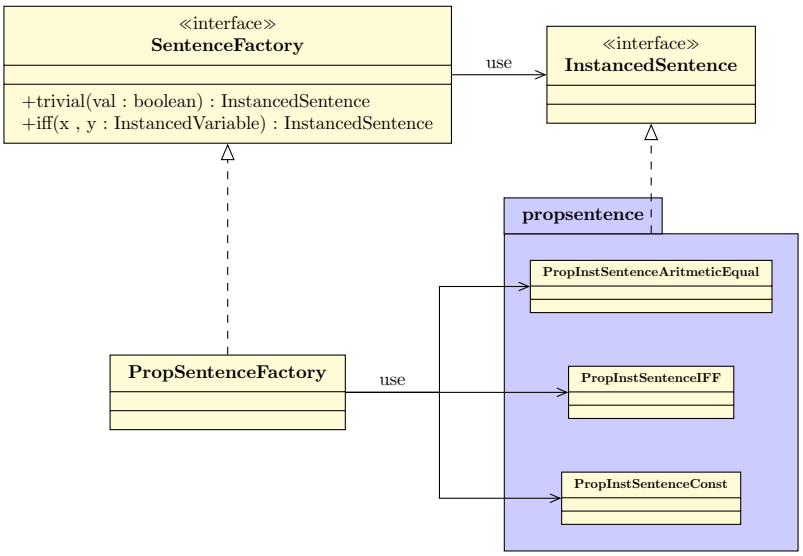

Fig. 3.14: Diagrama de clases de Sentence Factory

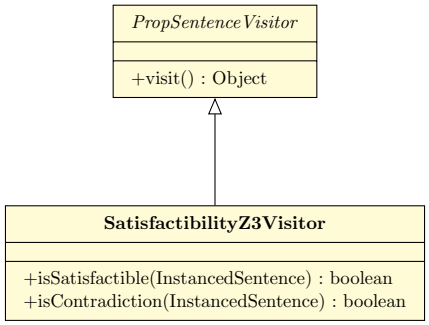

Fig. 3.15: Diagrama de clases Visitor

aritmétia a las estructuras usadas por esas bibliotecas. Aprovechando el patrón de diseño usado se implementó un *visitor* encargado de realizar la transformación. En la figura 3.15 puede encontrarse el diseño del mismo.

El `SatisfactibilityZ3visitor` implementa los métodos `isContradiction` y `isSatisfactible`.

Al momento de corroborar la validez o contradicción de una sentencia, se usan los métodos `satisfaction` y `isContradiction` de las `InstancedSentence`. Estos métodos, en el caso de proposicional crean un `SatisfactibilityZ3visitor` y utilizan los métodos `isContradiction` y `isSatisfactible` del visitor. Estos métodos se encargan de construir los objetos necesarios a partir de las sentencias (visitándolas estructuralmente) para luego hacer uso del solver y retornar si son o no satisfacibles.

3.6. Estructuras de Datos

En esta sección se detallan los distintos objetos que fue necesario implementar y qué decisiones se tomaron para sus estructuras. Encontraremos aquí estructuras *ad-hoc* que resuelven falencias de otras pre-existentes y la representación de los mensajes, las GCMs, los contextos y los estados del Sistema Sincrónico.

3.6.1. Tuplas y VariableSet

Algunas estructuras que fue necesario implementar fueron las 2-tuplas, 3-tuplas y el `VariableSet`. Las primeras 2 se eligieron para modularizar la representación de algunos datos que resultaba conveniente manejar en conjunto. El `VariableSet` se trata de un conjunto de variables con algunas particularidades.

La creación de un `VariableSet` surge a partir de las limitaciones que encontramos al usar un `Set<Variable>`, ya sea por el uso de variables instanciadas (encarnaciones distintas serían variables distintas y se requería que fueran tomadas como iguales) o bien por variables de igual nombre que eran instancias distintas. Para solucionar esto optamos por implementar un `VariableSet` cuya clase se puede ver en la figura 3.16

Como se ve, los métodos son muy similares a los métodos de `Set`.

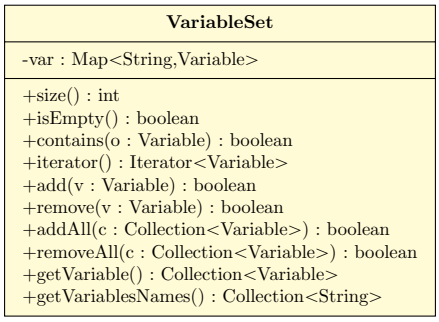

Fig. 3.16: Clase VariableSet

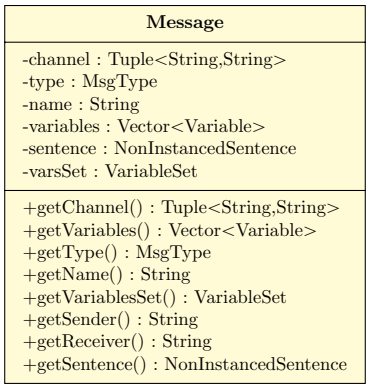

Fig. 3.17: Clase Message

3.6.2. Mensajes

La clase `Message` representa los mensajes de las GCMs. Para esto es necesario considerar el canal del mensaje (máquinas involucradas), el tipo de mensaje (envío o recepción), los datos (variables) enviados/recibidos y las condiciones sobre los datos.

En su estructura (Fig. 3.17) esta clase cuenta con un `Vector<Variable>` para almacenar las variables correspondientes a los datos enviados o recibidos. Al momento de decidir la validez de las condiciones de los datos entre los mensajes de un canal, las variables de los mensajes de cada GCM se relacionan posición a posición: $X_1 \leftrightarrow Y_1, ..., X_n \leftrightarrow Y_n$, siendo X el vector de datos del mensaje de envío e Y el vector de datos del mensaje recibido.

También su representación interna tiene un `VariableSet` para almacenar las mismas variables. La utilidad de esto es contar con otra estructura que contenga las variables y permita verificar la existencia de una variable de una manera más eficiente.

3.6.3. Communicating Machines (GCMs)

La representación interna de las *Guarded Communicating Machines* debe garantizar:

- Almacenamiento de datos como el identificador y nombre de la máquina y el estado inicial.
- Fácil acceso a todas las variables involucradas en la máquina para evitar conflictos

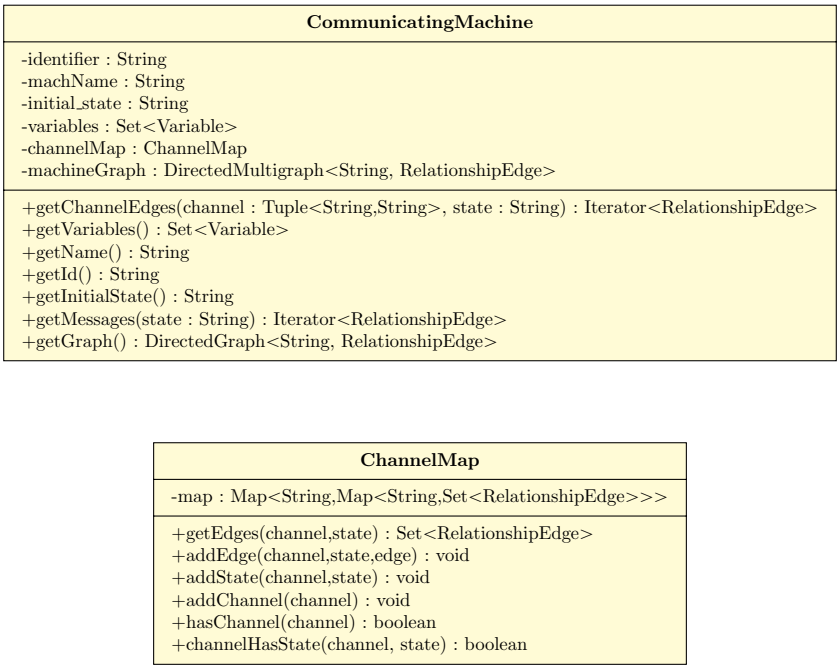

Fig. 3.18: Clase Communicating Machine y ChannelMap

por nombres de variables entre máquinas al momento de construir el conocimiento de los estados del sistema sincrónico.

- Iteración eficiente de los ejes según el estado y canal involucrado.
- Representación de la máquina de estado completa usando un multigrafo que permita visualizarla

Para cumplir con esos objetivos, la estructura elegida es la vista en la figura 3.18.

Como vemos, la estructura guarda la información sobre la máquina, el grafo correspondiente y mantiene el conjunto de variables que nos permite conocer todas las variables involucradas en la máquina. Para el acceso eficiente a los mensajes se usa un `ChannelMap`.

La clase `ChannelMap` permite manipular los mensajes de la máquina pudiendo acceder a ellos de manera rápida según el canal y el estado que querramos buscar.

Las instancias de estas clases son construidas por el `parser` al *parsear* el *input* con la codificación de las GCMs. En ese momento se realizan algunas verificaciones para corroborar que la máquina sea consistente.

- La máquina creada pertenece al canal de todos los mensajes.
- Los canales de los mensajes son consistentes con el tipo de mensajes (send/receive).
- El estado inicial propuesto es parte de los estados de la máquina.
- No hay nombres de variables repetidos.
- Un estado sólo puede tener envíos o recepciones. No pueden salir ambos tipos de mensajes.

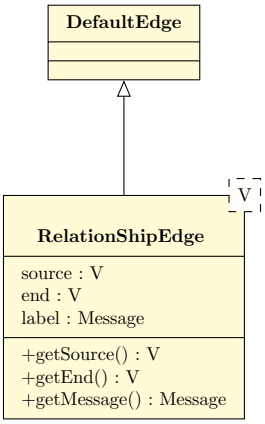

Fig. 3.19: Clase RelationShipEdge

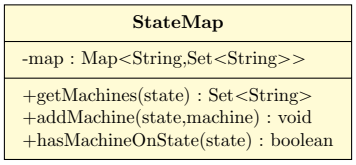

Fig. 3.20: Clase StateMap

- No hay estados inalcanzables.

Si alguno de esos puntos no llegara a cumplirse, la máquina en cuestión no puede ser generada y se produce un error. También se identifican estados *sink*, estados sin trancisiones de salida. A priori no son considerados perjudiciales para el algoritmo.

El multigrafo y su visualización se logra gracias a la utilización de la biblioteca JGraphT vista anteriormente.

Por último, para la correcta representación de las máquinas hizo falta definir un nuevo tipo de eje para el grafo que permita la inclusión de mensajes en las transiciones entre estados. Para esto extendimos la clase `DefaultEdge` con `RelationShipEdge` quien cuenta entre sus miembros con el campo `label` para almacenar el mensaje de la transición. El resto puede verse en la figura 3.19.

En una segunda versión de la implementación, se agregó la estructura `StateMap` cuyo diagrama está en la figura 3.20. Esto sucedió luego de analizar los resultados de los casos de estudio y ver que podía mejorarse el rendimiento. Esta estructura tiene como función conocer rápidamente las máquinas con las que puedo comunicarme en un estado dado de la Communicating Machine. Para más información sobre el uso y razón de esta estructura ver el capítulo de *Casos de Uso y Resultados*.

3.6.4. State Context

El `StateContext` es el contexto de cada estado del sistema sincrónico, esto quiere decir que aquí se encuentra toda la información generada para cada estado: las variables involucradas, la encarnación de cada una y el conocimiento (*knowledge*) para cada encarnación

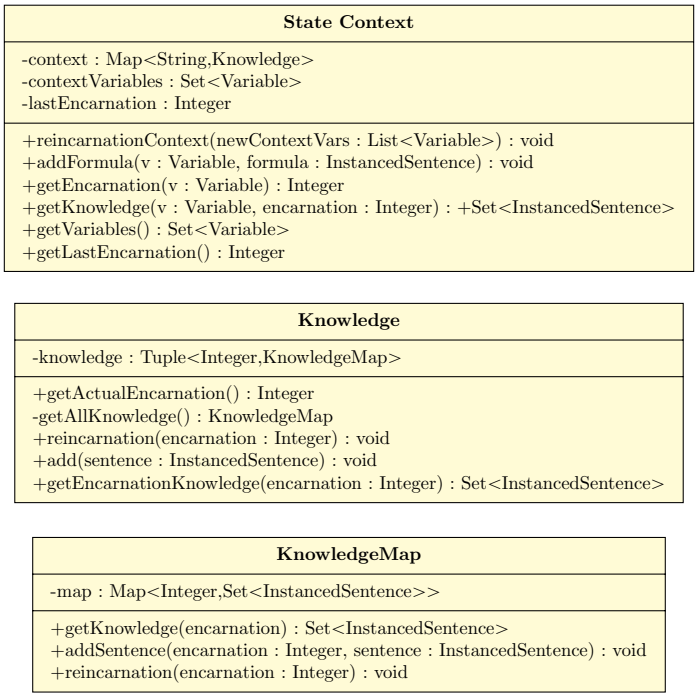

Fig. 3.21: Diagrama de clases de State Context

de variable. Es una pieza fundamental del algoritmo ya que no solo mantiene el conocimiento alcanzado hasta el momento sino que implementa la reencarnación de las variables con su consecuente adquisición de conocimiento. Las sentencias, que representan el conocimiento, aquí se encuentran instanciadas, esto quiere decir que el conocimiento habla de variables encarnadas. La clases involucradas se observan en la figura 3.21. Veámoslo más en detalle.

El conocimiento se acumula en el campo `context`. Se trata de un `Map` que contiene el `knowledge` de cada variable del sistema sincrónico. Al momento de creación del sistema se construye el contexto con todas las variables de todas las máquinas involucradas. En un comienzo las variables no habrán sido instanciadas (su encarnación será cero) y el conocimiento será vacío.

La manipulación del conocimiento se encapsuló en la clase `Knowledge`. La estructura interna de esta clase es una tupla con el valor de la última encarnación de la cual habla la instancia y un `KnowledgeMap` con el conocimiento acumulado para cada encarnación de la variable en cuestión.

El `KnowledgeMap` se representa con un `Map` que mantiene para cada encarnación el conjunto de `InstancedSentence` correspondientes al conocimiento.

Para la reencarnación del contexto solo basta con informar las variables que deben reencarnarse para agregar una nueva encarnación de esas variables, manteniendo todo el conocimiento que hayan generado hasta el momento y pudiendo agregar nuevo conocimiento a estas encarnaciones. Esta situación se produce al moverse por un mensaje de un canal: se reencarnan las variables correspondientes a los datos recibidos/enviados.

Al recibir el mensaje de reencarnación con la lista de variables a reencarnar, el `StateContext` envía el mensaje de `reincarnation` al `Knowledge` de cada variable. El

`Knowledge` actualiza el valor de la encarnación y envía el mensaje de `reincarnation` al `KnowledgeMap` para que almacene el conocimiento de esta nueva encarnación, que a priori será el mismo que la encarnación anterior y luego le agregara las nuevas sentencias en caso de estar involucrada esa variable. Para mayor claridad en este punto, en la figura 3.22 está parte del pseudocódigo visto en la sección 2.3 donde puede verse como se generan los nuevos conocimientos.

$$K'(v,i) \leftarrow \begin{cases} x_n^{inc'(x_n)} = y_n^{inc'(y_n)} \wedge X' & \text{if } v = x_n \wedge i = inc'(v) \\ x_n^{inc'(x_n)} = y_n^{inc'(y_n)} & \text{if } v = y_n \wedge i = inc'(v) \\ K_{\mathsf{C}}(v,i) \wedge X' & \text{if } v \in \overrightarrow{\chi} \wedge i = inc'(v) \\ K_{\mathsf{C}}(v,i) & \text{oterwhise} \end{cases}$$

Fig. 3.22: Construcción del nuevo conocimiento

Se incorporan también los métodos necesarios para extraer información importante como el conocimiento de la encarnación de una variable o el último valor de encarnación. También los métodos para agregar conocimiento a variables puntuales. Estas nuevas sentencias se agregan a la última encarnación de la variable en cuestión.

3.6.5. Synchronous Transition System State

Esta clase representa el estado del sistema sincrónico propiamente dicho. En su interior se encuentra el `StateContext` visto anteriormente y la lista de estados en la que se encuentra cada una de las máquinas involucradas en el sistema.

A su vez, se delega en esta clase la operación de construcción de un nuevo estado a partir de una transición desde el estado actual. Esta construcción implica: el cambio en la lista de estados y la generación del nuevo contexto.

El cambio de estado se produce ya que al moverse por un mensaje las dos máquinas involucradas (recordemos que los canales son binarios, es decir que involucran a dos participantes exclusivamente) cambian de estado, por lo tanto el `STSState` armará su lista de estados representando esta situación.

La creación del nuevo contexto recibe como parámetros las variables a reencarnar y el nuevo conocimiento conseguido por realizar una transición (enviar/recibir un mensaje). Usará el método de reencarnación del `StateContext` informando las variables reencarnadas y luego agregará el nuevo conocimiento al nuevo contexto generado.

La figura 3.23 muestra esta clase.

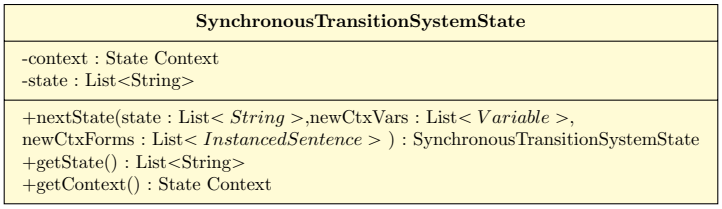

Fig. 3.23: Clase `SynchronousTransitionSystemState`

3.7. Synchronous Transition System

Llegamos finalmente a la clase encargada de construir el sistema sincrónico, es decir, de implementar el algoritmo. La entrada para la construcción del sistema es la lista de GCMs involucradas y el `SentenceFactory` correspondiente al lenguaje que se esté utilizando para las sentencias.

Antes de comenzar, corrobora que no haya variables repetidas entre las máquinas para evitar conflictos al ir armando el conocimiento de los estados del STS. Luego guarda la lista de GCMs y crea el `STSState` inicial con la lista de estados de las máquinas en el mismo orden que la lista de GCMs y con los estados iniciales de cada máquina. Al crear el `STSState` inicial el contexto contiene variables no encarnadas (encarnación en 0) y sin conocimiento.

La implementación general del algoritmo es bastante similar al pseudocódigo visto en la Sec. 2.3 :

1. Toma un `STSState` del conjunto de estados del STS a explorar (en un principio solo contiene al inicial).

2. Recorre las máquinas tomándolas de a pares y observa el estado en el que se encuentra cada una.

3. Se fija si existen mensajes desde el estado de cada máquina que involucre a ese canal (es decir, a esas dos máquinas)

4. De haberlos, los recorre tomándolos de a pares y busca que haya mensajes de igual nombre y compatibles: que uno envíe y otro reciba.

5. Toca el turno de revisar que las las condiciones sobre los datos (sentencias) tienen todas sus variables instanciadas o bien son parte de los datos a enviar/recibir. (ver método `unboundVariables`)

6. Si todo continua satisfactoriamente, se *bindean* las condiciones de los datos usando el conocimiento adquirido hasta el momento (contexto del `STSState` actual), generando sentencias instanciadas.

7. Se arma el *posible* nuevo conocimiento: la relación entre las variables enviadas y recibidas, y la condición instanciada del envío.

8. Se crea el nuevo `STSState`: cambian los estados de las dos máquinas involucradas, se informan las variables que participan en la comunicación y el nuevo conocimiento generado para cada una de ellas. También se le agrega el conocimiento a las variables involucradas en las condiciones pero que no son parte del envío/recepción.

9. Ahora se pasa a decidir la compatibilidad de las condiciones. Se construye la hipótesis: se hace la conjunción el conocimiento de todas las variables involucradas en el envío, en las condiciones del transmisor y en las condiciones del receptor. (ver método `buildHypothesis`)

10. Una vez construida la hipótesis, se corrobora que sea consistente (que no sea una contradicción) y luego vemos si podemos inferir las condiciones de recepción a partir de la hipótesis.

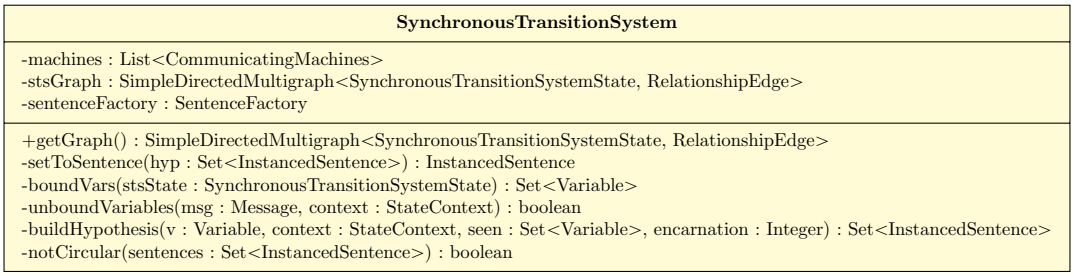

Fig. 3.24: Clase `SynchronousTransitionSystem`

11. En caso afirmativo, basta observar que no exista ya otro `STSState` equivalente: que los estados de las máquinas sean iguales y el conocimiento para cada una de las variables del sistema sea equivalente. (ver método `boundVars` y `contextsAreEquivalent`)

12. Llegado este punto solo falta agregar el nuevo `STSState` al sistema sincrónico (si no hubo otro equivalente) y la correspondiente transición entre el estado que estábamos revisando y el nuevo estado creado (o su equivalente). Por último, se agrega este estado a la lista de estados del STS a explorar.

Un dato a remarcar es que al realizar esta serie de pasos para un `STSState` no hace falta volver a explorarlo, ya que el hecho de agregar el nuevo estado STS al sistema no cambia la situación del resto de los estados. En otras palabras, la exploración de cada estado del STS depende únicamente de si mismo: el estado actual de cada máquina y el conocimiento adquirido hasta ese momento. Esto agiliza la exploración de los estados del STS para la construcción del sistema ya que solo se explora 1 vez cada estado nuevo.

Algunos puntos de esta implementación fueron modificados luego de analizar los resultados de los casos de estudio. En el capítulo correspondiente se detallan estos cambios. La clase puede verse en la figura 3.24.

3.7.1. Bound y Unbound Variables

En el transcurso del algoritmo se hace necesario identificar qué variables del conocimiento están instanciadas y cuales todavía no fueron instanciadas. Esto lo cumple el método `boundVars`.

Este método recibe el `STSState` que querramos revisar y devuelve cuales variables del conocimiento ya fueron instanciadas, es decir, su encarnación es mayor que cero. Esta revisión permite hacer un primer chequeo entre la equivalencia de conocimiento entre dos estados del STS distintos: si las variables instanciadas no son las mismas, es decir, existe al menos una variable que en un estado está instanciada y en el otro no, entonces no pueden ser conocimientos equivalentes.

Por otra parte, el método `unboundVariables` sirve para revisar que todas las variables libres de un mensaje están instanciadas en un contexto dado, osea están instanciadas por el contexto. Este método sirve al momento de corroborar si un mensaje puede ser enviado en un cierto momento (contexto): eso sólo puede suceder si todas las variables involucradas en ese mensaje están instanciadas en ese momento.

3.7.2. Build Hypothesis

La decisión de si un mensaje puede ser enviado tiene como momento crucial la compatibilidad entre los contratos de las 2 máquinas involucradas. Para tomar la decisión es necesario asegurar que con el conocimiento adquirido hasta el momento se puede implicar las condiciones de la recepción del mensaje.

Sobre la implicación, ya vimos que depende del lenguaje y la clase `Sentence` es quien se ocupa de eso. Veamos ahora el paso previo: la acumulación del conocimiento adquirido hasta ese momento, la hipótesis.

El objetivo es acumular todo el conocimiento adquirido de todas las variables involucradas en el mensaje y de todas las variables involucradas en ese conocimiento. Algo así como la clausura del conocimiento. Como vemos, el algoritmo para encontrar la hipótesis será recursivo.

El método `buildHypothesis` es la implementación de este algoritmo. Toma una variable junto a su encarnación de la cual queremos su hipótesis, el contexto (todo el conocimiento adquirido hasta el momento) y el conjunto de variables ya revisadas. Luego construye el conocimiento particular de esa variable:

1. El `knowledge` en un comienzo es todo lo que sabemos hasta el momento de esa variable: se busca el conocimiento de esa variable en el contexto.
2. Agregamos esta variable al conjunto de variables ya recorridas.
3. Pasamos a buscar todas las variables que sean parte del `knowledge` quitando aquellas que ya hayamos revisado.
4. Llamamos recursivamente a `buildHypothesis` para cada una de las variables que todavía no recorrimos.
5. Finalmente, se hace la conjunción de todo el conocimiento.

Estos pasos se hacen para cada una de las variables involucradas en el mensaje que estamos analizando.

Luego de realizar esta clausura se verifica que no haya dependencia circular ni más de una encarnación por variable llamando al método `notCircular`. De haberlas, se lanza una excepción.

El método `buildHypothesis` retorna un conjunto de sentencias que luego se hace la conjunción gracias al método `setToSentence` que simplemente itera sobre las sentencias y realiza la conjunción de las mismas para devolver una única sentencia que representa la hipótesis completa.

3.7.3. Not Circular

Como se explicó en la Sec. 2.2.4, el criterio de equivalencia propuesto requiere que no exista más de una encarnación por variable en cada clausura de conocimiento. Por este motivo, el método `notCircular` (nombre adjudicado por garantizar la no dependencia circular) corrobora este requerimiento cada vez que se llama al `buildHypotesis`.

El algoritmo implementado toma un conjunto de sentencias (la hipótesis). El mecanismo es sencillo: se recorre toda la hipótesis verificando que haya una única encarnación para cada variable. Caso contrario, el criterio de equivalencia no puede garantizarse.

Cabe mencionar que este criterio de equivalencia y sus correspondientes requerimientos sobre los contextos, restringen mucho la construcción de máquinas para especificar protocolos. Queda pendiente profundizar y estudiar con mayor rigurosidad este criterio de equivalencia para que sea menos restrictivo. Más información al respecto puede encontrarse en la sección 5.2.

3.7.4. Contexts Are Equivalent

En esta sección veremos el método que implementa el criterio de equivalencia desarrollado en la sección 2.2.4.

Este método recibe dos `StateContext` a quienes se los quiere comparar para identificar si son equivalentes. Para esto se construye un `antecedente` y un `consecuente` de la siguiente manera:

- Para cada variable del contexto (recordemos que tienen las mismas variables) se arma la hipótesis, la clausura de conocimiento en cada contexto.
- Para ambas hipótesis, se renombran todas las variables con un prefijo en común para cada contexto (i.e. "old" y "new"). (ver `PropSentenceRenameVisitor` más adelante)
- Se agrega al `antecedente` la igualdad entre las variables de la hipótes del viejo contexto y del nuevo contexto (las variables que tienen el mismo nombre y, tal vez, distinta encarnación).
- Por último se agrega al `consecuente` $hyp\_old \leftrightarrow hyp\_new$. Esto quiere decir, la equivalencia entre ambas hipótesis

Repasemos: en el `antecedente` tenemos la igualdad entre las variables con, tal vez, distintas encarnaciones, mientras que en el consecuente tenemos la equivalencia entre las hipótesis de cada variable.

Finalmente, lo que queremos decidir es si el `antecedente` implica el `consecuente`. Usando el `SatisfactibilityZ3Visitor` encontramos la respuesta a esa pregunta.

PropSentenceRenameVisitor

En el diagrama 3.25 podemos ver el `PropSentenceRenameVisitor`, un *visitor* que dado un prefijo se encarga de renombrar todas las variables de la sentencia.

Hasta acá hemos visto el sustento teórico de la propuesta para la consideración de datos en los protocolos de comunicación, la presentación de un algoritmo para la construcción del Sistema de Transición Sincrónico y su completa implementación. Se vieron también puntos a tener en cuenta para su extensibilidad.

En el próximo capítulo se van a presentar casos de estudio usados para probar la herramienta así como también el análisis de los resultados obtenidos.

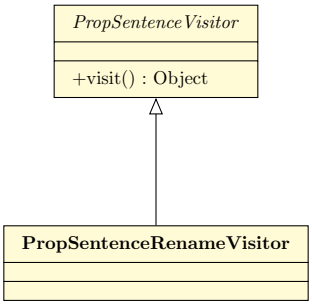

Fig. 3.25: Diagrama de clases Rename Visitor

4. CASOS DE ESTUDIO Y RESULTADOS

En este capítulo pondremos a prueba el desarrollo hecho. Para esto hemos desarrollado diversos casos de estudio separados en dos grupos: un primer grupo de casos *tipo* reales y un segundo grupos con casos puramente sintéticos cuyo objetivo principal es *testear* el desempeño del algoritmo de una manera más controlada.

Entre los casos del primer grupo encontramos:

- Vendedor de Vuelos (Sec. 4.2): Un servicio que consulta los precios de pasajes de distintas agencias y garantiza la reserva de aquel pasaje con menor precio.
- Gestión de Temperatura (Sec. 4.1): basado en el problema de controlar el termostato de una habitación de manera remota.
- Autenticación de Facebook (Sec. 4.3): Caso creado a partir del protocolo de autenticación en servicios de terceros provisto por Facebook.

El segundo grupo (Sec. 4.6) está conformado por una batería de casos construidos a partir de variar la cantidad de participantes involucrados, de comunicaciones realizadas y la existencia o no de datos. Estos casos los bautizamos como "Circulares"por la forma circular que tienen las GCMs correspondientes.

4.1. Gestión de Temperatura

Este ejemplo es el de un sistema que permite controlar la temperatura de un ambiente. El mismo cuenta con 3 participantes involucrados en la comunicación, un número de participantes generalmente usado por protocolos en ambientes reales. El análisis que se hace en esta sección se enfoca en la construcción del Sistema Sincrónico y en las razones de cómo y por qué, con la presencia de ciclos en la comunicación, el algoritmo genera estructuras similares dentro del STS.

Las GCMs de las Figs. 4.1, 4.2 y 4.3 representan los participantes involucrados[1].

El **frontend** es la aplicación del usuario, quien envía la temperatura deseada del ambiente. El **backend** se encarga de garantizar que eso se cumpla comunicándose con los sensores. Por último, el **sensor** puede aumentar o disminuir la temperatura y enviar información de la temperatura en un momento dado.

Veamos cómo sería el funcionamiento de este sistema:

Al comenzar, cada máquina estará en su estado inicial, de esta forma tanto el *sensor* como el *backend* se encuentran en un estado de reposo a la espera de comunicaciones entrantes. El *frontend*, quien modela la interfaz de usuario, comienza el proceso comunicando una temperatura deseada al *backend* usando el mensaje *tempDeseada*.

El *backend* se encarga entonces de cumplir el pedido. Para esto se comunica con el *sensor* para conocer la temperatura actual. Con ese dato, se garantiza que el *backend*

[1] Las GCMs que aparecen para este caso abusan de la notación al usar variables repetidas. Esto fue hecho para priorizar la legibilidad y entendimiento del caso.

Recordemos que esto, por el criterio de equivalencia usado, no es posible. Para realizar lo que aparece en las figuras hubo que renombrar todas las variables y agregar condiciones para igualarlas en los casos donde correspondía.

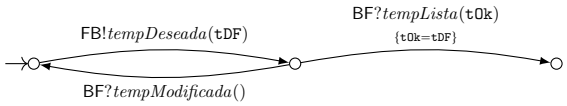

Fig. 4.1: Temperatura Frontend GCM

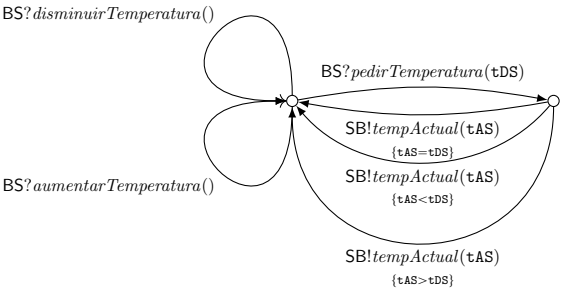

Fig. 4.2: Temperatura Sensor GCM

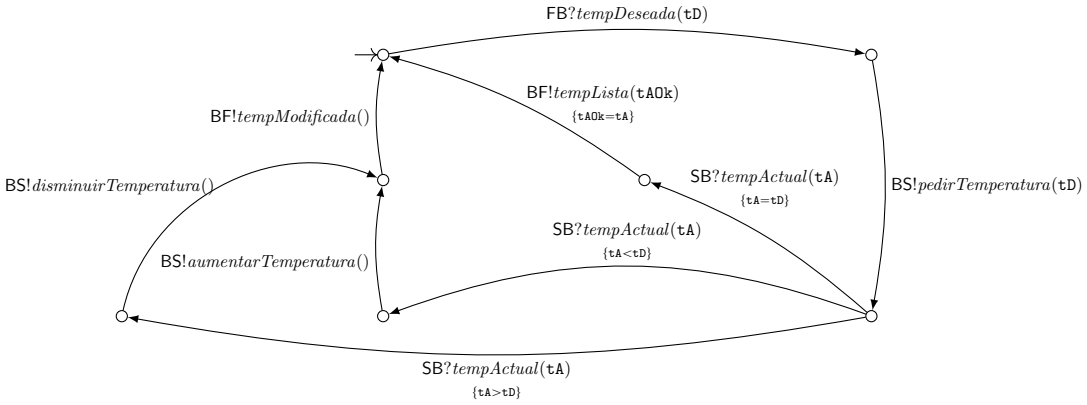

Fig. 4.3: Temperatura Backend GCM

enviará el mensaje de aumentar o disminuir temperatura sólo si corresponde. Esto lo hace gracias a las condiciones sobre los datos en la recepción del mensaje *tempActual*. En caso de realizarse esa comunicación, el *sensor* aumenta o disminuye la temperatura y luego el *backend* informa la modificación de la temperatura.

Se repite el procedimiento hasta alcanzar la temperatura deseada.

Finalmente se garantiza que el *backend* enviará el mensaje *tempLista* sólo en caso de coincidir la temperatura actual obtenida del *sensor* con la temperatura que el usuario había informado a través del *frontend*.

A continuación se va a explicar el Sistema de Transición Sincrónico que construye el algoritmo. Comprender este ejemplo es de gran ayuda ya que las lineas generales de este procedimiento se repiten para el resto de los casos.

En la Fig. 4.4 puede verse el diagrama del Sistema Sincrónico generado. Expliquemos algunas cuestiones:

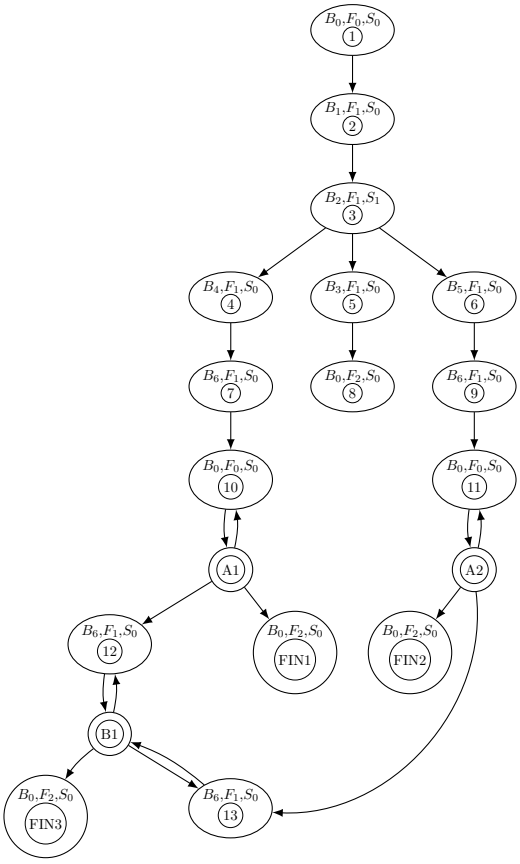

Fig. 4.4: Sistema Sincrónico de Temperatura

- Cada estado está numerado (o identificado) y tienen la configuración de las máquinas correspondientes a los participantes involucrados en la comunicación.

- Los estados numerados del 1 al 11 vamos a tomarlos como la "estructura genérica" del STS.

- El estado (3) corresponde al momento previo al intercambio del mensaje *tempActual* entre el backend y el sensor. Luego de este mensaje se abren 3 posibilidades (vamos a decirles "casos"):
  - El caso donde efectivamente la temperatura es la deseada y concluye la comunicación (estados (5) y (8)).
  - Los casos donde no coincide la temperatura y hay que aumentar o disminuir.
- El estado (10) y (11) son el comienzo de un nuevo ciclo de comunicaciones: la configuración de las máquinas es nuevamente la inicial, sin embargo el conocimiento adquirido hace que estos estados no sean equivalentes con el estado inicial.
- (A1) y (A2) representan la "estructura genérica". Sus estados no son equivalentes a los anteriores (pues tienen conocimiento adquirido que antes no se tenía) ni entre sí (ya que una estructura proviene de que la temperatura deseada sea mayor que la actual y la otra proviene de que sea menor que la actual).
- Desde ambas estructuras vuelven a aparecer 3 casos:
  - El caso donde la temperatura es la deseada y concluye la comunicación ((FIN1) y (FIN2)). Estos dos estados no son equivalentes entre si ni tampoco con (8) por sus contextos.
  - Los casos donde no coincide la temperatura y hay que aumentarla o disminuirla.
  - Podemos notar que tanto de (A1) como de (A2) hay transiciones a (10) y (11). Esto corresponde a recorrer la estructura y tener el mismo caso (de aumento o disminución) que antes. Esta vez sí se llega a estados equivalentes y por lo tanto no se abre una nueva rama.
  - Si el caso es el contrario (la primera vez se aumentó y la siguiente se disminuyó la temperatura o viceversa), (A1) va a (12) y (A2) va a (13)
- Desde (12) sucede algo similar que antes: paso a recorrer la "estructura genérica" (B1) que no es equivalente a ninguna de las anteriores, pues tiene el conocimiento de haber disminuido la temperatura primero y aumentado después.
- Veamos que desde (13) pasa algo similar a (12), sólo que esta vez se pasa de aumentar la temperatura primero a disminuirla luego. Si bien el orden en que realizan las acciones es distinto, la estructura que se va a recorrer ahora es equivalente a la de (12).
- Nuevamente hay 3 casos luego de recorrer (B1):
  - El caso donde la temperatura es la deseada y concluye la comunicación llegando a (FIN3). Otra vez, este caso no es equivalente a los otros estados finales pues

tiene el conocimiento de haber aumentado **Y** disminuido la temperatura para alcanzar la temperatura deseada.

- Y los otros dos casos donde hay que disminuir o aumentar y van a (12) y (13) correspondientemente para luego volver a recorrer (B1) ya que no se siguen generando "estructuras genéricas" pues son equivalentes a (B1). Es decir, ya no se genera conocimiento nuevo.

Este ejemplo sirve para visualizar cómo la presencia de ciclos en las comunicaciones entre participantesgeneran nuevas estructuras. A su vez, vemos que el criterio de equivalencia es fundamental para evitar que esas estructuras se vuelvan infinitas y el algoritmo converja.

El envío del dato de la temperatura deseada en la comunicación del mensaje *pedirTemperatura* permite que el sensor pueda garantizar que su respuesta sea menor, mayor o igual a la temperatura deseada. Esto es necesario para la conformación del sistema, sin embargo no parece realista que el protocolo del sensor requiera la temperatura deseada para informar la temperatura actual del ambiente. Esta es una limitación que aparece para armar los protocolos. Si no enviásemos ese dato no podríamos garantizar los distintos casos de la relación entre la temperatura actual y la deseada.

Por otro lado, que la repetición del ciclo (que sucede luego del envío del mensaje *tempModificada*) sea a partir de un nuevo envío de la temperatura deseada, vuelve a parecer poco realista pero es necesario para reencarnar esa variable y no generar dependencia circular. Estas limitaciones, sumadas al renombre de todas las variables, convierten a la construcción de los participantes algo poco intuitivo.

4.2. Vendedor de Vuelos

Este ejemplo recrea un servicio de búsqueda de vuelos económicos. Luego de presentar a los participantes involucrados se observará cómo crece el Sistema Sincrónico generado al aumentar la cantidad de agencias (uno de los participantes). Al finalizar las pruebas, se dejarán algunos puntos a considerar respecto al uso de variables renombradas (es decir, no permitir variables de igual nombre) y cómo esta decisión condiciona la construcción de las GCMs y su consecuente impacto en el tamaño del STS generado.

Veamos el ejemplo: hay un servicio **vendedor** (Fig. 4.6)que se comunica con las distintas **agencias** de vuelo (Fig. 4.7) para consultar el precio de un pasaje. Luego esta información es comunicada al **analizador de precios** (Fig. 4.5) quien actua en función de los datos recibidos reservando el pasaje, en este caso, más económico.

En este caso se puede ver cómo la incorporación de condiciones nos permite expresar garantías funcionales sobre el protocolo. Podrían pensarse otros analizadores con otras garantías respecto a los precios de los pasajes, adquiriendo aquel de mayor precio o de precio promedio.

En el ejemplo que se muestra en las Fig. 4.5 y Fig. 4.6 usamos sólo 2 agencias. El vendedor puede optar por preguntar primero a A_1 y luego a A_2 por el precio o viseversa. Una vez obtenidos los precios los comunica al analizador. El analizador recibirá el mensaje del vendedor garantizando que reservará el de menor costo. De haberse reservado enviará el mensaje *ok*, caso contrario *noOk*.

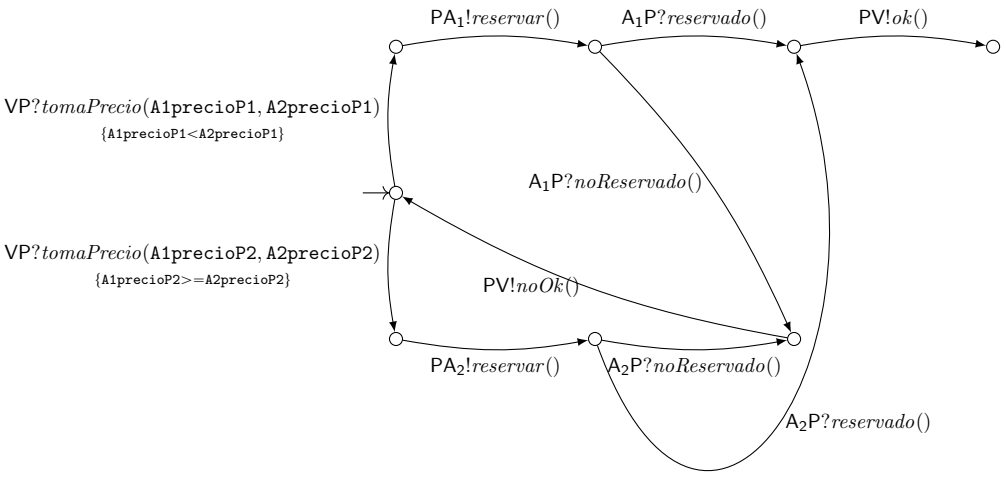

Fig. 4.5: Analizador de precios de vuelos GCM

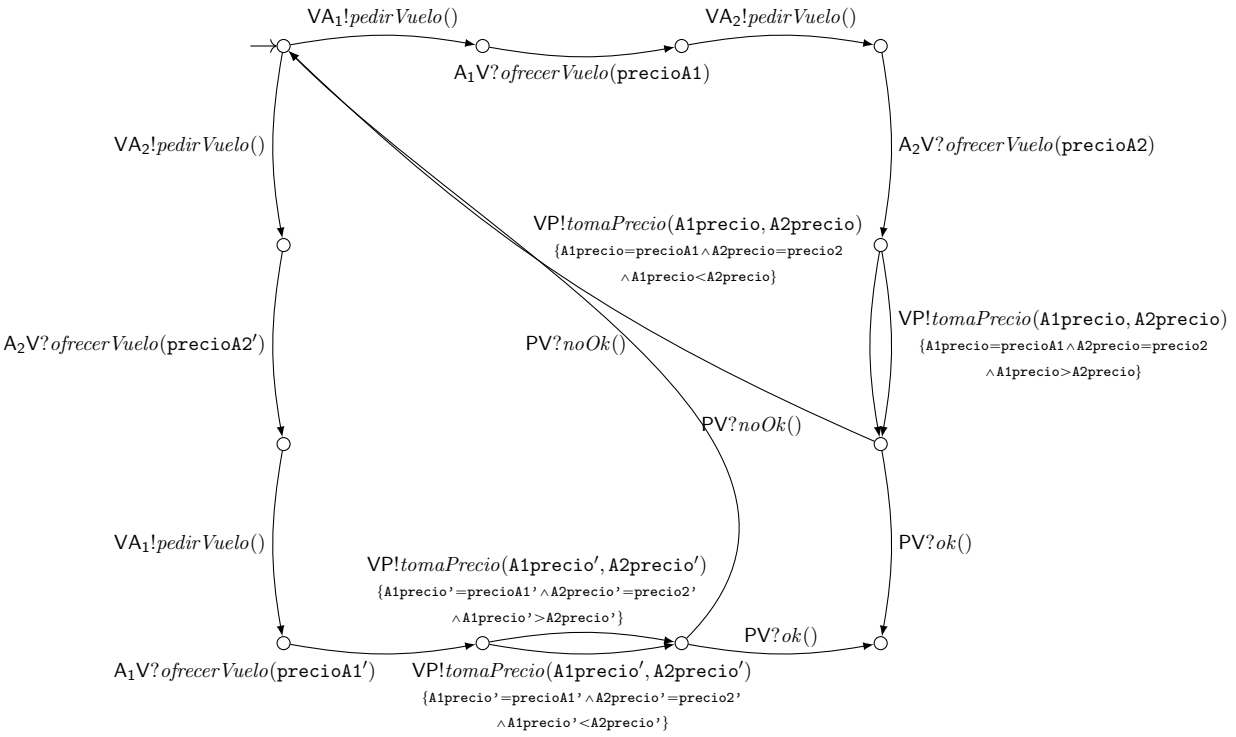

Fig. 4.6: Vendedor de Vuelos GCM

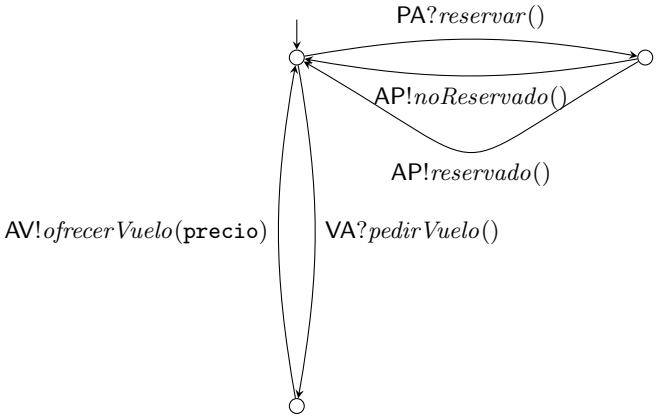

Fig. 4.7: Agencia de Vuelos GCM

Por su parte, la agencia acepta mensajes de pedidos de vuelos y responde con el precio. También permite mensajes de reserva respondiendo si se realizó o no la reserva.

Si vemos la Fig. 4.6 puede verse la GCM del vendedor de vuelos y sus caminos. Cada camino (en ese caso son solo 2, en el caso de 3 agencias serán 6) tiene un envío del mensaje *tomaPrecio* por cada posibilidad en la comparación de precios (i.e. cuál es el menor precio).

El caso de 2 agencias obtiene un Sistema Sincrónico de 699 estados. Por su tamaño se hace muy complejo incluir en este informe un detalle del sistema generado pero puede recorrerse utilizando la interfaz implementada.

Este caso resulta atractivo para analizar qué sucede cuando aumentamos la cantidad de agencias a consultar. Veamos cómo cambian las GCMs

- El analizador de precios aumenta 1 camino por cada nueva agencia. El camino del que hablamos es el generado por las transiciones *tomaPrecio, reservar, reservado/-noReservado*. Esto se debe a que ahora tiene una posibilidad nueva de menor precio.

- Se agrega 1 GCM nueva por cada agencia.

- El vendedor tiene en total *n!* caminos distintos para tomar, siendo n la cantidad de agencia. En el caso de la Fig. 4.6 son dos agencias. Cada camino corresponde a preguntar primero a A1 y luego a A2 y viceversa. Si tuviesemos tres, tendríamos 6 caminos distintos para tomar.

- La cantidad de transiciones *tomaPrecio* aumenta en 1 por cada agencia, en cada camino en el vendedor.

Con este razonamiento se hicieron pruebas con 3 agencias. El aumento de caminos y su consecuente cantidad de estados era esperado, sin embargo el resultado fue poco esperanzador: se alcanzaron 60 mil estados y una excepción de falta de memoria (habiendo establecido 32 Gb para el consumo del *solver*).

Para realizar un análisis más exhaustivo vamos a observar qué pasa con este caso en dos dimensiones distintas: la cantidad de caminos y la cantidad de opciones (quién ofrece el menor precio de vuelo). Siendo 3 agencias, la cantidad total de caminos es 6 y la cantidad de opciones es 3 (1 por cada agencia).

Caminos	# STS States	# Transiciones
1	19	19
2	85	90
3	319	347
4	1037	1148
5	3531	3955
6	11581	13086

Tab. 4.1: Estados generados aumentando la cantidad de caminos para 3 agencias, 1 opción

Caminos	# STS States	# Transiciones
1	54	58
2	911	1031
3	11115	12811
4	OutOfMemory (60473)	OutOfMemory (69208)

Tab. 4.2: Estados generados aumentando la cantidad de caminos para 3 agencias, 2 opciones

La cantidad de caminos se fue recortando eliminando las transiciones correspondientes en el Vendedor de Vuelos y la cantidad de opciones depende de las transiciones *tomaPrecio*.

Las tablas 4.1, 4.2 y 4.4 muestran los resultados obtenidos [2].

En la tabla 4.1 podemos ver que en cada aumento de camino aumentan más de 3 veces la cantidad de estados generados. Si vemos esto mismo en el caso de 2 opciones nos encontramos con que en cada paso hay un aumento de estados de más de 10 veces. Si bien para 3 opciones no hay datos suficientes para dar alguna conclusión, es esperable que no mejore.

Por otro lado, si comparamos la cantidad de estados de 1, 2 y 3 caminos para 1 y 2 opciones vemos que hay una diferencia de 2, 10 y más de 30 veces. Si comparamos también el valor llegado al momento de la excepción *OutOfMemory* hay una diferencia de casi 60 veces. Esto muestra que la la cantidad de estados crece muy rápido.

Teniendo en cuenta el crecimiento que fue teniendo el caso de 2 opciones y los pocos datos obtenidos para 3, si bien no podemos estar seguros de cuanto aumentará, si podemos ver que habrá bastante diferencia entre esos dos casos. Por lo tanto, pretender generar el Sistema Sincrónico de 3 opciones para los 6 caminos es inviable.

[2] Los casos con OutOfMemory tienen entre paréntesis la cantidad de estados y transiciones alcanzadas al momento de producirse la excepción.

Caminos	# STS States	# Transiciones
1	146	162
2	2632	3012
3	OutOfMemory (>60000)	OutOfMemory (>60000)

Tab. 4.3: Estados generados aumentando la cantidad de caminos para 3 agencias, 3 opciones

4.3. Autenticación en Facebook

El siguiente es un caso de análisis basado en un protocolo real, el protocolo de autenticación de Facebook Login[3] usado para acceder a sitios de particulares utilizando el usuario y contraseña de Facebook. Se verá primero cómo se modeló y simplificó el protocolo para realizar los estudios. Luego, partiendo de 3 casos distintos (logueo correcto, logueo incorrecto y token inválido) se mostrará el correcto funcionamiento del algoritmo analizando los STSs generados.

El primer paso fue modelar el protocolo el cual se hizo usando 4 GCMs:

- Cliente
- WebSite
- Autenticador
- Validador

El **Cliente** es quien quiere acceder al **Website**. Este lo deriva al **Autenticador** con quien realiza el logueo con su cuenta de facebook. Como contrapartida el Autenticador le otorga al Cliente un token el cual envía al Website. El Website finalmente corrobora que el logueo haya sido satisfactorio comunicándose con el **Validador** enviando el token. Este último verifica que el token sea el correcto y otorga el acceso al Website.

En las Figs. 4.8, 4.9, 4.10 y 4.11 pueden encontrarse las 4 GCMs. Como se puede ver, no solo se modeló el acceso sino también la comunicación entre el Cliente y el Website para hacer los pedidos de recursos con su consecuente verificación de acceso a los recursos.

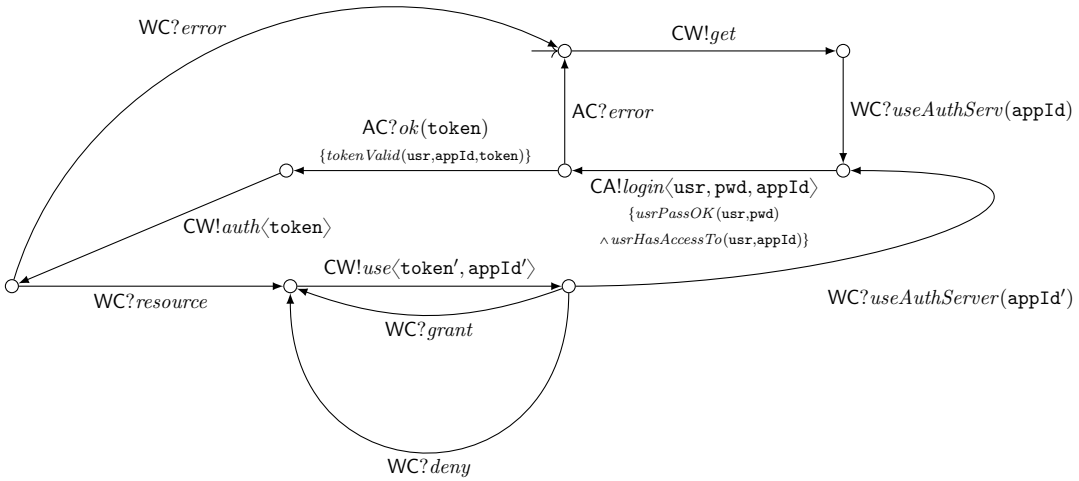

Fig. 4.8: Cliente GCM Facebook

Para el análisis decidimos simplificar el protocolo con el objetivo de poder modelar mejor las condiciones sobre los datos usando lógica proposicional con aritmética.

[3] https://developers.facebook.com/docs/facebook-login

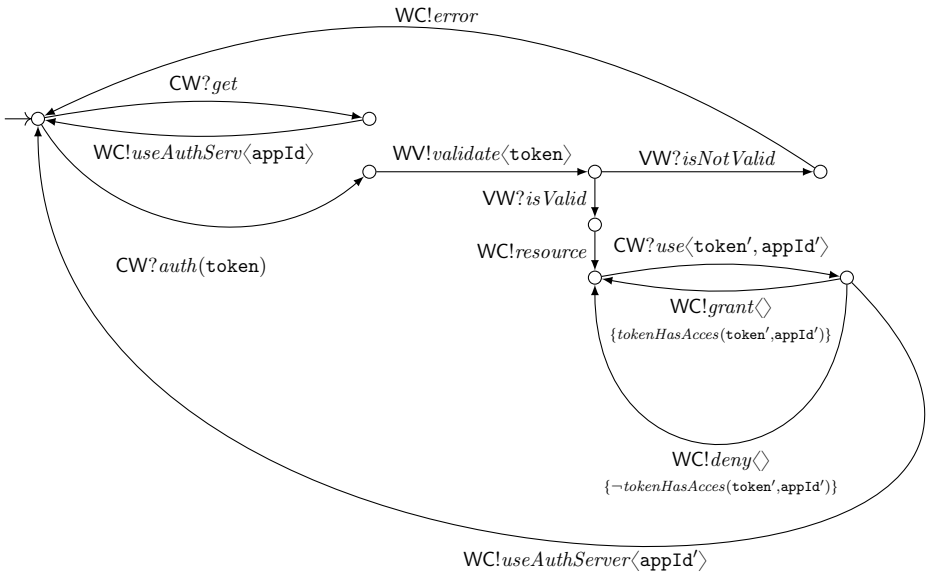

Fig. 4.9: WebSite GCM Facebook

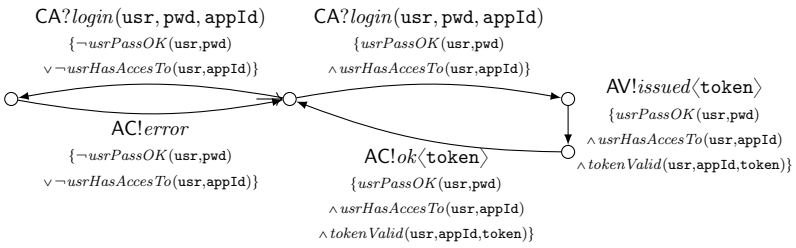

Fig. 4.10: Auth GCM Facebook

La simplificación se hizo quitando los pedidos de recursos posteriores al logueo. En las Figs. 4.12 y 4.13 puede verse esa simplificación que solo modifica al Cliente y al Website. Repasemos entonces como funciona el sistema de logueo con los cambios:

- El **cliente** envía el mensaje de *login* con el usuario, contraseña y la app a la que quiere acceder. Este mensaje es enviado al autenticador.
- La recepción del mensaje por parte del **autenticador** dependerá de la validez de los datos enviados. El autenticador entonces garantiza que en caso de no ser validos los datos responderá con un mensaje de error. Caso contrario, generará un token que enviará tanto al validador (mensaje *issued*) como al cliente (mensaje *ok*).
- El **cliente**, una vez enviado el mensaje de login esperará la respuesta que será o bien errónea y entonces volverá al estado inicial; o bien afirmativa conteniendo el token correspondiente. De ser afirmativa enviará el mensaje *auth* al website con el token.

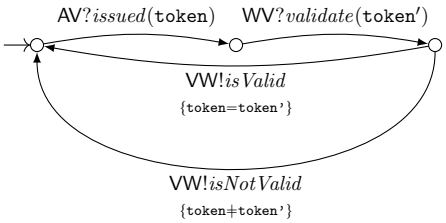

Fig. 4.11: Validator GCM Facebook

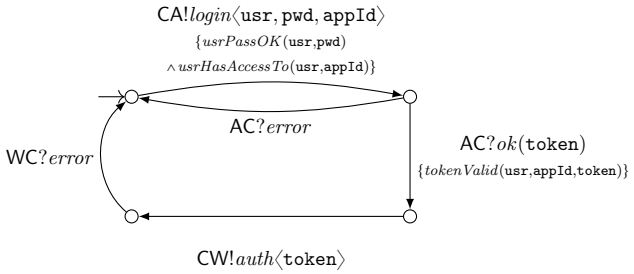

Fig. 4.12: Cliente GCM Facebook Simple

- El **website** recibirá el mensaje *auth* del cliente con el token. Envirá este dato al validador para corroborar su validez lo cual sabrá según el mensaje que reciba por parte del validador: *isValid* o *isNotValid*.
- El **validador** por su parte, habrá recibido el mensaje *issued* del autenticador con el token válido y luego el mensaje del website con el token que le suministró el cliente. Es aquí donde el validador garantiza que enviará el mensaje *isValid* sólo en caso de que ambos tokens sean igual. Caso contrario enviará el mensaje *isNotValid*.
- Por último, el **website** recibirá la respuesta del autenticador y en caso de no ser afirmativa enviará el mensaje de error al cliente.

Finalmente se adaptó el caso a la implementación del algoritmo y a la lógica usada. Se renombraron las variables de envíos y recepciones, se agregaron condiciones de igualdad entre las variables necesarias y se adaptó el acceso y verificación de datos al lenguaje de lógica proposicional con aritmética.

Las Figs. 4.14, 4.15, 4.16 y 4.17 son las máquinas que codificamos para hacer las pruebas.

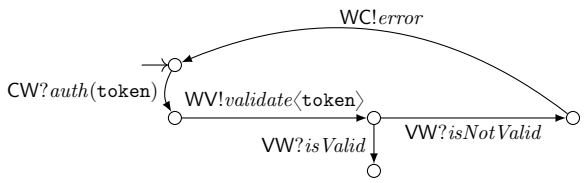

Fig. 4.13: WebSite GCM Facebook Simple

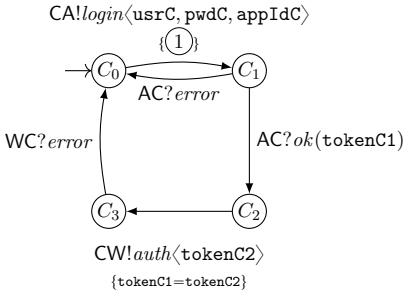

Fig. 4.14: Cliente GCM Facebook Simple Con Datos

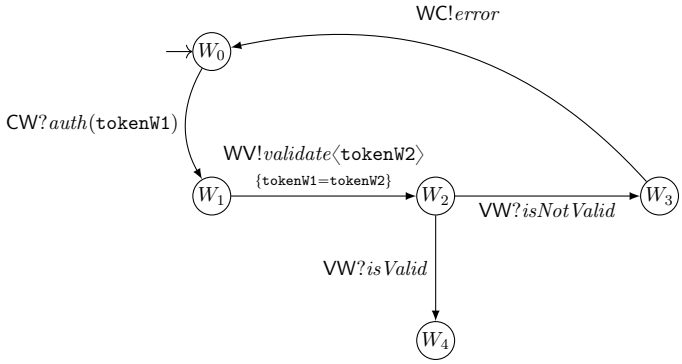

Fig. 4.15: WebSite GCM Facebook Simple Con Datos

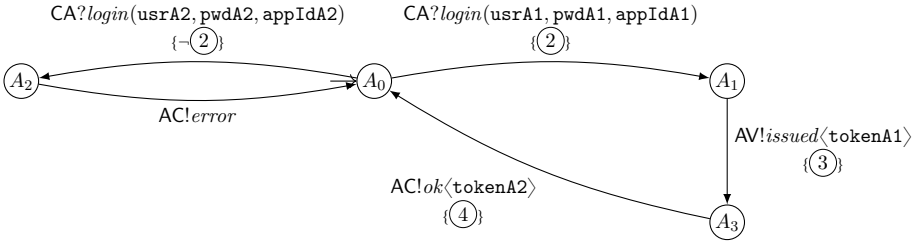

Fig. 4.16: Auth GCM Facebook Con Datos

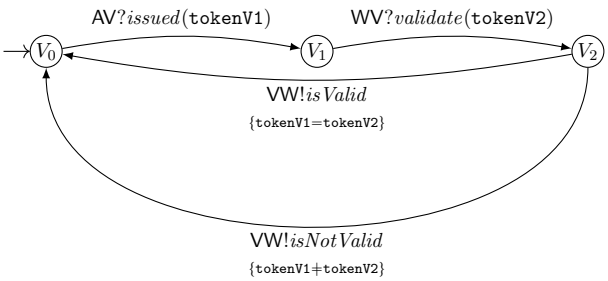

Fig. 4.17: Validator GCM Facebook Con Datos

Estas son las condiciones que no se detallan en las figuras y se definiran para cada caso de estudio:

- (1) Los datos de logueo que envía el usuario.
- (2) Las condiciones que garantizan que el logueo es válido o no.
- (3) El token que el autenticador envía al validador.
- (4) El token que el autenticador envía al usuario.

A continuación analizaremos la construcción del STS y el conocimiento generado para 3 casos distintos. El contexto completo de cada Sistema Sincrónico puede encontrarse en el apéndice.

4.3.1. Sistema Sincrónico en caso donde el logueo es correcto

El primer caso a analizar es cuando el logueo se realiza correctamente. Para garantizar esto vamos a ver las condiciones sobre los datos que usamos:

- (1) pwdA1= usrA1+1[4] $\wedge$ appIdA1=usrA1+2[5]
- (2) pwdA1= usrA1+1 $\wedge$ appIdA1=usrA1+2
- (3) tokenA1=1234
- (4) tokenA2=1234

Recordemos que (2) es la condición de logueo correcto, esto justifica los datos puestos en (1). (3) y (4) tienen el mismo valor para que el token coincida.

Con estas condiciones el Sistema Sincrónico generado está en la Fig. 4.18. En la figura se observan los estados de cada GCM y la referencia al conocimiento adquirido hasta ese momento.

Si se observa la secuencia de transición de estados de cada una de las máquinas puede verse que efectivamente el camino recorrido es el correspondiente al logueo correcto. Puede verse que la máquina correspondiente al autenticador en el estado con contexto (2) se encuentra en el estado A_1. Como se ve en la Fig. 4.16 este estado corresponde a la recepción del mensaje login con datos válidos.

Del mismo modo podemos apreciar que luego de la última transición, donde el validador pasa del estado V_2 al estado V_0 (correspondiente con la respuesta de si el token es o no válido), el estado del website finalmente es W_4, correspondiente con la recepción del mensaje *isValid*.

[4] Esto reemplaza al predicado *usrPassOK(usr,pwd)* de la Fig. 4.12
[5] Esto reemplaza al predicado *usrHasAccessTo(usr,appId)* de la Fig. 4.12

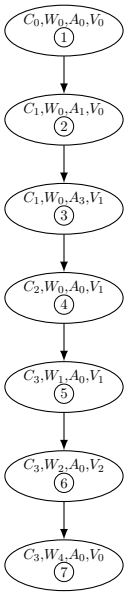

Fig. 4.18: Sistema Sincrónico de Facebook con datos válidos

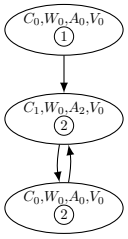

Fig. 4.19: Sistema Sincrónico de Facebook con AppId erroneo

4.3.2. Sistema Sincrónico en caso donde el logueo es incorrecto

En este otro caso se cambiaron los datos enviados para que no se satisfagan las condiciones de logueo:

- (1) ∼(pwdA1= usrA1+1 ∧ appIdA1=usrA1+2)
- (2) pwdA1= usrA1+1 ∧ appIdA1=usrA1+2
- (3) tokenA1=1234
- (4) tokenA2=1234

En este caso, no se satisface la condición de logueo correcto. En la Fig. 4.19 nos encontramos con que el el Cliente y el Autenticador se comunican y el segundo llega al estado A_2 enviando luego el mensaje de error de logueo como vimos en la Fig. 4.16. El sistema entra entonces en un ciclo ya que los datos no cambian y siempre se producirá el camino de logueo erroneo.

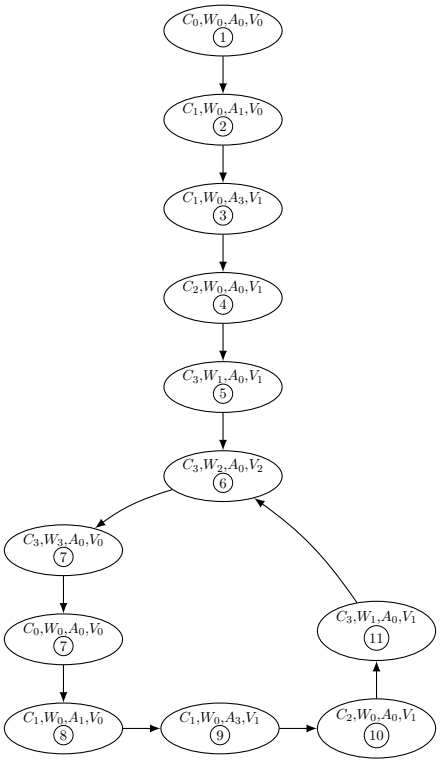

Fig. 4.20: Sistema Sincrónico de Facebook con token inválido

4.3.3. Sistema Sincrónico en caso donde el token es incorrecto

Por último revisamos el caso donde el token que se quiere validar es incorrecto. Para esto hicimos que el Autenticador envíe distintos tokens al Cliente y al Validador.

- (1) pwdA1= usrA1+1 $\wedge$ appIdA1=usrA1+2
- (2) pwdA1= usrA1+1 $\wedge$ appIdA1=usrA1+2
- (3) tokenA1=1234
- (4) tokenA2=123

Como se ve en el Sistema Sincrónico (Fig. 4.20) el logueo se realiza correctamente (el autenticador alcanza el estado A_1). Luego se continua el mismo camino que el del logueo correcto hasta llegar al momento de verificación de los tokens (esto se da en el estado con contexto (6)): el validador se encuentra en el estado V_2 y el website en W_2. Luego de la transición, dados los datos del contexto (ver apéndice) el validador envía el mensaje *isNotValid* llegando al estado V_0 mientras el website al W_3. Luego de esto comunica el error al cliente y comienza nuevamente el ciclo, pero esta vez con todo el conocimiento generado (por eso no se transiciona al primer estado del Sistema Sincrónico).

Como los datos no cambian, el Sistema entra en un ciclo que, ahora así, comienza a ser equivalente.

	# STS States	# Transiciones
Con Datos y Condiciones	699	820
Con Datos y Sin Condiciones	9401	13744
Sin Datos	22	32

Tab. 4.4: Comparativa de estados y transiciones generadas para el caso de Vendedor de Vuelos (caso con 2 agencias)

	# STS States	# Transiciones
Con Datos y Condiciones	34	38
Con Datos y Sin Condiciones	1143	2003
Sin Datos	8	9

Tab. 4.5: Comparativa de estados y transiciones generadas para el caso de Temperatura

4.4. Comparativa de casos con y sin datos

En esta sección vamos a ver la comparativa de estados y transiciones entre construir el Sistema Sincrónico a partir de GCMs con y sin datos.

Esta comparativa fue hecha para los 3 casos vistos anteriormente. El ejemplo de las agencias de vuelos usado fue el de 2 agencias, mientras que en el de Facebook se construyó el ejemplo con todas las transiciones disponibles (logueo correcto, incorrecto y con error de token). Los resultado pueden verse en las tablas 4.4, 4.5 y 4.6.

En primer lugar, respecto a las diferencias encontradas entre tener datos con condiciones y no tener datos se deben, principalmente a la equivalencia de contextos cuando no hay datos. Esto quiere decir que dos estados del Sistema Sincrónico que con datos se diferenciaban por sus contextos, cuando no tienen datos los contextos son equivalentes y por lo tanto se reduce la cantidad total de estados.

Podemos deducir que aquellos sistemas que no involucren ciclos tendrán una diferencia menor entre la cantidad de estados si tienen o no datos. De echo, si suponemos que las condiciones se satisfacen siempre, deberían tener exactamente la misma cantidad de estados. Esto se debe a que, en casos con datos, al entrar en un ciclo existe al menos 1 copia de la estructura original por la presencia de conocimiento no equivalente (ya que la primera vez que se recorrió no se tenía esa información), mientras que sin datos sólo se genera una sola vez la estructura cíclica, las siguientes serán equivalentes por no tener conocimiento.

En la tabla 4.5, del caso Gestión de Temperatura, podemos ver que sin datos se generan 8 estados y con datos y condiciones 34. Como vimos al analizar ese ejemplo, la “estructura” se repetía unas 4 veces, lo cual va en consonancia con los resultados presentados.

Por otro lado, la comparación entre tener y no tener condiciones es imponente. Si

	# STS States	# Transiciones
Con Datos y Condiciones	42	46
Con Datos y Sin Condiciones	577	920
Sin Datos	9	13

Tab. 4.6: Comparativa de estados y transiciones generadas para el caso de Login de Facebook

comparamos las filas con datos y condiciones contra las filas con datos y sin condiciones de las tablas 4.4, 4.5 y 4.6 vemos que la cantidad de estados aumenta enormemente, llegando a ser 30 veces mayor. Esto es causado por poder recorrer todas las transiciones y combinaciones posibles que sino son recortadas por las condiciones.

4.5. Rendimiento de CPU

En esta sección vamos a analizar el consumo de CPU de los métodos para los 3 casos vistos hasta el momento. Para esto vamos a ver el rendimiento de los principales métodos del algoritmo. Los resultados pueden verse en las tablas 4.7, 4.8 y 4.9

Método	Tiempo de CPU (%)	# Invocaciones
isContradiction	11,27	956
satisfaction	10,90	956
contextsAreEquivalent	69,04	1869
buildHypothesis	6,63	2600

Tab. 4.7: Consumo de métodos en caso de Vuelos (2 agencias) (tiempo total en ms: 78460)

Método	Tiempo de CPU (%)	# Invocaciones
isContradiction	33,17	61
satisfaction	24,55	61
contextsAreEquivalent	27,06	8
buildHypothesis	11,49	244

Tab. 4.8: Consumo de métodos en caso de Temperatura (tiempo total en ms: 3193)

Método	Tiempo de CPU (%)	# Invocaciones
isContradiction	31,81	60
satisfaction	20,36	54
contextsAreEquivalent	21,45	5
buildHypothesis	16,12	240

Tab. 4.9: Consumo de métodos en caso de Login de Facebook (tiempo total en ms: 2760)

Lo primero que salta a la vista es el rol fundamental que tiene `contextsAreEquivalent` en el consumo general del algoritmo. Aun en los casos donde se invoca al método menos de 10 veces, su participación en el consumo es mayor al 20 % . Es claro entonces que el algoritmo depende de cuan costoso sea chequear la equivalencia entre contextos. A su vez, dependera también de que tan costoso sea el método de demostración del lenguaje lógico elegido.

Si tomamos en cuenta el tiempo total de cada caso, podemos calcular que por cada invocación `isContradiction` consume entre 10 y 20 ms, `satisfaction` alrededor de 10 ms y `buildHypothesis` 2 ms. Mientras que `contextsAreEquivalent` consume 30 ms en un caso y arriba de 100 en los otros dos. Si bien una mejora y un análisis más profundo es necesario para todos los métodos, se hace necesario optimizar `contextsAreEquivalent`.

4.6. Circulares

Los siguientes serán casos sintéticos pensados para el análisis de rendimiento del algoritmo. Luego de presentar los casos de estudio se presentarán los resultados obtenidos. A partir de los resultados se explicará a qué se debe el tamaño de los Sistemas Sincrónicos generados y un detalle del consumo de distintos métodos principales del algoritmo junto con las modificaciones hechas para mejorar el rendimiento.

Se contará con 3 GCMs:

- A: Será la máquina principal que se comunicara con B y C. Tiene un ciclo de n estados y en cada estado tiene otro ciclo.
- B: Será la máquina que se comunique en los n ciclos de cada estado.
- C: Será la máquina que se comunique en el ciclo de los n estados.

En la Fig. 4.21 puede verse la máquina A para 3 casos distintos y en la Fig. 4.22 pueden verse las máquinas de los otros dos participantes. El análisis se hizo para 4 conjuntos de máquinas:

- Una única máquina B y sin datos de envío ni recepción.
  - Con 100 ciclos.
  - Con 500 ciclos.
  - Con 1000 ciclos.
  - Con 2500 ciclos.
  - Con 5000 ciclos.
- Una única máquina B y con datos de envío y recepción.
  - Con 1 ciclo.
  - Con 2 ciclos.
  - Con 3 ciclos.
  - Con 4 ciclos.
  - Con 5 ciclos.
  - Con 6 ciclos.
  - Con 7 ciclos.
- N máquinas B, una para cada ciclo de A, y sin datos de envío ni recepción.
  - Con 100 ciclos.
  - Con 500 ciclos.
- N máquinas B, una para cada ciclo de A, y con datos de envío y recepción.
  - Con 1 ciclo.
  - Con 2 ciclos.
  - Con 3 ciclos.

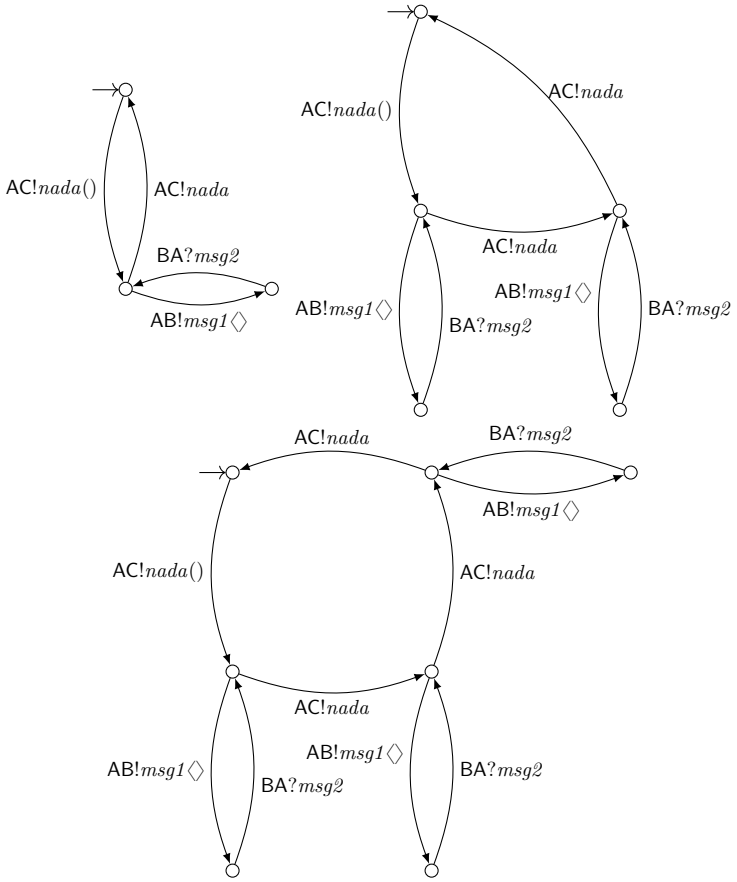

Fig. 4.21: GCM A para 1, 2 y 3 ciclos

- Con 4 ciclos.
- Con 5 ciclos.
- Con 6 ciclos.
- Con 7 ciclos.
- Con 8 ciclos.
- Con 9 ciclos.

El objetivo fue poder analizar el rendimiento tanto de CPU como de memoria haciendo enfoque en las distintas partes cruciales del algoritmo. Por este motivo hay 4 conjuntos de análisis, combinando la existencia de datos y la cantidad de máquinas. De este modo se pudo hacer comparaciones en cuanto al consumo del solver versus la construcción de la estructura.

Para los casos con datos se le agregó el envío de 3 variables X, Y y Z al *msg1* que envía la máquina A y recibe/n la/s maquina/s B. Si bien a priori uno pensaría que al no haber condiciones sobre los datos (únicamente se envían y se reciben pero no tienen sentencias asociadas) el análisis posible del solver es pobre, nos encontramos con problemas que no habían aparecido hasta el momento y se lograron mejoras muy significativas en cuanto al rendimiento. Estas mejoras se desarrollan más adelante en esta sección.

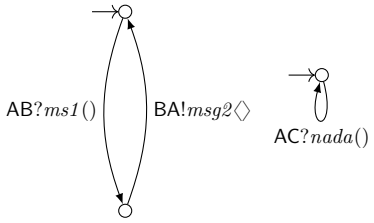

Fig. 4.22: GCM B y C

Ciclos	# STS States
100	201
500	1001
1000	2001
2500	5001
5000	10001

Tab. 4.10: Estados generados en caso de una máquina B y sin datos

Una vez pensados y codificados los casos de estudio se realizaron pruebas de rendimiento de la implementación. En esta primer prueba hubo varios datos interesantes. Al ver las tablas **??** y **??**, es evidente la limitación que nos da el uso o no de datos en las comunicaciones: sin datos podemos construir casos de miles de ciclos mientras que con datos apenas los contamos con los dedos de la mano.

Un dato que generó curiosidad al ver estas tablas fue la cantidad de STS States generados en cada caso. Para los casos sin datos la cantidad de estados generados es igual a la cantidad de estados que tiene la máquina A, que son exactamente: la cantidad de ciclos $\times 2 + 1$ (el estado incial). Pero en el caso de comunicaciones con datos se encuentra mucha diferencia si hay una única máquina B o si hay N máquinas B. Expliquemos esto.

Supongamos que tenemos 2 ciclos (ya que con 1 ciclo la cantidad es la misma), tendríamos entonces la máquina A de la Fig. 4.23. Veamos qué pasa si tengo una única máquina B (su variable será Z):

- Recorro el primer ciclo primero y después el segundo y vuelvo al estado inicial:

Ciclos	Unica máquina B	N máquinas B
1	5	5
2	19	16
3	64	44
4	197	112
5	566	272
6	1543	640
7	4040	1472
8	no se pudo correr	3328
9	no se pudo correr	7424

Tab. 4.11: Estados generados en casos con datos

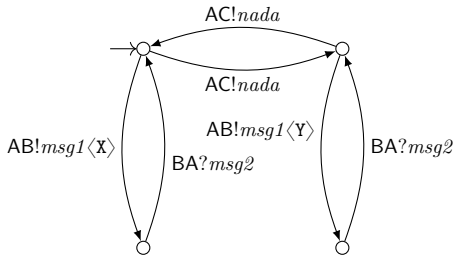

Fig. 4.23: GCM A 2 ciclos con datos

  - Al finalizar el primer ciclo tengo: $X_1 = Z_1$
  - Luego del segundo ciclo tendré: $Y_2 = Z_2$

- Si recorro de manera inversa, primero el segundo ciclo y al volver al estado inicial recorro el primer ciclo:

  - Al finalizar el segundo ciclo tengo: $Y_1 = Z_1$
  - Luego del primer ciclo tendré: $X_2 = Z_2$

En este ejemplo vemos que llegamos al mismo estado pero el conocimiento no es equivalente.

Si contamos con N (N = 2 en este caso) máquinas B (Siendo Z1 y Z2 sus respectivas variables) obtendremos:

- Recorro el primer ciclo primero y después el segundo y vuelvo al estado inicial:

  - Al finalizar el primer ciclo tengo: $X_1 = Z1_1$
  - Luego del segundo ciclo tendré: $Y_2 = Z2_2$

- Si recorro de manera inversa, primero el segundo ciclo y al volver al estado inicial recorro el primer ciclo:

  - Al finalizar el segundo ciclo tengo: $Y_1 = Z2_1$
  - Luego del primer ciclo tendré: $X_2 = Z1_2$

En este caso el conocimiento generado sí es equivalente y por lo tanto no serán 2 STS States distintos.

Extendiendo esta idea puede entenderse por qué al haber N máquinas B distintas se generan menos estados.

Pasemos ahora al análisis más minucioso de rendimiento y consumo de CPU.

Los siguientes análisis fueron hechos a partir de los resultados obtenidos con las primeras versiones del algoritmo. Estas versiones utilizaban un criterio de equivalencia que no tenía sustento para garantizar la equivalencia de estados y que al no ser correcto se dejó de utilizar.

De todos modos, los resultados y el análisis hecho, así como los cambios que estos análisis provocaron en la implementación fueron correctos y con un alto valor para la mejora del rendimiento general.

Luego de estos primeros datos y cambios se encuentran los resultados obtenidos con la última implementación que incorpora el criterio de equivalencia definitivo para este trabajo.

Antes de continuar, cabe decir que el cambio en el criterio de equivalencia no modifica la cantidad de estados generados ni en el comportamiento general del algoritmo ya que los participantes no incluían condiciones en sus mensajes.

Dicho esto pasemos a observar las tablas de resultados de rendimiento.

4.6.1. Resultados con criterio de equivalencia obsoleto

Ciclos	Tiempo de CPU (%)
100	3,13
500	6,8
1000	10,96
2500	25,91
5000	39,07

Tab. 4.12: Consumo de AbstractList.Equals() en caso de una máquina B y sin datos

Ciclos	Tiempo de CPU (%)
100	53,34
500	99,54

Tab. 4.13: Consumo de CommunicatingMachine.hasChannel() en caso de N máquinas B y sin datos

Ciclos	Tiempo de CPU (%)
1	15,59
2	27,12
3	31,37
4	33,13
5	33,19
6	31,17
7	28,52

Tab. 4.14: Consumo de sts.nextState en caso de una máquina B y con datos

En la tabla 4.12 se observa como a medida que aumenta la cantidad de ciclos aumenta el consumo del método de igualdad entre listas. A partir de estos datos fue rápido encontrar que la implementación del algoritmo hasta ese momento era deficiente: al generar un nuevo STS State y querer corroborar si existía otro equivalente, se recorrían todos los estados generados previamente comparando, en primer lugar, la *lista* de estados de las máquinas. Aquí se encontraba el principal factor de consumo de este método.

El otro método que resalta en el caso sin datos fue el de `hasChannel`. En la tabla 4.13 puede verse el consumo (solo para esos dos casos ya que más grandes no pudieron correrse) donde más de la mitad o prácticamente la totalidad del consumo se la lleva este método. La razón es que, al iterar por las distintas máquinas de a pares se realizaba la consulta

Ciclos	Tiempo de CPU (%)
1	12,37
2	22,28
3	31,30
4	37,45
5	40,37
6	41,04
7	41,32
8	39,98
9	33,98

Tab. 4.15: Consumo de sts.nextState en caso de N máquinas B y con datos

de canales (`hasChannel`) para encontrar máquinas candidatas a comunicarse. En casos chicos, con pocas máquinas, era imperceptible este consumo, pero en estos casos donde 100 o 500 ciclos implican 100 o 500 máquinas, la cantidad de llamadas al método se elevan a 251.504.002 (más de 251 millones).

Por otro lado, en las tablas 4.14 y 4.15 podemos ver el consumo del método `nextState` en los casos con datos. Este consumo ronda el 30 % cuando tenemos una única máquina B y llega al 40 % cuando tenemos N máquinas B. Este método es el utilizado para construir un nuevo STS State. El mismo se encarga de reencarnar las variables involucradas en la comunicación que se está realizando en ese momento y de *mantener el conocimiento generado hasta el momento.*

Para realizar esto último se había usado la librería de Cloner (el método `deepClone`) para copiar sin pérdida el conocimiento adquirido. Estas tablas reflejan que el costo de hacer eso era muy elevado.

Para estos problemas se implementaron las siguientes soluciones:

Para evitar la extensa comparación entre listas (se llegó a llamar 62.617.976 de veces el método `equals`) se mantuvo un diccionario (`Map`) de STS States ordenados por la lista de estados. De esta forma para encontrar STS States equivalentes sólo basta con buscar la entrada en el diccionario equivalente a la lista de estados de las máquinas y recorrer un grupo (mucho más) reducido de potenciales STS States equivalentes.

El método `hasChannel` se mantuvo intacto, sin embargo se cambió la implementación de la construcción de las *Communicating Machines.* Al momento del *parseo* de la codificación se mantiene una estructura donde para cada estado de la CM se obtienen las máquinas con las que se comunica. De esta manera sólo se llama al método en los casos necesarios. De esta forma se pasó de más de 251 millones de llamadas a tan solo 3002. Este cambio permitió correr casos igual de grandes que con una única máquina B sin datos.

En cuanto al uso del `deepClone`, mejoramos la implementación de `nextState` manteniendo referencias a estructuras que no se modifican y copiando sólo aquellas que hacían falta.

Las mejoras de estos cambios son evidentes: el método `equals` de listas no se utiliza, mientras que el consumo de `nextState` puede verse en la tabla 4.16.

Con estos cambios, el consumo de CPU pasó a depender en un 80 % del funcionamiento del SMT Solver. El resto del procesamiento se divide generalmente entre los métodos que revisan la equivalencia de conocimiento, las variables ligadas, la consturcción de hipótesis y el parseo. (*knowledgeIsEquivalent(), boundedVars(), parse() y buildHypothesis()*). Sin

Ciclos	Tiempo de CPU (%)
1	0,49
2	2,19
3	2,33
4	2,52
5	1,13
6	0,85
7	0,88

Tab. 4.16: Consumo de sts.nextState en caso de una máquina B y con datos luego de cambios

Ciclos	Tiempo de CPU (%)
100	13,66
500	13,79
1000	13,80
2500	13,93
5000	13,89

Tab. 4.17: Consumo de ContextsAreEquivalent en caso de una máquina B y sin datos

embargo, el consumo de memoria era muy elevado, llegando a generar 22Gb de memoria virtual que dudosamente se requería (sobre todo teniendo en cuenta que el caso de estudio solo envía y recibe variables pero no pone condiciones al respecto).

Se revisó entonces el uso de la librería Z3 y el funcionamiento del SMT Solver limitando la memoria que el solver podía usar para funcionar. El excesivo uso de memoria nos llevó a buscar problemas de memory leaks. Se encontró que era necesario poner a disposición la memoria reservada por el solver una vez que se haya utilizado. De esta forma el Garbage Collector podía disponer de esa memoria si la necesitaba.

Con estos cambios se realizó una tercera implementación y el consumo de memoria bajó drásticamente estabilizándose a costas de bajar levemente el rendimiento del CPU (por el Garbage Collector).

Las tablas de rendimiento completas se pueden encontrar en el apéndice.

4.6.2. Resultados con criterio de equivalencia final

Comencemos por analizar los casos sin datos. En las tablas 4.17 y 4.18 vemos cómo a pesar de no tener condiciones, más del 10 % del consumo del CPU en todos los casos se lo lleva la comparación de contextos. Dentro de este consumo, prácticamente la totalidad (arriba del 95 %) se destina al uso de la biblioteca Z3.

Sorprende que éste sea el consumo del método cuando los contextos solo tienen *true* y las igualdades entre variables en sus conocimientos.

Ciclos	Tiempo de CPU (%)
100	11,48
500	12,80

Tab. 4.18: Consumo de ContextsAreEquivalent en caso de N máquinas B y sin datos

Ciclos	Tiempo de CPU (%)
1	8,03
2	38,78
3	47,85
4	54,95
5	64,93
6	62,28

Tab. 4.19: Consumo de ContextsAreEquivalent en caso de una máquina B y con datos

Ciclos	Tiempo de CPU (%)
1	7,77
2	41,15
3	49,55
4	57,43
5	61,70
6	65,37
7	71,26
8	60,70
9	51,85

Tab. 4.20: Consumo de ContextsAreEquivalent en caso de N máquinas B y con datos

Veamos qué pasa cuando incorporamos datos: En las tablas 4.19 y 4.20 notamos que, descartando los casos de un único ciclo, en promedio más de la mitad del tiempo de ejecución se destina a la comparación de contextos.

De la otra mitad, cerca de un 40 % se divide entre `isContradiction` y `satisfaction`, con menos de un 10 % para el `buildHypothesis`.

Con estos datos podemos concluir que una mejora en la implementación del método o un cambio en el criterio de equivalencia resultan necesarios para mejorar el rendimiento del algoritmo.

Por último, veamos la comparación entre la cantidad de estados generados y el tiempo de CPU. Veamos las tablas 4.21 y 4.22:

De estos datos podemos concluir que el tiempo tiene una relación lineal con la cantidad de estados finales generados por el STS. Podemos suponer a priori que en el medio se descartaron estados que eran equivalentes a otros. Por lo tanto, esta relación nos dice que

Ciclos	# Estados	Tiempo de CPU (ms)
1	5	1171
2	19	1212
3	64	4729
4	197	16118
5	566	61982
6	1543	153427

Tab. 4.21: Consumo de CPU en caso de una máquina B y con datos

Ciclos	# Estados	Tiempo de CPU (ms)
1	5	1236
2	16	2994
3	44	3554
4	112	9812
5	272	26815
6	640	68870
7	1472	233180
8	3328	484164
9	7424	1377621

Tab. 4.22: Consumo de CPU en caso de N máquinas B y con datos

si logramos reducir la cantidad de estados generados entonces lográs reducir el tiempo de ejecución en la misma proporción.

La tabla de resultados completa se puede encontrar en el apéndice.

5. CONCLUSIONES Y TRABAJO FUTURO

5.1. Conclusiones

Vamos a compartir las conclusiones alcanzadas luego de implementar y analizar los casos de estudio.

En primer lugar, aún con aspectos a mejorar y repensar, se pudieron correr los casos de estudios y evaluar sus resultados. La convergencia del algoritmo fue comprobada y esto deja una base suficiente para continuar los estudios.

Tal como se mencionó en los análisis, el criterio de equivalencia y su implementación merecen ser revisados y/o replanteados. No solo por cuestiones de rendimiento sino también por las restricciones que generan en el armado de las máquinas.

Por ejemplo, para el caso de las agencias de vuelos, el uso de variables distintas en cada mensaje genera una gran cantidad de estados. Si pudiésemos unificar el envío de *tomaPrecio* para que salga de un único estado en vez de que haya uno (o más) por cada camino, reduciríamos radicalmente la cantidad de estados del sistema generado pues no se ramificarían las estructuras y se evitaría el crecimiento exponencial. Sin embargo, para eso es necesario evitar los renombres de variables, algo que choca directamente con el criterio de equivalencia propuesto.

En las figuras 5.1 y 5.2 podemos comparar como serían las estructuras genéricas para el caso de 2 agencias si pudiésemos unificar el envío de *tomaPrecio*. A medida que aumentamos la cantidad de agencias la estructura se ramifica más, con la unificación evitamos el crecimiento explosivo.

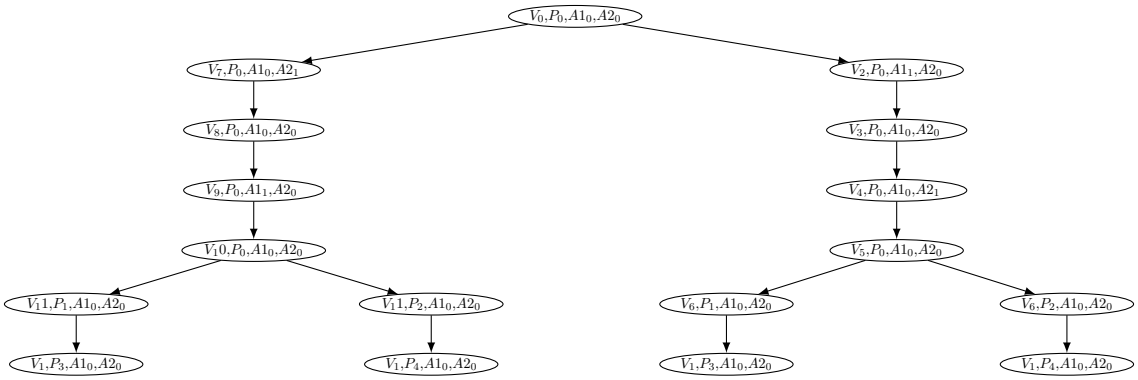

Fig. 5.1: Sistema Sincrónico de Vuelos con tomaPrecio para cada camino

No debemos dejar de mencionar la salvedad respecto a las condiciones usadas sobre los datos. Esta condiciones, aunque útiles para el estudio y para mostrar el alcance de la propuesta, no son muy complejas y por lo tanto no podemos asegurar qué sucede si estas condiciones crecen en complejidad. Será parte del trabajo futuro hacer estudios con ejemplos más complejos.

Por último, los ejemplos utilizados sirvieron como muestra del potencial que tienen estos formalismos sobre los protocolos. El uso de *Guarded Communicating Machines* como formalismo para representar los participantes de una comunicación, gracias al uso de

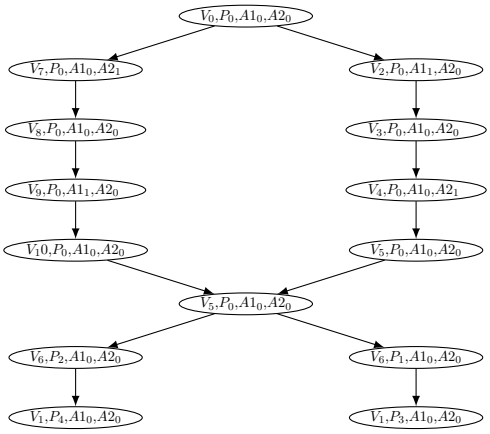

Fig. 5.2: Sistema Sincrónico de Vuelos con tomaPrecio unificado

datos y condiciones sobre los datos, nos permite expresar protocolos más interesantes de una manera bastante sintética y en la misma línea que los formalismos usados hasta el momento.

Queda pendiente estudiar cuál es el poder expresivo de esta herramienta tal como está ahora. Poder realizar una comparación exhaustiva entre esta propuesta de formalismo y las ya existentes para corroborar si se trata de un mayor poder expresivo o una manera más sintética de expresar protocolos.

5.2. Trabajo Futuro

En este apartado dejamos algunas puntas para el trabajo futuro a partir de lo realizado hasta ahora, orientado a:

- Consolidar un criterio de equivalencia coherente, eficiente y útil.
- Continuar el desarrollo de la herramienta implementada en este trabajo para futuras investigaciones y análisis, mejorando su interfaz y agregando herramientas para el estudio de protocolos.
- En la Sec. 5.2.4 se proponen algunas puntas para profundizar los análisis hechos en este trabajo con estudios que exceden el objetivo de esta tesis.

5.2.1. Criterio de equivalencia

El criterio de equivalencia presentó serias limitaciones en algunos casos debido a la explosión en la cantidad de estados. Por lo tanto habría que hacer un esfuerzo mayor por estudiar qué pasa en casos reales (por ejemplo, protocolos estándar como los usados en [11] pero con datos).

Vimos también que encontrarle solución al renombre de variables reduciría el tamaño de la estructura y esto depende del criterio de equivalencia. Pensar soluciones a ese problema a partir de este criterio puede mejorar notoriamente el rendimiento.

La profundización en el análisis del método `contextsAreEquivalent` para encontrar mejoras algorítmicas o incluso la reformulación del criterio, son alternativas posibles para resolver los problemas encontrados en los análisis hechos.

5.2.2. BPEL

Una investigación que formó parte de este trabajo fue el estudio del lenguaje BPEL[1] como alternativa para llevar a la práctica la codificación de los protocolos usando los estándares de Web Services. BPEL es un lenguaje estandarizado para la composición de Web Services basado en XML.

Si la herramienta pudiera utilizar BPEL o una extensión de BPEL podría incentivar a que la gente que trabaja con este lenguaje pruebe la herramienta.

El desarrollo de casos de estudio para formalizar protocolos y analizar el funcionamiento del algoritmo en un entorno cliente-servidor sería un aporte más que interesante para alcanzar ese objetivo. Para esto es necesario dar una formalización de BPEL en términos de GCMs y eso requiere una investigación, estudio y desarrollo en sí mismo.

5.2.3. GUI

Si bien la interfaz gráfica implementada cumple el objetivo pretendido, la mejora de la interfaz puede ayudar al estudio y desarrollo del problema. Actualmente se dificulta la visualización de STS grandes y contar con funcionalidades que permitan navegar por el Sistema Sincrónico de manera más fluida e incluso diseñar las GCMs en el mismo entorno serían de gran utilidad.

5.2.4. Rendimiento

Los análisis de rendimiento hechos para este trabajo no son totales, su objetivo fue dar una noción de la capacidad y de las limitaciones principales del algoritmo y mostrar su funcionamiento e implementación. Un estudio exhaustivo de la complejidad algorítmica es necesario para mostrar el potencial real de la propuesta desarrollada. Este análisis debe incluir en sus estudios la comparación entre distintos *solvers* y un análisis bien discriminado de la complejidad estructural del algoritmo y de *solving*.

5.2.5. Decisiones

Queda pendiente por resolver cómo determinar si las máquinas con datos son compatibles. El criterio utilizado para máquinas sin datos no aplica para estos casos pues, por ejemplo en las máquinas sin datos no podés eliminar ningún *choice*, mientras que en las máquinas con datos tiene sentido que los *choices* que no se van a poder realizar porque no se cumplen las condiciones se eliminen y eso no significa, necesariamente que las máquinas no sean compatibles.

Otro punto a tener en cuenta es que en el criterio de compatibilidad de [11] sólo se consideran CMs determinísticas mientras que muchas de las máquinas consideradas en este trabajo son no determinísticas pero a raíz del uso de condiciones se vuelven determinísticas.

[1] https://es.wikipedia.org/wiki/WS-BPEL

Apéndice

Apéndices

A. CONTEXTOS DE SISTEMAS SINCRÓNICOS

A.1. Contextos de caso Facebook con login correcto

①

usrA1= 0 // {}
usrA2= 0 // {}
appIdC= 0 // {}
tokenW1= 0 // {}
tokenV1= 0 // {}
appIdA1= 0 // {}
tokenW2= 0 // {}
appIdA2= 0 // {}
tokenV2= 0 // {}
pwdA1= 0 // {}
pwdA2= 0 // {}
pwdC= 0 // {}
tokenC1= 0 // {}
tokenC2= 0 // {}
usrC= 0 // {}
tokenA1= 0 // {}
tokenA2= 0 // {}

②

usrA1= 1 // {$usrA1_1$: $usrC_1$ = $usrA1_1$)}
usrA2= 0 // {}
appIdC= 1 // {$appIdC_1$: ($appIdC_1$ = $appIdA1_1$) & ($usrC_1$ = 1) & ($pwdC_1$ = $usrC_1$+1) & ($appIdC_1$ = $usrC_1$+2)}
tokenW1= 0 // {}
tokenV1= 0 // {}
appIdA1= 1 // {$appIdA1_1$: ($appIdC_1$ = $appIdA1_1$)}
tokenW2= 0 // {}
appIdA2= 0 // {}
tokenV2= 0 // {}
pwdA1= 1 // {$pwdA1_1$: ($pwdC_1$ = $pwdA1_1$)}
pwdA2= 0 // {}
pwdC= 1 // {$pwdC_1$: ($pwdC_1$ = $pwdA1_1$) & ($usrC_1$ = 1) & ($pwdC_1$ = $usrC_1$+1) & ($appIdC_1$ = $usrC_1$+2)}
tokenC1= 0 // {}
tokenC2= 0 // {}
usrC= 1 // {$usrC_1$: ($usrC_1$ = $usrA1_1$) & ($usrC_1$ = 1) & ($pwdC_1$ = $usrC_1$+1) & ($appIdC_1$ = $usrC_1$+2)}
tokenA1= 0 // {}
tokenA2= 0 // {}

③

usrA1= 1 // {$usrA1_1$: $usrC_1$ = $usrA1_1$)}
usrA2= 0 // {}
appIdC= 1 // {$appIdC_1$: ($appIdC_1$ = $appIdA1_1$) & ($usrC_1$ = 1) & ($pwdC_1$ = $usrC_1$+1) & ($appIdC_1$ = $usrC_1$+2)}
tokenW1= 0 // {}
tokenV1= 2 // {$tokenV1_2$: ($tokenA1_2$ = $tokenV1_2$)}
appIdA1= 1 // {$appIdA1_1$: ($appIdC_1$ = $appIdA1_1$)}
tokenW2= 0 // {}
appIdA2= 0 // {}
tokenV2= 0 // {}
pwdA1= 1 // {$pwdA1_1$: ($pwdC_1$ = $pwdA1_1$)}
pwdA2= 0 // {}
pwdC= 1 // {$pwdC_1$: ($pwdC_1$ = $pwdA1_1$) & ($usrC_1$ = 1) & ($pwdC_1$ = $usrC_1$+1) & ($appIdC_1$ = $usrC_1$+2)}
tokenC1= 0 // {}
tokenC2= 0 // {}
usrC= 1 // {$usrC_1$: ($usrC_1$ = $usrA1_1$) & ($usrC_1$ = 1) & ($pwdC_1$ = $usrC_1$+1) & ($appIdC_1$ = $usrC_1$+2)}
tokenA1= 2 // {$tokenA1_2$: ($tokenA1_2$ = $tokenV1_2$) & ($tokenA1_2$ = 1234)}
tokenA2= 0 // {}

④

usrA1= 1 // {$usrA1_1$: $usrC_1$ = $usrA1_1$)}
usrA2= 0 // {}
appIdC= 1 // {$appIdC_1$: ($appIdC_1$ = $appIdA1_1$) & ($usrC_1$ = 1) & ($pwdC_1$ = $usrC_1$+1) & ($appIdC_1$ = $usrC_1$+2)}
tokenW1= 0 // {}
tokenV1= 2 // {$tokenV1_2$: ($tokenA1_2$ = $tokenV1_2$)}
appIdA1= 1 // {$appIdA1_1$: ($appIdC_1$ = $appIdA1_1$)}
tokenW2= 0 // {}
appIdA2= 0 // {}
tokenV2= 0 // {}
pwdA1= 1 // {$pwdA1_1$: ($pwdC_1$ = $pwdA1_1$)}
pwdA2= 0 // {}
pwdC= 1 // {$pwdC_1$: ($pwdC_1$ = $pwdA1_1$) & ($usrC_1$ = 1) & ($pwdC_1$ = $usrC_1$+1) & ($appIdC_1$ = $usrC_1$+2)}
tokenC1= 3 // {$tokenC1_3$: ($tokenA2_3$ = $tokenC1_3$)}
tokenC2= 0 // {}
usrC= 1 // {$usrC_1$: ($usrC_1$ = $usrA1_1$) & ($usrC_1$ = 1) & ($pwdC_1$ = $usrC_1$+1) & ($appIdC_1$ = $usrC_1$+2)}
tokenA1= 2 // {$tokenA1_2$: ($tokenA1_2$ = $tokenV1_2$) & ($tokenA1_2$ = 1234)}
tokenA2= 3 // {$tokenA2_3$: ($tokenA2_3$ = $tokenC1_3$) & ($tokenA2_3$ = 1234)}

(5)

usrA1= 1 // {$usrA1_1$: $usrC_1$ = $usrA1_1$)}
usrA2= 0 // {}
appIdC= 1 // {$appIdC_1$: ($appIdC_1$ = $appIdA1_1$) & ($usrC_1$ = 1) & ($pwdC_1$ = $usrC_1$+1) & ($appIdC_1$ = $usrC_1$+2)}
tokenW1= 4 // {$tokenW1_4$: ($tokenC2_4$ = $tokenW1_4$)}
tokenV1= 2 // {$tokenV1_2$: ($tokenA1_2$ = $tokenV1_2$)}
appIdA1= 1 // {$appIdA1_1$: ($appIdC_1$ = $appIdA1_1$)}
tokenW2= 0 // {}
appIdA2= 0 // {}
tokenV2= 0 // {}
pwdA1= 1 // {$pwdA1_1$: ($pwdC_1$ = $pwdA1_1$)}
pwdA2= 0 // {}
pwdC= 1 // {$pwdC_1$: ($pwdC_1$ = $pwdA1_1$) & ($usrC_1$ = 1) & ($pwdC_1$ = $usrC_1$+1) & ($appIdC_1$ = $usrC_1$+2)}
tokenC1= 3 // {$tokenC1_3$: ($tokenA2_3$ = $tokenC1_3$)}
tokenC2= 4 // {$tokenC2_4$: ($tokenC2_4$ = $tokenW1_4$) & ($tokenC2_4$ = $tokenC1_3$)}
usrC= 1 // {$usrC_1$: ($usrC_1$ = $usrA1_1$) & ($usrC_1$ = 1) & ($pwdC_1$ = $usrC_1$+1) & ($appIdC_1$ = $usrC_1$+2)}
tokenA1= 2 // {$tokenA1_2$: ($tokenA1_2$ = $tokenV1_2$) & ($tokenA1_2$ = 1234)}
tokenA2= 3 // {$tokenA2_3$: ($tokenA2_3$ = $tokenC1_3$) & ($tokenA2_3$ = 1234)}

(6)

usrA1= 1 // {$usrA1_1$: $usrC_1$ = $usrA1_1$)}
usrA2= 0 // {}
appIdC= 1 // {$appIdC_1$: ($appIdC_1$ = $appIdA1_1$) & ($usrC_1$ = 1) & ($pwdC_1$ = $usrC_1$+1) & ($appIdC_1$ = $usrC_1$+2)}
tokenW1= 4 // {$tokenW1_4$: ($tokenC2_4$ = $tokenW1_4$) & ($tokenW1_4$ = $tokenW2_5$)}
tokenV1= 2 // {$tokenV1_2$: ($tokenA1_2$ = $tokenV1_2$)}
appIdA1= 1 // {$appIdA1_1$: ($appIdC_1$ = $appIdA1_1$)}
tokenW2= 5 // {$tokenW2_5$: ($tokenW2_5$ = $tokenV2_5$) & ($tokenW1_4$ = $tokenW2_5$)}
appIdA2= 0 // {}
tokenV2= 5 // {$tokenV2_5$: ($tokenW2_5$ = $tokenV2_5$)}
pwdA1= 1 // {$pwdA1_1$: ($pwdC_1$ = $pwdA1_1$)}
pwdA2= 0 // {}
pwdC= 1 // {$pwdC_1$: ($pwdC_1$ = $pwdA1_1$) & ($usrC_1$ = 1) & ($pwdC_1$ = $usrC_1$+1) & ($appIdC_1$ = $usrC_1$+2)}
tokenC1= 3 // {$tokenC1_3$: ($tokenA2_3$ = $tokenC1_3$)}
tokenC2= 4 // {$tokenC2_4$: ($tokenC2_4$ = $tokenW1_4$) & ($tokenC2_4$ = $tokenC1_3$)}
usrC= 1 // {$usrC_1$: ($usrC_1$ = $usrA1_1$) & ($usrC_1$ = 1) & ($pwdC_1$ = $usrC_1$+1) & ($appIdC_1$ = $usrC_1$+2)}
tokenA1= 2 // {$tokenA1_2$: ($tokenA1_2$ = $tokenV1_2$) & ($tokenA1_2$ = 1234)}
tokenA2= 3 // {$tokenA2_3$: ($tokenA2_3$ = $tokenC1_3$) & ($tokenA2_3$ = 1234)}

(7)

usrA1= 1 // {$usrA1_1$: $usrC_1$ = $usrA1_1$)}
usrA2= 0 // {}
appIdC= 1 // {$appIdC_1$: ($appIdC_1$ = $appIdA1_1$) & ($usrC_1$ = 1) & ($pwdC_1$ = $usrC_1$+1) & ($appIdC_1$ = $usrC_1$+2)}
tokenW1= 4 // {$tokenW1_4$: ($tokenC2_4$ = $tokenW1_4$) & ($tokenW1_4$ = $tokenW2_5$)}
tokenV1= 2 // {$tokenV1_2$: ($tokenA1_2$ = $tokenV1_2$) & ($tokenV1_2$ = $tokenV2_5$)}
appIdA1= 1 // {$appIdA1_1$: ($appIdC_1$ = $appIdA1_1$)}
tokenW2= 5 // {$tokenW2_5$: ($tokenW2_5$ = $tokenV2_5$) & ($tokenW1_4$ = $tokenW2_5$)}
appIdA2= 0 // {}
tokenV2= 5 // {$tokenV2_5$: ($tokenW2_5$ = $tokenV2_5$) & ($tokenV1_2$ = $tokenV2_5$)}
pwdA1= 1 // {$pwdA1_1$: ($pwdC_1$ = $pwdA1_1$)}
pwdA2= 0 // {}
pwdC= 1 // {$pwdC_1$: ($pwdC_1$ = $pwdA1_1$) & ($usrC_1$ = 1) & ($pwdC_1$ = $usrC_1$+1) & ($appIdC_1$ = $usrC_1$+2)}
tokenC1= 3 // {$tokenC1_3$: ($tokenA2_3$ = $tokenC1_3$)}
tokenC2= 4 // {$tokenC2_4$: ($tokenC2_4$ = $tokenW1_4$) & ($tokenC2_4$ = $tokenC1_3$)}
usrC= 1 // {$usrC_1$: ($usrC_1$ = $usrA1_1$) & ($usrC_1$ = 1) & ($pwdC_1$ = $usrC_1$+1) & ($appIdC_1$ = $usrC_1$+2)}
tokenA1= 2 // {$tokenA1_2$: ($tokenA1_2$ = $tokenV1_2$) & ($tokenA1_2$ = 1234)}
tokenA2= 3 // {$tokenA2_3$: ($tokenA2_3$ = $tokenC1_3$) & ($tokenA2_3$ = 123)}

A.2. Contextos de caso Facebook con login incorrecto

①

```
usrA1= 0 // {}
usrA2= 0 // {}
appIdC= 0 // {}
tokenW1= 0 // {}
tokenV1= 0 // {}
appIdA1= 0 // {}
tokenW2= 0 // {}
appIdA2= 0 // {}
tokenV2= 0 // {}
pwdA1= 0 // {}
pwdA2= 0 // {}
pwdC= 0 // {}
tokenC1= 0 // {}
tokenC2= 0 // {}
usrC= 0 // {}
tokenA1= 0 // {}
tokenA2= 0 // {}
```

②

```
usrA1= 0 // {}
usrA2= 1 // {usrA2_1 : usrC_1 = usrA2_1)}
appIdC= 1 // {appIdC_1 : (appIdC_1 = appIdA1_1) & ~((usrC_1 = 1) & (pwdC_1 = usrC_1+1) & (
    appIdC_1 = usrC_1+2))}
tokenW1= 0 // {}
tokenV1= 0 // {}
appIdA1= 0 // {}
tokenW2= 0 // {}
appIdA2= 1 // {appIdA2_1 : (appIdC2_1 = appIdA2_1)}
tokenV2= 0 // {}
pwdA1= 0 // {}
pwdA2= 1 // {pwdA2_1 : (pwdC_1 = pwdA2_1)}
pwdC= 1 // {pwdC_1 : (pwdC_1 = pwdA1_1) & ~((usrC_1 = 1) & (pwdC_1 = usrC_1+1) & (appIdC_1 =
     usrC_1+2))}
tokenC1= 0 // {}
tokenC2= 0 // {}
usrC= 1 // {usrC_1 : (usrC_1 = usrA1_1) & ~((usrC_1 = 1) & (pwdC_1 = usrC_1+1) & (appIdC_1 =
    usrC_1+2))}
tokenA1= 0 // {}
tokenA2= 0 // {}
```

A.3. Contextos de caso Facebook con token correcto

①

```
usrA1= 0 // {}
usrA2= 0 // {}
appIdC= 0 // {}
tokenW1= 0 // {}
tokenV1= 0 // {}
appIdA1= 0 // {}
tokenW2= 0 // {}
appIdA2= 0 // {}
tokenV2= 0 // {}
pwdA1= 0 // {}
pwdA2= 0 // {}
pwdC= 0 // {}
tokenC1= 0 // {}
tokenC2= 0 // {}
usrC= 0 // {}
tokenA1= 0 // {}
tokenA2= 0 // {}
```

②

```
usrA1= 1 // {usrA1_1 : usrC_1 = usrA1_1)}
usrA2= 0 // {}
appIdC= 1 // {appIdC_1 : (appIdC_1 = appIdA1_1) & (usrC_1 = 1) & (pwdC_1 = usrC_1+1) & (
    appIdC_1 = usrC_1+2)}
tokenW1= 0 // {}
tokenV1= 0 // {}
appIdA1= 1 // {appIdA1_1 : (appIdC_1 = appIdA1_1)}
tokenW2= 0 // {}
appIdA2= 0 // {}
tokenV2= 0 // {}
pwdA1= 1 // {pwdA1_1 : (pwdC_1 = pwdA1_1)}
pwdA2= 0 // {}
pwdC= 1 // {pwdC_1 : (pwdC_1 = pwdA1_1) & (usrC_1 = 1) & (pwdC_1 = usrC_1+1) & (appIdC_1 =
    usrC_1+2)}
tokenC1= 0 // {}
tokenC2= 0 // {}
usrC= 1 // {usrC_1 : (usrC_1 = usrA1_1) & (usrC_1 = 1) & (pwdC_1 = usrC_1+1) & (appIdC_1 =
    usrC_1+2)}
tokenA1= 0 // {}
tokenA2= 0 // {}
```

③

```
usrA1= 1 // {usrA1_1 : usrC_1 = usrA1_1)}
usrA2= 0 // {}
appIdC= 1 // {appIdC_1 : (appIdC_1 = appIdA1_1) & (usrC_1 = 1) & (pwdC_1 = usrC_1+1) & (
    appIdC_1 = usrC_1+2)}
tokenW1= 0 // {}
tokenV1= 2 // {tokenV1_2 : (tokenA1_2 = tokenV1_2)}
appIdA1= 1 // {appIdA1_1 : (appIdC_1 = appIdA1_1)}
tokenW2= 0 // {}
appIdA2= 0 // {}
tokenV2= 0 // {}
pwdA1= 1 // {pwdA1_1 : (pwdC_1 = pwdA1_1)}
pwdA2= 0 // {}
pwdC= 1 // {pwdC_1 : (pwdC_1 = pwdA1_1) & (usrC_1 = 1) & (pwdC_1 = usrC_1+1) & (appIdC_1 =
    usrC_1+2)}
tokenC1= 0 // {}
tokenC2= 0 // {}
usrC= 1 // {usrC_1 : (usrC_1 = usrA1_1) & (usrC_1 = 1) & (pwdC_1 = usrC_1+1) & (appIdC_1 =
    usrC_1+2)}
tokenA1= 2 // {tokenA1_2 : (tokenA1_2 = tokenV1_2) & (tokenA1_2 = 1234)}
tokenA2= 0 // {}
```

④

```
usrA1= 1 // {usrA1_1 : usrC_1 = usrA1_1)}
usrA2= 0 // {}
appIdC= 1 // {appIdC_1 : (appIdC_1 = appIdA1_1) & (usrC_1 = 1) & (pwdC_1 = usrC_1+1) & (
    appIdC_1 = usrC_1+2)}
tokenW1= 0 // {}
tokenV1= 2 // {tokenV1_2 : (tokenA1_2 = tokenV1_2)}
appIdA1= 1 // {appIdA1_1 : (appIdC_1 = appIdA1_1)}
tokenW2= 0 // {}
appIdA2= 0 // {}
tokenV2= 0 // {}
pwdA1= 1 // {pwdA1_1 : (pwdC_1 = pwdA1_1)}
pwdA2= 0 // {}
pwdC= 1 // {pwdC_1 : (pwdC_1 = pwdA1_1) & (usrC_1 = 1) & (pwdC_1 = usrC_1+1) & (appIdC_1 =
    usrC_1+2)}
tokenC1= 3 // {tokenC1_3 : (tokenA2_3 = tokenC1_3)}
tokenC2= 0 // {}
usrC= 1 // {usrC_1 : (usrC_1 = usrA1_1) & (usrC_1 = 1) & (pwdC_1 = usrC_1+1) & (appIdC_1 =
    usrC_1+2)}
tokenA1= 2 // {tokenA1_2 : (tokenA1_2 = tokenV1_2) & (tokenA1_2 = 1234)}
tokenA2= 3 // {tokenA2_3 : (tokenA2_3 = tokenC1_3) & (tokenA2_3 = 123)}
```

(5)

usrA1= 1 // {$usrA1_1$: $usrC_1$ = $usrA1_1$)}
usrA2= 0 // {}
appIdC= 1 // {$appIdC_1$: ($appIdC_1$ = $appIdA1_1$) & ($usrC_1$ = 1) & ($pwdC_1$ = $usrC_1$+1) & ($appIdC_1$ = $usrC_1$+2)}
tokenW1= 4 // {$tokenW1_4$: ($tokenC2_4$ = $tokenW1_4$)}
tokenV1= 2 // {$tokenV1_2$: ($tokenA1_2$ = $tokenV1_2$)}
appIdA1= 1 // {$appIdA1_1$: ($appIdC_1$ = $appIdA1_1$)}
tokenW2= 0 // {}
appIdA2= 0 // {}
tokenV2= 0 // {}
pwdA1= 1 // {$pwdA1_1$: ($pwdC_1$ = $pwdA1_1$)}
pwdA2= 0 // {}
pwdC= 1 // {$pwdC_1$: ($pwdC_1$ = $pwdA1_1$) & ($usrC_1$ = 1) & ($pwdC_1$ = $usrC_1$+1) & ($appIdC_1$ = $usrC_1$+2)}
tokenC1= 3 // {$tokenC1_3$: ($tokenA2_3$ = $tokenC1_3$)}
tokenC2= 4 // {$tokenC2_4$: ($tokenC2_4$ = $tokenW1_4$) & ($tokenC2_4$ = $tokenC1_3$)}
usrC= 1 // {$usrC_1$: ($usrC_1$ = $usrA1_1$) & ($usrC_1$ = 1) & ($pwdC_1$ = $usrC_1$+1) & ($appIdC_1$ = $usrC_1$+2)}
tokenA1= 2 // {$tokenA1_2$: ($tokenA1_2$ = $tokenV1_2$) & ($tokenA1_2$ = 1234)}
tokenA2= 3 // {$tokenA2_3$: ($tokenA2_3$ = $tokenC1_3$) & ($tokenA2_3$ = 123)}

(6)

usrA1= 1 // {$usrA1_1$: $usrC_1$ = $usrA1_1$)}
usrA2= 0 // {}
appIdC= 1 // {$appIdC_1$: ($appIdC_1$ = $appIdA1_1$) & ($usrC_1$ = 1) & ($pwdC_1$ = $usrC_1$+1) & ($appIdC_1$ = $usrC_1$+2)}
tokenW1= 4 // {$tokenW1_4$: ($tokenC2_4$ = $tokenW1_4$) & ($tokenW1_4$ = $tokenW2_5$)}
tokenV1= 2 // {$tokenV1_2$: ($tokenA1_2$ = $tokenV1_2$)}
appIdA1= 1 // {$appIdA1_1$: ($appIdC_1$ = $appIdA1_1$)}
tokenW2= 5 // {$tokenW2_5$: ($tokenW2_5$ = $tokenV2_5$) & ($tokenW1_4$ = $tokenW2_5$)}
appIdA2= 0 // {}
tokenV2= 5 // {$tokenV2_5$: ($tokenW2_5$ = $tokenV2_5$)}
pwdA1= 1 // {$pwdA1_1$: ($pwdC_1$ = $pwdA1_1$)}
pwdA2= 0 // {}
pwdC= 1 // {$pwdC_1$: ($pwdC_1$ = $pwdA1_1$) & ($usrC_1$ = 1) & ($pwdC_1$ = $usrC_1$+1) & ($appIdC_1$ = $usrC_1$+2)}
tokenC1= 3 // {$tokenC1_3$: ($tokenA2_3$ = $tokenC1_3$)}
tokenC2= 4 // {$tokenC2_4$: ($tokenC2_4$ = $tokenW1_4$) & ($tokenC2_4$ = $tokenC1_3$)}
usrC= 1 // {$usrC_1$: ($usrC_1$ = $usrA1_1$) & ($usrC_1$ = 1) & ($pwdC_1$ = $usrC_1$+1) & ($appIdC_1$ = $usrC_1$+2)}
tokenA1= 2 // {$tokenA1_2$: ($tokenA1_2$ = $tokenV1_2$) & ($tokenA1_2$ = 1234)}
tokenA2= 3 // {$tokenA2_3$: ($tokenA2_3$ = $tokenC1_3$) & ($tokenA2_3$ = 123)}

(7)

usrA1= 1 // {$usrA1_1$: $usrC_1$ = $usrA1_1$)}
usrA2= 0 // {}
appIdC= 1 // {$appIdC_1$: ($appIdC_1$ = $appIdA1_1$) & ($usrC_1$ = 1) & ($pwdC_1$ = $usrC_1$+1) & ($appIdC_1$ = $usrC_1$+2)}
tokenW1= 4 // {$tokenW1_4$: ($tokenC2_4$ = $tokenW1_4$) & ($tokenW1_4$ = $tokenW2_5$)}
tokenV1= 2 // {$tokenV1_2$: ($tokenA1_2$ = $tokenV1_2$) & ($tokenV1_2$ ≠ $tokenV2_5$)}
appIdA1= 1 // {$appIdA1_1$: ($appIdC_1$ = $appIdA1_1$)}
tokenW2= 5 // {$tokenW2_5$: ($tokenW2_5$ = $tokenV2_5$) & ($tokenW1_4$ = $tokenW2_5$)}
appIdA2= 0 // {}
tokenV2= 5 // {$tokenV2_5$: ($tokenW2_5$ = $tokenV2_5$) & ($tokenV1_2$ ≠ $tokenV2_5$)}
pwdA1= 1 // {$pwdA1_1$: ($pwdC_1$ = $pwdA1_1$)}
pwdA2= 0 // {}
pwdC= 1 // {$pwdC_1$: ($pwdC_1$ = $pwdA1_1$) & ($usrC_1$ = 1) & ($pwdC_1$ = $usrC_1$+1) & ($appIdC_1$ = $usrC_1$+2)}
tokenC1= 3 // {$tokenC1_3$: ($tokenA2_3$ = $tokenC1_3$)}
tokenC2= 4 // {$tokenC2_4$: ($tokenC2_4$ = $tokenW1_4$) & ($tokenC2_4$ = $tokenC1_3$)}
usrC= 1 // {$usrC_1$: ($usrC_1$ = $usrA1_1$) & ($usrC_1$ = 1) & ($pwdC_1$ = $usrC_1$+1) & ($appIdC_1$ = $usrC_1$+2)}
tokenA1= 2 // {$tokenA1_2$: ($tokenA1_2$ = $tokenV1_2$) & ($tokenA1_2$ = 1234)}
tokenA2= 3 // {$tokenA2_3$: ($tokenA2_3$ = $tokenC1_3$) & ($tokenA2_3$ = 123)}

(8)

usrA1= 6 // {$usrA1_6$: $usrC_6$ = $usrA1_6$), $usrA1_1$: $usrC_1$ = $usrA1_1$)}
usrA2= 0 // {}
appIdC= 6 // {$appIdC_6$: ($appIdC_6$ = $appIdA1_6$) & ($usrC_6$ = 1) & ($pwdC_6$ = $usrC_6$+1) & ($appIdC_6$ = $usrC_6$+2), $appIdC_1$: ($appIdC_1$ = $appIdA1_1$) & ($usrC_1$ = 1) & ($pwdC_1$ = $usrC_1$+1) & ($appIdC_1$ = $usrC_1$+2)}
tokenW1= 4 // {$tokenW1_4$: ($tokenC2_4$ = $tokenW1_4$) & ($tokenW1_4$ = $tokenW2_5$)}
tokenV1= 2 // {$tokenV1_2$: ($tokenA1_2$ = $tokenV1_2$) & ($tokenV1_2$ ≠ $tokenV2_5$)}
appIdA1= 6 // {$appIdA1_6$: ($appIdC_6$ = $appIdA1_6$), $appIdA1_1$: ($appIdC_1$ = $appIdA1_1$)}
tokenW2= 5 // {$tokenW2_5$: ($tokenW2_5$ = $tokenV2_5$) & ($tokenW1_4$ = $tokenW2_5$)}
appIdA2= 0 // {}
tokenV2= 5 // {$tokenV2_5$: ($tokenW2_5$ = $tokenV2_5$) & ($tokenV1_2$ ≠ $tokenV2_5$)}
pwdA1= 6 // {$pwdA1_6$: ($pwdC_6$ = $pwdA1_6$), $pwdA1_1$: ($pwdC_1$ = $pwdA1_1$)}
pwdA2= 0 // {}
pwdC= 6 // {$pwdC_6$: ($pwdC_6$ = $pwdA1_6$) & ($usrC_6$ = 1) &($pwdC_6$ = 2) & ($appIdC_6$ = 3), $pwdC_1$: ($pwdC_1$ = $pwdA1_1$) & ($usrC_1$ = 1) & ($pwdC_1$ = $usrC_1$+1) & ($appIdC_1$ = $usrC_1$+2)}
tokenC1= 3 // {$tokenC1_3$: ($tokenA2_3$ = $tokenC1_3$)}
tokenC2= 4 // {$tokenC2_4$: ($tokenC2_4$ = $tokenW1_4$) & ($tokenC2_4$ = $tokenC1_3$)}
usrC= 6 // {$usrC_6$: ($usrC_6$ = $usrA1_6$) & ($usrC_6$ = 1) & ($pwdC_6$ = $usrC_6$+1) & ($appIdC_6$ = $usrC_6$+2), $usrC_1$: ($usrC_1$ = $usrA1_1$) & ($usrC_1$ = 1) & ($pwdC_1$ = $usrC_1$+1) & ($appIdC_1$ = $usrC_1$+2)}
tokenA1= 2 // {$tokenA1_2$: ($tokenA1_2$ = $tokenV1_2$) & ($tokenA1_2$ = 1234)}
tokenA2= 3 // {$tokenA2_3$: ($tokenA2_3$ = $tokenC1_3$) & ($tokenA2_3$ = 123)}

(9)

```
usrA1= 6 // {usrA1_6 : usrC_6 = usrA1_6), usrA1_1 : usrC_1 = usrA1_1)}
usrA2= 0 // {}
appIdC= 6 // {appIdC_6 : (appIdC_6 = appIdA1_6) & (usrC_6 = 1) & (pwdC_6 = usrC_6+1) & (
    appIdC_6 = usrC_6+2), appIdC_1 : (appIdC_1 = appIdA1_1) & (usrC_1 = 1) & (pwdC_1 = usrC_1+1)
     & (appIdC_1 = usrC_1+2)}
tokenW1= 4 // {tokenW1_4 : (tokenC2_4 = tokenW1_4) & (tokenW1_4 = tokenW2_5)}
tokenV1= 7 // {tokenV1_7 : (tokenA1_7 = tokenV1_7), tokenV1_2 : (tokenA1_2 = tokenV1_2) & (
    tokenV1_2 ≠ tokenV2_5)}
appIdA1= 6 // {appIdA1_6 : (appIdC_6 = appIdA1_6), appIdA1_1 : (appIdC_1 = appIdA1_1)}
tokenW2= 5 // {tokenW2_5 : (tokenW2_5 = tokenV2_5) & (tokenW1_4 = tokenW2_5)}
appIdA2= 0 // {}
tokenV2= 5 // {tokenV2_5 : (tokenW2_5 = tokenV2_5) & (tokenV1_2 ≠ tokenV2_5)}
pwdA1= 6 // {pwdA1_6 : (pwdC_6 = pwdA1_6), pwdA1_1 : (pwdC_1 = pwdA1_1)}
pwdA2= 0 // {}
pwdC= 6 // {pwdC_6 : (pwdC_6 = pwdA1_6) & (usrC_6 = 1) &(pwdC_6 = 2) & (appIdC_6 = 3),
    pwdC_1 : (pwdC_1 = pwdA1_1) & (usrC_1 = 1) & (pwdC_1 = usrC_1+1) & (appIdC_1 = usrC_1+2)}
tokenC1= 3 // {tokenC1_3 : (tokenA2_3 = tokenC1_3)}
tokenC2= 4 // {tokenC2_4 : (tokenC2_4 = tokenW1_4) & (tokenC2_4 = tokenC1_3)}
usrC= 6 // {usrC_6 : (usrC_6 = usrA1_6) & (usrC_6 = 1) & (pwdC_6 = usrC_6+1) & (appIdC_6 =
    usrC_6+2), usrC_1 : (usrC_1 = usrA1_1) & (usrC_1 = 1) & (pwdC_1 = usrC_1+1) & (appIdC_1 =
    usrC_1+2)}
tokenA1= 7 // {tokenA1_7 : (tokenA1_7 = tokenV1_7) & (tokenA1_7 = 1234), tokenA1_2 : (tokenA1_2
     = tokenV1_2) & (tokenA1_2 = 1234)}
tokenA2= 3 // {tokenA2_3 : (tokenA2_3 = tokenC1_3) & (tokenA2_3 = 123)}
```

(10)

```
usrA1= 6 // {usrA1_6 : usrC_6 = usrA1_6), usrA1_1 : usrC_1 = usrA1_1)}
usrA2= 0 // {}
appIdC= 6 // {appIdC_6 : (appIdC_6 = appIdA1_6) & (usrC_6 = 1) & (pwdC_6 = usrC_6+1) & (
    appIdC_6 = usrC_6+2), appIdC_1 : (appIdC_1 = appIdA1_1) & (usrC_1 = 1) & (pwdC_1 = usrC_1+1)
     & (appIdC_1 = usrC_1+2)}
tokenW1= 4 // {tokenW1_4 : (tokenC2_4 = tokenW1_4) & (tokenW1_4 = tokenW2_5)}
tokenV1= 7 // {tokenV1_7 : (tokenA1_7 = tokenV1_7), tokenV1_2 : (tokenA1_2 = tokenV1_2) & (
    tokenV1_2 ≠ tokenV2_5)}
appIdA1= 6 // {appIdA1_6 : (appIdC_6 = appIdA1_6), appIdA1_1 : (appIdC_1 = appIdA1_1)}
tokenW2= 5 // {tokenW2_5 : (tokenW2_5 = tokenV2_5) & (tokenW1_4 = tokenW2_5)}
appIdA2= 0 // {}
tokenV2= 5 // {tokenV2_5 : (tokenW2_5 = tokenV2_5) & (tokenV1_2 ≠ tokenV2_5)}
pwdA1= 6 // {pwdA1_6 : (pwdC_6 = pwdA1_6), pwdA1_1 : (pwdC_1 = pwdA1_1)}
pwdA2= 0 // {}
pwdC= 6 // {pwdC_6 : (pwdC_6 = pwdA1_6) & (usrC_6 = 1) &(pwdC_6 = 2) & (appIdC_6 = 3),
    pwdC_1 : (pwdC_1 = pwdA1_1) & (usrC_1 = 1) & (pwdC_1 = usrC_1+1) & (appIdC_1 = usrC_1+2)}
tokenC1= 8 // {tokenC1_8 : (tokenA2_8 = tokenC1_8), tokenC1_3 : (tokenA2_3 = tokenC1_3)}
tokenC2= 4 // {tokenC2_4 : (tokenC2_4 = tokenW1_4) & (tokenC2_4 = tokenC1_3)}
usrC= 6 // {usrC_6 : (usrC_6 = usrA1_6) & (usrC_6 = 1) & (pwdC_6 = usrC_6+1) & (appIdC_6 =
    usrC_6+2), usrC_1 : (usrC_1 = usrA1_1) & (usrC_1 = 1) & (pwdC_1 = usrC_1+1) & (appIdC_1 =
    usrC_1+2)}
tokenA1= 7 // {tokenA1_7 : (tokenA1_7 = tokenV1_7) & (tokenA1_7 = 1234), tokenA1_2 : (tokenA1_2
     = tokenV1_2) & (tokenA1_2 = 1234)}
tokenA2= 8 // {tokenA2_8 : (tokenA2_8 = tokenC1_8) & (tokenA2_8 = 123), tokenA2_3 : (tokenA2_3
    = tokenC1_3) & (tokenA2_3 = 123)}
```

(11)

usrA1= 6 // {$usrA1_6$: $usrC_6$ = $usrA1_6$), $usrA1_1$: $usrC_1$ = $usrA1_1$)}
usrA2= 0 // {}
appIdC= 6 // {$appIdC_6$: ($appIdC_6$ = $appIdA1_6$) & ($usrC_6$ = 1) & ($pwdC_6$ = $usrC_6$+1) & ($appIdC_6$ = $usrC_6$+2), $appIdC_1$: ($appIdC_1$ = $appIdA1_1$) & ($usrC_1$ = 1) & ($pwdC_1$ = $usrC_1$+1) & ($appIdC_1$ = $usrC_1$+2)}
tokenW1= 9 // {$tokenW1_9$: ($tokenC2_9$ = $tokenW1_9$), $tokenW1_4$: ($tokenC2_4$ = $tokenW1_4$) & ($tokenW1_4$ = $tokenW2_5$)}
tokenV1= 7 // {$tokenV1_7$: ($tokenA1_7$ = $tokenV1_7$), $tokenV1_2$: ($tokenA1_2$ = $tokenV1_2$) & ($tokenV1_2$ $\neq$ $tokenV2_5$)}
appIdA1= 6 // {$appIdA1_6$: ($appIdC_6$ = $appIdA1_6$), $appIdA1_1$: ($appIdC_1$ = $appIdA1_1$)}
tokenW2= 5 // {$tokenW2_5$: ($tokenW2_5$ = $tokenV2_5$) & ($tokenW1_4$ = $tokenW2_5$)}
appIdA2= 0 // {}
tokenV2= 5 // {$tokenV2_5$: ($tokenW2_5$ = $tokenV2_5$) & ($tokenV1_2$ $\neq$ $tokenV2_5$)}
pwdA1= 6 // {$pwdA1_6$: ($pwdC_6$ = $pwdA1_6$), $pwdA1_1$: ($pwdC_1$ = $pwdA1_1$)}
pwdA2= 0 // {}
pwdC= 6 // {$pwdC_6$: ($pwdC_6$ = $pwdA1_6$) & ($usrC_6$ = 1) &($pwdC_6$ = 2) & ($appIdC_6$ = 3), $pwdC_1$: ($pwdC_1$ = $pwdA1_1$) & ($usrC_1$ = 1) & ($pwdC_1$ = $usrC_1$+1) & ($appIdC_1$ = $usrC_1$+2)}
tokenC1= 8 // {$tokenC1_8$: ($tokenA2_8$ = $tokenC1_8$), $tokenC1_3$: ($tokenA2_3$ = $tokenC1_3$)}
tokenC2= 9 // {$tokenC2_9$: ($tokenC2_9$ = $tokenW1_9$) & ($tokenC2_9$ = $tokenC1_8$), $tokenC2_4$: ($tokenC2_4$ = $tokenW1_4$) & ($tokenC2_4$ = $tokenC1_3$)}
usrC= 6 // {$usrC_6$: ($usrC_6$ = $usrA1_6$) & ($usrC_6$ = 1) & ($pwdC_6$ = $usrC_6$+1) & ($appIdC_6$ = $usrC_6$+2), $usrC_1$: ($usrC_1$ = $usrA1_1$) & ($usrC_1$ = 1) & ($pwdC_1$ = $usrC_1$+1) & ($appIdC_1$ = $usrC_1$+2)}
tokenA1= 7 // {$tokenA1_7$: ($tokenA1_7$ = $tokenV1_7$) & ($tokenA1_7$ = 1234), $tokenA1_2$: ($tokenA1_2$ = $tokenV1_2$) & ($tokenA1_2$ = 1234)}
tokenA2= 8 // {$tokenA2_8$: ($tokenA2_8$ = $tokenC1_8$) & ($tokenA2_8$ = 123), $tokenA2_3$: ($tokenA2_3$ = $tokenC1_3$) & ($tokenA2_3$ = 123)}

B. RESULTADOS DE CASO CIRCULAR

B.1. Segunda versión del algoritmo

	UNICO SIN DATOS			
	Method	**CPU**	**%**	**Meth Invoc**
100 Ciclos	isContradiction	6375	45,79	301
201 STS States	isSatisfactible	6401	45,98	301
Tiempo	parse	736	5,29	1
13922	CM.hasChannel()	0	0,00	
	AbtractList.Equals	0	0,00	
500 Ciclos	isContradiction	18558	46,52	1501
1001 STS States	isSatisfactible	18615	46,66	1501
Tiempo	parse	1665	4,17	1
39891	CM.hasChannel()	0	0,00	
	AbtractList.Equals	0	0,00	
1000 Ciclos	isContradiction	40451	47,69	3001
2001 STS States	isSatisfactible	40565	47,83	3001
Tiempo	parse	1956	2,31	1
84814	CM.hasChannel()	0	0,00	
	AbtractList.Equals	0	0,00	
2500 Ciclos	isContradiction	115415	45,25	7501
5001 STS States	isSatisfactible	129064	50,60	7501
Tiempo	parse	6020	2,36	1
255088	CM.hasChannel()	0	0,00	
	AbtractList.Equals	0	0,00	
5000 Ciclos	isContradiction	248447	46,08	15001
10001 STS States	isSatisfactible	271146	50,29	15001
Tiempo	parse	5155	0,96	1
539207	CM.hasChannel()	0	0,00	
	AbtractList.Equals	0	0,00	

Fig. B.1: Resultados caso circular una máquina B sin datos

	DISTINTOS SIN DATOS			
	Method	CPU	%	Meth Invoc
100 Ciclos	CM.hasCHannel	18	0,22	602
201 STS States	isContradiction	3622	44,55	301
Tiempo	isSatisfactible	3241	39,86	301
8131	parse	785	9,65	1
	AbtractList.Equals	0	0,00	0
500 Ciclos	CM.hasCHannel	28	0,07	3002
1001 STS States	isContradiction	19094	45,04	1501
Tiempo	isSatisfactible	19220	45,34	1501
42389	parse	2829	6,67	1
	AbtractList.Equals	0	0,00	0

Fig. B.2: Resultados caso circular n máquinas B sin datos

	UNICO CON DATOS			
	Method	CPU	%	Meth Invoc
1 Ciclos	isContradiction	674	55,29	7
5 STS States	isSatisfactible	134	10,99	7
Tiempo	knowledgeEquiv	69	5,66	12
1219	sts.nextState	6	0,49	7
	CM.hasChannel()	1	0,08	14
	parse	165	13,54	1
	boundedVars	1	0,08	12
2 Ciclos	isContradiction	363	39,76	29
19 STS States	isSatisfactible	364	39,87	29
Tiempo	knowledgeEquiv	29	3,18	82
913	sts.nextState	20	2,19	29
	CM.hasChannel()	3	0,33	58
	boundedVars	8	0,88	104
3 Ciclos	isContradiction	1307	39,07	103
64 STS States	isSatisfactible	1463	43,74	103
Tiempo	knowledgeEquiv	135	4,04	419
3345	sts.nextState	78	2,33	103
	CM.hasChannel()	7	0,21	206
	boundedVars	65	1,94	890
4 Ciclos	isContradiction	3449	43,25	329
197 STS States	isSatisfactible	3476	43,59	329
Tiempo	knowledgeEquiv	226	2,83	1750
7974	sts.nextState	201	2,52	7182
	CM.hasChannel()	12	0,15	658
	boundedVars	143	1,79	329

Fig. B.3: Resultados caso circular una máquina B con datos (primera parte)

5 Ciclos	**isContradiction**	10858	43,67	971
566 STS States	**isSatisfactible**	11445	46,03	971
Tiempo	**knowledgeEquiv**	563	2,26	6308
24866	**sts.nextState**	282	1,13	971
	CM.hasChannel()	31	0,12	1942
	boundedVars	584	2,35	49998
6 Ciclos	**isContradiction**	34030	45,36	2701
1543 STS States	**isSatisfactible**	34220	45,61	2701
Tiempo	**knowledgeEquiv**	1464	1,95	20470
75024	**sts.nextState**	637	0,85	2701
	CM.hasChannel()	55	0,07	5402
	boundedVars	1949	2,60	320608
7 Ciclos	**isContradiction**	81848	43,31	7183
4040 STS States	**isSatisfactible**	83007	43,92	7183
Tiempo	**knowledgeEquiv**	4367	2,31	65660
188994	**sts.nextState**	1660	0,88	7183
	CM.hasChannel()	120	0,06	14366
	boundedVars	10963	5,80	1946246

Fig. B.4: Resultados caso circular una máquina B con datos (segunda parte)

	DISTINTOS CON DATOS			
	Method	CPU	%	Meth Invoc
1 Ciclos	isContradiction	378	46,90	7
5 STS States	isSatisfactible	85	10,55	7
Tiempo	knowledgeEquiv	43	5,33	12
806	sts.nextState	6	0,74	7
	bind	16	1,99	7
	CM.hasChannel()	1	0,12	14
	parse	161	19,98	1
	buildHyp	9	1,12	12
2 Ciclos	isContradiction	414	40,59	24
16 STS States	isSatisfactible	339	33,24	24
Tiempo	knowledgeEquiv	74	7,25	72
1020	sts.nextState	20	1,96	24
	buildHyp	35	3,43	96
	CM.hasChannel()	2	0,20	48
	parse	46	4,51	1
	boundedVars	8	0,78	96
3 Ciclos	isContradiction	770	38,27	68
44 STS States	isSatisfactible	794	39,46	68
Tiempo	knowledgeEquiv	119	5,91	288
2012	sts.nextState	55	2,73	68
	buildHyp	68	3,38	288
	CM.hasChannel()	6	0,30	136
	parse	81	4,03	1
	boundedVars	41	2,04	426
4 Ciclos	isContradiction	2022	41,13	176
112 STS States	isSatisfactible	1955	39,77	176
Tiempo	knowledgeEquiv	270	5,49	960
4916	sts.nextState	133	2,71	176
	buildHyp	140	2,85	768
	CM.hasChannel()	11	0,22	352
	boundedVars	152	3,09	2016

Fig. B.5: Resultados caso circular n máquinas B con datos (primera parte)

5 Ciclos	**isContradiction**	4425	42,28	432
272 STS States	**isSatisfactible**	4432	42,35	432
Tiempo	**knowledgeEquiv**	413	3,95	2880
10465	**sts.nextState**	259	2,47	432
	buildHyp	290	2,77	1920
	CM.hasChannel()	17	0,16	864
	boundedVars	267	2,55	10230
6 Ciclos	**isContradiction**	10896	43,76	1024
640 STS States	**isSatisfactible**	10939	43,93	1024
Tiempo	**knowledgeEquiv**	817	3,28	8064
24900	**sts.nextState**	425	1,71	1024
	buildHyp	437	1,76	4608
	CM.hasChannel()	29	0,12	2048
	boundedVars	661	2,65	48184
7 Ciclos	**isContradiction**	27568	44,19	2368
1472 STS States	**isSatisfactible**	27637	44,30	2368
Tiempo	**knowledgeEquiv**	1853	2,97	21504
62381	**sts.nextState**	829	1,33	2368
	buildHyp	756	1,21	10752
	CM.hasChannel()	50	0,08	4736
	boundedVars	2203	3,53	222190

Fig. B.6: Resultados caso circular n máquinas B con datos (segunda parte)

8 Ciclos	**isContradiction**	54498	41,89	5376
3328 STS States	**isSatisfactible**	55202	42,43	5376
Tiempo	**knowledgeEquiv**	4356	3,35	55296
130101	**sts.nextState**	1663	1,28	5376
	buildHyp	1206	0,93	24576
	CM.hasChannel()	92	0,07	10752
	boundedVars	10117	7,78	996722
9 Ciclos	**isContradiction**	121186	38,76	12032
7424 STS States	**isSatisfactible**	122271	39,11	12032
Tiempo	**knowledgeEquiv**	11506	3,68	138240
312635	**sts.nextState**	3322	1,06	12032
	buildHyp	2360	0,75	55296
	CM.hasChannel()	149	0,05	24064
	boundedVars	44182	14,13	4439102

Fig. B.7: Resultados caso circular n máquinas B con datos (tercera parte)

B.2. Tercera versión del algoritmo

	UNICO SIN DATOS			
	Method	CPU	%	Meth Invoc
100 Ciclos	isContradiction	5209	45,77	301
201 STS States	isSatisfactible	5223	45,90	301
Tiempo	parse	663	5,83	1
11380	CM.hasChannel()	31	0,27	602
500 Ciclos	isContradiction	30860	47,30	1501
1001 STS States	isSatisfactible	30750	47,14	1501
Tiempo	parse	2786	4,27	1
65237	CM.hasChannel()	41	0,06	3002
1000 Ciclos	isContradiction	37322	47,97	3001
2001 STS States	isSatisfactible	37223	47,84	3001
Tiempo	parse	1699	2,18	1
77809	CM.hasChannel()	74	0,10	6002
2500 Ciclos	isContradiction	146102	48,70	7501
5001 STS States	isSatisfactible	144825	48,27	7501
Tiempo	parse	4804	1,60	1
300024	CM.hasChannel()	257	0,09	15002
5000 Ciclos	isContradiction	313957	47,71	15001
10001 STS States	isSatisfactible	312602	47,51	15001
Tiempo	parse	20984	3,19	1
658003	CM.hasChannel()	808	0,12	30002

Fig. B.8: Resultados caso circular una máquina B sin datos

	DISTINTOS SIN DATOS			
	Method	CPU	%	Meth Invoc
100 Ciclos	**CM.hasCHannel**	19	0,25	602
201 STS States	**isContradiction**	3262	43,65	301
Tiempo	**isSatisfactible**	2898	38,78	301
7473	**parse**	819	10,96	1
500 Ciclos	**CM.hasCHannel**	20	0,05	3002
1001 STS States	**isContradiction**	18478	45,52	1501
Tiempo	**isSatisfactible**	18095	44,57	1501
40596	**parse**	2901	7,15	1

Fig. B.9: Resultados caso circular n máquinas B sin datos

	UNICO CON DATOS			
	Method	CPU	%	Meth Invoc
1 Ciclos	isContradiction	385	46,67	7
5 STS States	isSatisfactible	90	10,91	7
Tiempo	parse	181	21,94	1
825	sts.nextState	4	0,48	7
	HashMap Iterator.Next ()		0,00	
	CM.hasChannel()	1	0,12	14
	knowledgeEquiv	42	5,09	12
	boundedVars	0,9	0,11	12
2 Ciclos	isContradiction	394	38,89	29
19 STS States	isSatisfactible	391	38,60	29
Tiempo	knowledgeEquiv	38	3,75	82
1013	sts.nextState	19	1,88	29
	parse	40	3,95	1
	CM.hasChannel()	2	0,20	58
	buildHyp	35	3,46	120
	boundedVars	8	0,79	104
3 Ciclos	isContradiction	1281	42,45	103
64 STS States	isSatisfactible	1238	41,02	103
Tiempo	knowledgeEquiv	110	3,64	419
3018	sts.nextState	65	2,15	103
	parse	50	1,66	1
	CM.hasChannel()	5	0,17	206
	buildHyp	109	3,61	234
	boundedVars	52	1,72	890
4 Ciclos	isContradiction	3496	43,07	329
197 STS States	isSatisfactible	3450	42,50	329
Tiempo	knowledgeEquiv	235	2,90	1750
8117	sts.nextState	157	1,93	329
	parse	76	0,94	1
	CM.hasChannel()	13	0,16	658
	buildHyp	238	2,93	792
	boundedVars	233	2,87	7182

Fig. B.10: Resultados caso circular una máquina B con datos (primera parte)

5 Ciclos	**isContradiction**	10396	44,58	971
566 STS States	**isSatisfactible**	10480	44,94	971
Tiempo	**knowledgeEquiv**	519	2,23	6308
23322	**sts.nextState**	303	1,30	971
	parse	89	0,38	1
	CM.hasChannel()	27	0,12	1942
	buildHyp	446	1,91	2430
	boundedVars	485	2,08	49998
6 Ciclos	**isContradiction**	27331	45,16	2701
1543 STS States	**isSatisfactible**	27433	45,33	2701
Tiempo	**knowledgeEquiv**	1223	2,02	20470
60525	**sts.nextState**	568	0,94	2701
	parse	107	0,18	1
	CM.hasChannel()	49	0,08	5402
	buildHyp	829	1,37	6948
	boundedVars	1640	2,71	320608
7 Ciclos	**isContradiction**	69531	43,17	7183
4040 STS States	**isSatisfactible**	69784	43,33	7183
Tiempo	**knowledgeEquiv**	4202	2,61	65660
161064	**sts.nextState**	1468	0,91	7183
	parser	97	0,06	1
	CM.hasChannel()	111	0,07	14366
	buildHyp	2049	1,27	18858
	boundedVars	9545	5,93	1946246

Fig. B.11: Resultados caso circular una máquina B con datos (segunda parte)

	DISTINTOS CON DATOS			
	Method	CPU	%	Meth Invoc
1 Ciclos	isContradiction	380	28,15	7
5 STS States	isSatisfactible	93	6,89	7
Tiempo	parse	714	52,89	1
1350	sts.nextState	3	0,22	7
	bind	9	0,67	7
	buildHyp	8	0,59	12
	knowledgeEquiv	41	3,04	12
	boundedVars	1	0,07	12
2 Ciclos	isContradiction	332	37,64	24
16 STS States	isSatisfactible	357	40,48	24
Tiempo	knowledgeEquiv	39	4,42	72
882	sts.nextState	19	2,15	24
	buildHyp	30	3,40	48
	CM.hasChannel()	2	0,23	48
	parse	51	5,78	1
	boundedVars	6	0,68	64
3 Ciclos	isContradiction	821	38,84	68
44 STS States	isSatisfactible	845	39,97	68
Tiempo	knowledgeEquiv	126	5,96	288
2114	sts.nextState	57	2,70	68
	buildHyp	76	3,60	144
	CM.hasChannel()	5	0,24	136
	parse	68	3,22	1
	boundedVars	43	2,03	426
4 Ciclos	isContradiction	2097	40,52	176
112 STS States	isSatisfactible	2095	40,48	176
Tiempo	knowledgeEquiv	246	4,75	960
5175	sts.nextState	144	2,78	176
	buildHyp	167	3,23	384
	CM.hasChannel()	17	0,33	352
	parser	134	2,59	1
	boundedVars	130	2,51	2016

Fig. B.12: Resultados caso circular n máquinas B con datos (primera parte)

5 Ciclos	**isContradiction**	4682	42,13	432
272 STS States	**isSatisfactible**	4637	41,73	432
Tiempo	**knowledgeEquiv**	456	4,10	2880
11112	**sts.nextState**	274	2,47	432
	buildHyp	255	2,29	960
	CM.hasChannel()	18	0,16	864
	parser	116	1,04	1
	boundedVars	362	3,26	10230
6 Ciclos	**isContradiction**	10931	43,92	1024
640 STS States	**isSatisfactible**	10990	44,15	1024
Tiempo	**knowledgeEquiv**	782	3,14	8064
24890	**sts.nextState**	420	1,69	1024
	buildHyp	428	1,72	2304
	CM.hasChannel()	29	0,12	2048
	parser	121	0,49	1
	boundedVars	579	2,33	48184
7 Ciclos	**isContradiction**	24579	43,92	2368
1472 STS States	**isSatisfactible**	24608	43,97	2368
Tiempo	**knowledgeEquiv**	1809	3,23	21504
55966	**sts.nextState**	755	1,35	2368
	buildHyp	707	1,26	5376
	CM.hasChannel()	48	0,09	4736
	parser	137	0,24	1
	boundedVars	2000	3,57	222190

Fig. B.13: Resultados caso circular n máquinas B con datos (segunda parte)

8 Ciclos	**isContradiction**	52935	42,34	5376
3328 STS States	**isSatisfactible**	52881	42,30	5376
Tiempo	**knowledgeEquiv**	4265	3,41	55296
125013	**sts.nextState**	1544	1,24	5376
	buildHyp	1635	1,31	12288
	CM.hasChannel()	85	0,07	10752
	parser	167	0,13	1
	boundedVars	8800	7,04	996722
9 Ciclos	**isContradiction**	127701	39,57	12032
7424 STS States	**isSatisfactible**	126222	39,12	12032
Tiempo	**knowledgeEquiv**	11673	3,62	138240
322691	**sts.nextState**	3277	1,02	12032
	buildHyp	2499	0,77	27648
	CM.hasChannel()	157	0,05	24064
	parser	307	0,10	1
	boundedVars	43386	13,45	4439010

Fig. B.14: Resultados caso circular n máquinas B con datos (tercera parte)

B.3. Cuarta versión del algoritmo con contextEquivalent

	UNICO SIN DATOS			
	Method	**CPU**	**%**	**Meth Invoc**
100 Ciclos	isContradiction	5519	41,21	301
201 STS States	isSatisfactible	5657	42,24	301
Tiempo	ctxAreEquiv	1829	13,66	101
13393				
500 Ciclos	isContradiction	18071	41,94	1501
1001 STS States	isSatisfactible	17753	41,20	1501
Tiempo	ctxAreEquiv	5942	13,79	501
43091				
1000 Ciclos	isContradiction	35653	42,24	3001
2001 STS States	isSatisfactible	34959	41,42	3001
Tiempo	ctxAreEquiv	11647	13,80	1001
84398				
2500 Ciclos	isContradiction	81191	41,97	7501
5001 STS States	isSatisfactible	81348	42,05	7501
Tiempo	ctxAreEquiv	26946	13,93	2501
193433				
5000 Ciclos	isContradiction	160300	41,76	15001
10001 STS States	isSatisfactible	160258	41,75	15001
Tiempo	ctxAreEquiv	53332	13,89	5001
383836				

Fig. B.15: Resultados caso circular una máquina B sin datos

	DISTINTOS SIN DATOS			
	Method	**CPU**	**%**	**Meth Invoc**
100 Ciclos	isContradiction	2894	37,54	299
201 STS States	isSatisfactible	2676	34,71	299
Tiempo	parse	363	4,71	1
7709	ctxAreEquiv	885	11,48	100
500 Ciclos	isContradiction	15445	38,21	1501
1001 STS States	isSatisfactible	15324	37,91	1501
Tiempo	parse	1110	2,75	1
40424	ctxAreEquiv	5173	12,80	501

Fig. B.16: Resultados caso circular n máquinas B sin datos

	UNICO CON DATOS			
	Method	CPU	%	Meth Invoc
1 Ciclos	isContradiction	261	22,29	7
5 STS States	isSatisfactible	70	5,98	7
Tiempo	parse	93	7,94	1
1171	sts.nextState	1	0,09	7
	ctxAreEquiv	94	8,03	3
	buildHyp	46	3,93	252
2 Ciclos	isContradiction	319	26,32	29
19 STS States	isSatisfactible	266	21,95	29
Tiempo	sts.nextState	4	0,33	29
1212	ctxAreEquiv	470	38,78	15
	buildHyp	113	9,32	60
	boundedVars	3	0,25	100
3 Ciclos	isContradiction	1040	21,99	103
64 STS States	isSatisfactible	1034	21,87	103
Tiempo	sts.nextState	15	0,32	103
4729	ctxAreEquiv	2263	47,85	68
	buildHyp	299	6,32	234
	boundedVars	21	0,44	904
4 Ciclos	isContradiction	2966	18,40	329
197 STS States	isSatisfactible	2925	18,15	329
Tiempo	sts.nextState	41	0,25	329
16118	ctxAreEquiv	8857	54,95	22
	buildHyp	739	4,58	792
	boundedVars	217	1,35	7170
5 Ciclos	isContradiction	8156	13,16	971
566 STS States	isSatisfactible	8078	13,03	971
Tiempo	sts.nextState	167	0,27	971
61982	ctxAreEquiv	40247	64,93	949
	buildHyp	3284	5,30	2430
	boundedVars	1828	2,95	50158
6 Ciclos	isContradiction	23003	14,99	2698
1543 STS States	isSatisfactible	22591	14,72	2698
Tiempo	sts.nextState	542	0,35	2698
153427	ctxAreEquiv	95552	62,28	3026
	buildHyp	8242	5,37	6942
	boundedVars	12975	8,46	320966

Fig. B.17: Resultados caso circular una máquina B con datos

	DISTINTOS CON DATOS			
	Method	**CPU**	**%**	**Meth Invoc**
1 Ciclos	isContradiction	245	19,82	7
5 STS States	isSatisfactible	67	5,42	7
Tiempo	ctxAreEquiv	96	7,77	3
1236	buildHyp	17	1,38	12
2 Ciclos	isContradiction	231	23,24	24
16 STS States	isSatisfactible	226	22,74	24
Tiempo	ctxAreEquiv	409	41,15	9
994	sts.nextState	4	0,40	24
	buildHyp	80	8,05	48
	boundedVars	2	0,20	78
3 Ciclos	isContradiction	741	20,85	68
44 STS States	isSatisfactible	757	21,30	68
Tiempo	ctxAreEquiv	1761	49,55	25
3554	sts.nextState	15	0,42	68
	buildHyp	202	5,68	144
	boundedVars	15	0,42	392
4 Ciclos	isContradiction	1670	17,02	176
112 STS States	isSatisfactible	1645	16,77	176
Tiempo	ctxAreEquiv	5635	57,43	65
9812	sts.nextState	32	0,33	176
	buildHyp	649	6,61	384
	boundedVars	98	1,00	2104
5 Ciclos	isContradiction	3786	14,12	432
272 STS States	isSatisfactible	3246	12,11	432
Tiempo	ctxAreEquiv	16545	61,70	161
26815	sts.nextState	73	0,27	432
	buildHyp	1471	5,49	960
	boundedVars	567	2,11	10400

Fig. B.18: Resultados caso circular n máquinas B con datos (primera parte)

6 Ciclos	isContradiction	8549	12,41	1024
640 STS States	isSatisfactible	8459	12,28	1024
Tiempo	ctxAreEquiv	45023	65,37	385
68870	sts.nextState	231	0,34	1024
	buildHyp	3224	4,68	2304
	boundedVars	3134	4,55	48476
7 Ciclos	isContradiction	19867	8,52	2368
1472 STS States	isSatisfactible	19554	8,39	2368
Tiempo	ctxAreEquiv	166172	71,26	897
233180	sts.nextState	650	0,28	2368
	buildHyp	9818	4,21	5376
	boundedVars	16864	7,23	222038
8 Ciclos	isContradiction	44440	9,18	5376
3328 STS States	isSatisfactible	43780	9,04	5376
Tiempo	ctxAreEquiv	293898	60,70	2049
484164	sts.nextState	1525	0,31	5376
	buildHyp	14564	3,01	12288
	boundedVars	85139	17,58	994480
9 Ciclos	isContradiction	99857	7,25	12032
7424 STS States	isSatisfactible	98805	7,17	12032
Tiempo	ctxAreEquiv	714282	51,85	4609
1377621	sts.nextState	3779	0,27	12032
	buildHyp	30449	2,21	27648
	boundedVars	429627	31,19	4609

Fig. B.19: Resultados caso circular n máquinas B con datos (segunda parte)

Bibliografía

[1] Z3: An efficient smt solver. On-line. Available at `http://research.microsoft.com/projects/z3/`.

[2] G. Alonso, F. Casati, H. A. Kuno, and V. Machiraju. *Web Services - Concepts, Architectures and Applications*. Data-Centric Systems and Applications. Springer, 2004.

[3] D. Brand and P. Zafiropulo. On communicating finite-state machines. *Journal of the ACM*, 30(2):323–342, 1983.

[4] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5(7):394–397, 1962.

[5] G. Delzanno and T. Bultan. Constraint-based verification of client-server protocols. In T. Walsh, editor, *Principles and Practice of Constraint Programming - CP 2001, 7th International Conference, CP 2001, Paphos, Cyprus, November 26 - December 1, 2001, Proceedings*, volume 2239 of *Lecture Notes in Computer Science*, pages 286–301. Springer, 2001.

[6] P. Deniélou and N. Yoshida. Multiparty session types meet communicating automata. In *ESOP*, pages 194–213, 2012.

[7] X. Fu, T. Bultan, and J. Su. Conversation protocols: a formalism for specification and verification of reactive electronic services. *Theoretical Computer Science*, 328(1-2):19–37, 2004.

[8] X. Fu, T. Bultan, and J. Su. Realizability of conversation protocols with message contents. *International Journal of Web Service Research*, 2(4):68–93, 2005.

[9] E. Gamma, R. Helm, R. Johnson, J. Vlissides, and G. Booch. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Addison-Wesley, 1980.

[10] N. Kavantzas, D. Burdett, G. Ritzinger, T. Fletcher, Y. Lafon, and C. Barreto. Web services choreography description language version 1.0. World Wide Web Consortium, Candidate Recommendation CR-ws-cdl-10-20051109, November 2005.

[11] J. Lange, E. Tuosto, and N. Yoshida. From communicating machines to graphical choreographies. In S. K. Rajamani and D. Walker, editors, *Proceedings of 42rd. Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, pages 221–232, New York, NY, USA, 2015. ACM.

[12] C. Peltz. Web services orchestration and choreography. *Computer*, 36(10):46–52, 2003.

[13] I. Vissani, C. G. Lopez Pombo, H. Melgratti, and E. Tuosto. Data-aware communicating finite state machines. 2016.