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

Bisimulación de *Data-aware Communicating Finite State Machines* con propiedades en las acciones

Tesis de Licenciatura en Ciencias de la Computación

Diego Norberto Senarruzza Anabia

Director: Hernan Claudio Melgratti
Director: Carlos Gustavo Lopez Pombo
Buenos Aires, 2023

BISIMULACIÓN DE *DATA-AWARE COMMUNICATING FINITE STATE MACHINES* CON PROPIEDADES EN LAS ACCIONES

Los nuevos paradigmas de computación como *service-oriented computing* (SOC) o *Cloud/Fog*, como así también el Internet de las cosas (IoT por su sigla en inglés), han impulsado enormemente lo que hoy se denomina *economía de APIs*. La idea que subyace en la economía de APIs es la posibilidad de construir nuevos servicios utilizando APIs provistas por terceras partes y, a su vez, hacer disponibles estos nuevos servicios, publicando sus propias APIs. La creciente interconexión entre aplicaciones y dispositivos hacen al surgimiento de nuevos y esenciales requerimientos por parte de las aplicaciones actuales, tales como *self-adaptiveness* y reconfiguración dinámica transparente.

En la industria actual, las APIs representan el último escalón de interoperabilidad, y ponen en relieve la necesidad de descripciones precisas como forma preponderante de documentación. Sin embargo, en la mayoría de los casos, los aspectos más importantes del comportamiento de las API son documentados informalmente, dificultando la validación del software que se obtiene como resultado de componer servicios a través de sus APIs, así como el establecimiento de propiedades y el mantenimiento de aplicaciones. En consecuencia, describir formalmente el comportamiento de las APIs de forma que este provea garantías a sus clientes constituye un desafío técnico clave en este contexto.

El presente proyecto de tesis asume una perspectiva en la que el ideal detrás de la ejecución de aplicaciones basadas en APIs se lleva a cabo sobre una infraestructura de comunicación y cómputo ubicua y preexistente y en la que un middleware es capaz de solicitar a un *service-broker* la búsqueda de un servicio al que, sujeto a una negociación de nivel servicios (SLA por su sigla en inglés, *Service Level Agreements*), pueda vincularse en forma completamente automática y transparente, para que colectivamente sea posible alcanzar cierto objetivo de negocios. Es necesario contar con descriptores de protocolos precisos a nivel de servicio, así como alguna definición de *compliance* que permita establecer equivalencia entre dichos protocolos. Como candidatos idóneos a estos, surgen las *Asserted Communicating Finite State Machines* (*a-CFSM*), capaces de describir el intercambio de mensajes entre participante (o servicios) y de establecer condiciones sobre las variables intercambiadas en dichos mensajes (pre-condiciones sobre mensajes enviados y post-condiciones sobre mensajes recibios).

Ligado a la necesidad de una definición de *compliance*, surge como problema el hecho de que las partes involucradas en la comunicación no necesariamente se conocen entre sí, por lo que descriptores que resulten equivalentes pueden encontrarse escritos utilizado terminología distinta.

En este trabajo nos concentraremos en: 1) abordar una noción de bisimulación a modo de *compliance* para las *a-CFSM*, asumiendo que los autómatas comparten terminología, 2) extender la definición de bisimulación construyendo un *matching* de nombres (un diccionario entre las terminologías de ambos autómatas) y 3) poner en práctica los resultados obtenidos en (1) y (2) a través de la construcción de una herramienta de software. Para facilitar (1), definiremos una abstracción de las *a-CFSM* a las que llamaremos *Assertable Finite State Machines* (*a-FSM*), con la que procederemos a definir tres nociones de bisimulación de manera incremental (cada una a partir de las limitaciones de la anterior), para posteriormente extender hacia las *a-CFSM*.

Palabras claves: Autómatas, a-CFSM, SOC, API, Bisimulación

AGRADECIMIENTOS

A mis directores Carlos y Hernan, por su compromiso y su paciencia a lo largo de este año, por sus sugerencias y correcciones, por sus buenas ideas y su predisposición a que siga las propias.
A mis padres Julia y Ricardo por su incondicional amor y dedicación, por su inquebrantable fe en mi, por estar conmigo en cada paso del camino (y por aquella computadora que hace ya muchos años compraron, donde todo esto inició).
A Victoria, por todo su amor y su apoyo, por estar en los días buenos y malos, por las golosinas para que me quede estudiando; mi compañera en estos años, y en los que están por venir.
A mis hermanos Rosi, Mica, Dario y Alejandro, por las peleas y las risas, por entenderme, por estar siempre que los necesito.
A Valeria y Graciela, mi familia, por sus comidas ricas, por sus consejos y su cariño, por ayudarme siempre y sin dudarlo.
A mis amigos, quienes me acompañan desde el colegio y quienes conocí en estos años de carrera, por entender mis tiempos, por las anécdotas compartidas, por estar siempre dispuestos a sacarme una sonrisa.
A la UBA, a la FCEyN y al DC, por su calidad humana, por su excelente nivel académico, tanto en docentes como alumnos, por su constante búsqueda de mejorar y que mejoremos. Por su educación pública.

Viaje antes que destino.

Índice general

1. INTRODUCCIÓN

A comienzos del siglo 21, ante la necesidad de concebir nuevos mecanismos de integración y comunicación de software debido a la proliferación de la web y los sistemas de comunicación en linea, surge un paradigma conocido como *computación orientada a servicios* (SOC). El objetivo consistía en tener componentes de software que podrían alojarse en una especie de "mercado de componentes", y ser descubiertas o accedidas mediante un *service-broker* en tiempo de ejecución, utilizando alguna noción de contrato y de forma de garantizar condiciones de interoperabilidad.
Los resultados, por parte de la industria, no lograron cumplir con las promesas realizadas a la hora de hacer que: sistemas de software desarrollados por gente distinta, en lugares distintos, interactuasen de forma (mas o menos) transparente. En lugar de eso se terminó promoviendo una lógica intra-empresarial en donde, mediante la creación de un *middleware*, distintas piezas de software construidas y definidas dentro de la propia empresa podían ser accesibles (una estrategia de diseño). A su vez surgieron propuestas por parte de la academia, en donde mediante grandes inversiones y múltiples colaboradores se desarrollaron lenguajes capaces de soportar y de (sobre todo) proveer semántica a esta nueva generación de sistemas.

El problema surge al intentar generalizar esta idea de composición en tiempo de ejecución basándonos en *application programming interfaces* (*APIs*). Al momento de diseñar un sistema, no se tiene ninguna referencia respecto de las componentes contra las que este se va a conectar; por lo tanto, lo que se pierde es la capacidad de asignarle semántica.
Esta mirada mucho mas general fue empujada, principalmente, por la academia, pero tomada de las máximas esperadas de la computación orientada a servicios. Lo que se esperaba de un sistema de esta naturaleza, de esta *utopía* es que, principalmente, fuera:

- Dinámicamente reconfigurable: es decir, se desconoce la arquitectura completa de un sistema, ya que esta se va descubriendo a medida que las componentes necesarias para la ejecución se van adquiriendo o contratando (desde un "mercado de componentes").
- Intrínsecamente heterogéneo: debido a que no se sabe nada de las componentes a contratar.
- Altamente no determinístico: no hay garantía de que, al momento de contratar una componente durante una ejecución, esta vaya a estar disponible al momento de volver a ejecutar. Mas aún, una misma *query* puede devolver resultados distintos dentro de una misma ejecución, al repetirla.

SEArch (*service execution architecture*) es una arquitectura de cómputo distribuido cuyo objetivo es el de promover esta visión. Ante este objetivo se encontró que, principalmente, debían proveerse dos cosas:

1. Reconfiguración dinámica de artefactos de software de manera transparente: dado un sistema, no es la idea que este genere manualmente un vínculo hacia la componente a contratar, sino permitir que **asuma** que tiene una componente con quien puede interactuar, siguiendo un determinado protocolo. Una abstracción de esta componente se encontraría conectada a un cierto puerto, con una determinada descripción del protocolo y en donde, al momento de enviar un dato a través de este puerto, se produciría la reconfiguración dinámica. Es decir, es necesario un tercer agente, un middleware, que monitoree el puerto y se encargue de contactar a un *service-broker* solicitando la componente que cumpla con la descripción del protocolo correspondiente al puerto. De esta manera, el mecanismo resulta completamente transparente para el sistema.

2. Población de repositorios: se necesita alguna noción respecto de como poblar estos repositorios de servicios, a través de ciertos descriptores que le sirvan al broker para poder buscarlos.

Según **SEArch**, la *configuración dinámica transparente* puede ser interpretada de la siguiente manera:

- Se tiene un *cliente* que necesita poder comunicarse con distintas componentes. Cada una de estas componentes esta asociada a un *puerto*.
- Cada uno de estos puertos contiene el contrato que especifica la visión de interoperabilidad que el cliente requiere que la componente satisfaga.
- El cliente se encuentra detrás de un *middleware*, encargado de administrar estos puertos.

Al momento de la ejecución, el cliente va a necesitar comunicarse con la componente que se encuentre detrás del puerto. El middleware, al conocer el puerto y, por ende, el *contrato de requerimiento* (R) que rige el protocolo utilizado a través de dicho puerto, se encarga de contactar al *service-broker* para solicitar un servicio capaz de satisfacer dicho contrato. El broker, al momento de recibir el contrato de requerimiento, realiza una *query* sobre un repositorio de servicios y obtiene candidatos, cada uno compuesto por un identificador del servicio (se puede pensar este identificador como un *URI*, *Uniform Resourse Identifier*) y un *contrato de provisión* (PR). Estos candidatos son filtrados, seleccionado aquel cuyo contrato de provisión satisfaga al contrato de requerimiento. Tras la selección del candidato ganador, este es devuelto al middleware del cliente, que procederá a comunicarse directamente con el middleware correspondiente al servicio elegido.

A su vez, el servicio contratado podría pasar por un proceso equivalente en caso de necesitar comunicarse con alguna componente extra para resolver un (sub)problema. De esta manera la configuración permanece completamente transparente para el cliente, ya que lo que único que hace es intercambiar datos con algún puerto específico (esperando que del otro lado haya una componente ya configurada).

Por otro lado, la *población de repositorios* puede entenderse como la existencia de un servicio con una implementación P, ubicado en algún lugar y descrito por una *URI* que afirma cumplir un contrato PR. La idea es que existe alguna autoridad certificante a la que se le puede solicitar, de alguna manera, garantizar que la implementación P satisface el contrato PR que dice cumplir. Se trata de una noción de coherencia en donde se solicita a alguien certificar que, lo que se afirma poder hacer, efectivamente es así.
Una vez que la autoridad decide que esto es correcto, procede a firmar el servicio mediante el contrato PR y la *URI*, y devuelve este certificado a quien haya solicitado la garantía. Es precisamente mediante estos certificados que alguien podría registrar un servicio en un repositorio para, eventualmente, ser consumido a través de una solicitud a un *service-broker*.

Ahora bien, a la hora de discutir esta propuesta, surgen ciertos problemas teóricos a resolver.

- Modelo de cómputo: es necesario un modelo que proporcione una semántica precisa para esta reconfiguración dinámica de la arquitectura. En [ŢF13] se introdujeron las *ARN* (*Asynchronous Relational Nets*); un dispositivo formal que permite modelar los mecanismos a través de los cuáles se estructuran actividades cuyos requerimientos son satisfechos por servicios descubiertos y vinculados en tiempo de ejecución. Luego, en [IVT16] se presentó una semántica operacional para sistemas modelados con *ARNs* de forma de contemplar los eventos de reconfiguración como acciones realizadas automáticamente por el ambiente cuando estas se hacen necesarias.

- Formalismo de los contratos: es necesario formalizar alguna noción de contrato que permita describir qué es un requerimiento, y qué es una provisión. Para esto, a su vez, necesitamos determinar ciertas características entre ambos servicios.

  1. Interoperabilidad: saber si los servicios van a interactuar correctamente. Debido a que todas las acciones se ejecutan en tiempo real, se requiere un protocolo que permita la comunicación entre el cliente y el servicio.
  2. Atributos funcionales: poder determinar qué es lo que hace un servicio (qué es lo que va a computar). En [MLb] se presentó una caracterización de qué es un atributo de calidad, en tanto conforma una cualidad medible del artefacto de software en cuestión. En [MLa] se usó esta noción de atributo de calidad, y se introdujeron mecanismos que posibilitan la negociación de contratos de calidad de servicios en forma automática.
  3. Atributos no funcionales: poder determinar cómo este cómputo es llevado a cabo, por ejemplo, expresando restricciones sobre el costo del servicio, tiempo de ejecución, recursos utilizados, etc.

1.1. Acerca del trabajo a realizar

En relación a toda esta plataforma de cómputo, en este trabajo nos concentraremos en aportar una solución que contemple los puntos (1) y (2).

Para el primer punto, necesitamos lograr que el broker pueda "mirar" un contrato y encontrar un servicio que haga lo que el cliente necesita, siguiendo el mismo protocolo de comunicación esperado. Ante la necesidad de un contrato de interoperabilidad, surge como candidato idóneo las *Communicating Finite State Machines*[BZ83] (*CFSM*), ya que estas fueron previamente utilizadas como descriptores de protocolos en, por ejemplo, [DY12], [LTY15] y [YZF21].

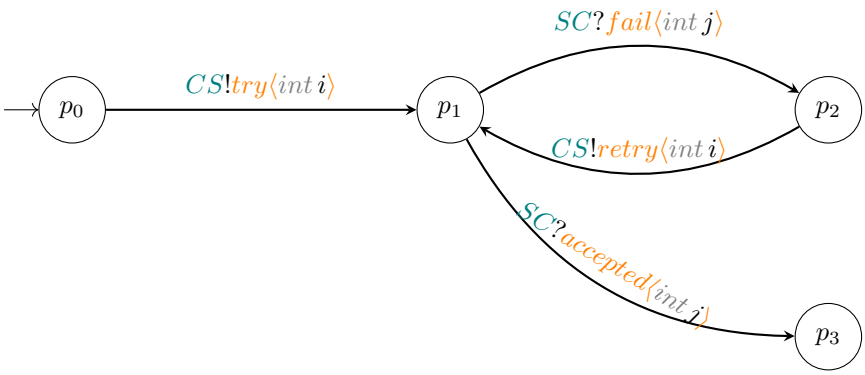


Fig. 1.1: Ejemplo de comunicación entre un cliente C y un servidor S, en una *CFSM*

Para el segundo punto, lo que necesitamos es poder determinar (y especificar) qué se hace con los datos intercambiados, por lo que surge como solución la posibilidad de utilizar (en los arcos) fórmulas que describan algo similar a lo que en un programa es una *pre-condición* o una *post-condición*. Si la CFSM describe el comportamiento del servicio, entonces:

- En una arista correspondiente a un envío de mensaje en donde el cliente es el emisor, la fórmula describe una *assertion* sobre los datos enviados, es decir asegura que cumplen lo descrito en la fórmula (*pre-condición*)

- En una arista correspondiente a un envío de mensaje en donde el cliente es el receptor, la fórmula describe una suposición sobre los datos recibidos (*post-condición).*

Lo que vamos a utilizar entonces, no son *CFSM* clásicas, sino su versión en donde las aristas pueden estar decoradas por assertions (fórmulas en lógica de primer orden), conocidas como *Asserted Communicating Finite State Machines*[GLS+22] (*a-CFSM*).

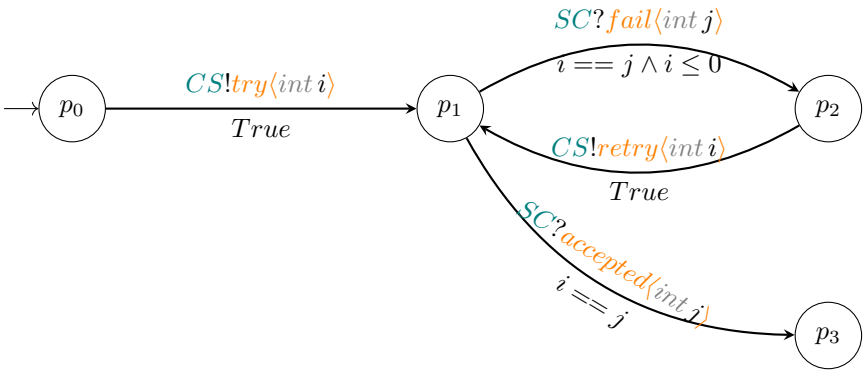


Fig. 1.2: Extensión del ejemplo en la Fig. 1.1 en una *a-CFSM*, en donde ahora las transiciones son restringidas utilizando assertions.

Ahora bien, la idea es trabajar basándonos en la visión explicada en [IVT16] Def. 10, en donde el *cliente* es descrito a través de un protocolo de comunicación *global* que rige el comportamiento del canal que los participantes deben respetar. Siguiendo el mecanismo *top-down* que se describe, se aplica una proyección de los participantes (componentes) sobre el descriptor global, obteniendo los respectivos descriptores *locales* (los contratos de *requerimiento*). El mecanismo además nos asegura que, si un descriptor local es bisimilar a su respectivo contrato de *provisión*, entonces es posible vincularlos.

La idea además es basarnos en [GLS+22] Sec. 4.1, y utilizar los *asserted choreography automatas* (*ac-automatas*) como descriptores globales, junto con la definición de proyección allí explicada (Def. 4.16). Estas proyecciones dan como resultado *a-CFSMs*.

Para que un *service-broker* pueda tomar un contrato de *requerimiento*, un contrato de *provisión* (ambos representados en *a-CFSM*) y pueda determinar si el de *provisión* satisface el de *requerimiento*, necesitamos entonces definir alguna noción de **bisimulación** sobre este tipo de autómatas.

1.2. Problema de matching de nombres

Sumado a todo lo anterior, surge como problema el hecho de que las partes involucradas en la comunicación no necesariamente se conocen entre sí. Esto quiere decir que, quien desarrolla un sistema (un cliente), utiliza nombres (de participantes, de variables y de mensajes) que considera que tienen sentido en su propio contexto. Lo mismo ocurre con quien desarrolla un servicio o una autoridad certificante. A partir de esto surge la necesidad de establecer un criterio de bisimilaridad para *a-CFSM* en donde, no solo se emparejen los nombres entre los autómatas que describen los protocolos, sino que se debe tener una estrategia en donde el matching de nombres se *construya* a medida que se *construye* la relación de bisimulación.

El objetivo de este trabajo es el de abordar la problemática de definir una noción de bisimulación para *a-CFSM*, lo que nos brindará la posibilidad de determinar si un servicio o componente a contratar

cumple con el protocolo de comunicación deseado, inclusive si las máquinas no comparten nombres. Para facilitar esta tarea, primero definiremos una abstracción de las *a-CFSM*, a la que llamaremos *Assertable Finite State Machines* (*a-FSM*). Sobre esta abstracción procederemos a definir tres nociones de bisimulación (siendo cada una extendida a partir de las limitaciones de su predecesora), sin considerar el problema del *matching* de nombres, y posteriormente extenderemos hacia las *a-CFSM* abarcando (ahora si) el problema de *matching*. Finalmente presentaremos una herramienta bajo la idea de poner en práctica los resultados obtenidos, permitiéndonos modelar *a-CFSM*, evaluar bisimilaridad y construir un *matching* de nombres en el proceso. Como complemento a esto realizaremos una serie de pruebas sobre el programa desarrollado, con el fin de capturar y discutir la complejidad del procedimiento.

Esta tesis se divide en cuatro capítulos. En el capítulo 2 describimos algunas nociones básicas de sistemas de transición etiquetados, definimos las *CFSM* y las *a-CFSM* con los que vamos a trabajar a partir de la proyección de *Asserted Choreography Automatas*[GLS+22] (*ac-automata*). En el capítulo 3 definimos las *a-FSM* y establecemos la propiedad de *sensitividad a la historia*, utilizada para definir finalmente las nociones de bisimulación. Posterior a eso extendemos la última de las definiciones hacia las *a-CFSM* considerando el problema *matching* de nombres. En el capítulo 4 damos paso a la explicación de la herramienta desarrollada, repasamos el algoritmo utilizado basado en la estratificación de autómatas finitos, y presentamos las pruebas realizadas. Para finalizar, en el capítulo 5 presentamos algunas conclusiones a partir del trabajo realizado a modo de cierre, junto una mención de ideas para un posible trabajo a futuro.

2. PRELIMINARES

En este capítulo introduciremos las nociones técnicas que serán de utilidad para presentar nuestras contribuciones en los próximos capítulos. Comenzaremos con las nociones de *LTS*, *FSA* y **bisimulación** tal como se presentan en [Mil99].

Definición 1 (Sistema de transición etiquetado (*LTS*))**.** Un sistema de transición etiquetado (*labeled transition system* en inglés, *LTS*) es una tupla $S = \langle \mathcal{Q}, \mathcal{L}, \rightarrow \rangle$ donde:

- $\mathcal{Q}$ es un conjunto de estados,
- $\mathcal{L}$ es un conjunto finito de etiquetas, y
- $\rightarrow \subseteq \mathcal{Q} \times \mathcal{L} \times \mathcal{Q}$ es un conjunto de transiciones etiquetadas.

Usamos la notación usual $q_1 \xrightarrow{l} q_2$ para una transición $(q_1, l, q_2) \in \rightarrow$.

Definición 2 (Autómata de estados finitos (*FSA*))**.** Definimos un autómata de estados finitos (*finite state machines* en ingles, *FSA*) como una tupla $\langle \mathcal{Q}, q_0, \mathcal{L}, \rightarrow \rangle$ en donde:

- $\mathcal{Q}$ es un conjunto de estados **finitos**,
- $q_0 \in \mathcal{Q}$ es el estado inicial, y
- $\langle \mathcal{Q}, \mathcal{L}, \rightarrow \rangle$ es un *LTS*.

Definición 3 (Camino)**.** Sea $\mathcal{S} = \langle \mathcal{Q}, \mathcal{L}, \rightarrow \rangle$ un sistema de transición etiquetado. Definimos a un **camino** en $\mathcal{S}$ como una secuencia de transiciones

$$\mathcal{P} = p_0 \xrightarrow{l_0} p_1 \rightarrow \cdots \rightarrow p_n \xrightarrow{l_n} p_{n+1}$$

Usaremos $\mathcal{P}$, $\mathcal{P}'$, $\mathcal{P}_i$ y demás derivaciones para denotar un camino.

Definimos a un **camino simple** en $\mathcal{S}$ como un camino en donde todos los estados aparecen una única vez.

Definición 4 (Bisimulación)**.** Sean $S_1 = \langle \mathcal{Q}_1, \mathcal{L}, \rightarrow_1 \rangle$, $S_2 = \langle \mathcal{Q}_2, \mathcal{L}, \rightarrow_2 \rangle$ dos sistemas de transición etiquetados. Una relación binaria $\mathcal{R}$ sobre $\mathcal{Q}_1 \times \mathcal{Q}_2$ es una *simulación* (fuerte) si para todo $(p, q) \in \mathcal{R}$ y $l \in \mathcal{L}$ vale que:

$$p \xrightarrow{l} p' \implies (\exists q') q \xrightarrow{l} q' \wedge (p', q') \in \mathcal{R}$$

Una relación binaria $\mathcal{R}$ sobre $\mathcal{Q}_1 \times \mathcal{Q}_2$ es una *bisimulación* (fuerte) si tanto $\mathcal{R}$ como $\mathcal{R}^{-1}$ son *simulaciones* (fuertes).
Decimos que p es *simulado* por q (escrito $p \lesssim q$) si existe una simulación que contenga al par (p, q).
Decimos que p es *bisimilar* a q (escrito $p \sim q$) si existe una bisimulación que contenga al par (p, q).

Definición 5 (Estratificación, [HM85] Sec. 2.1)**.** Sean $S_1 = \langle \mathcal{Q}_1, \mathcal{L}, \rightarrow_1 \rangle$, $S_2 = \langle \mathcal{Q}_2, \mathcal{L}, \rightarrow_2 \rangle$ dos sistemas de transición etiquetados. Definimos una relación binaria $\sim_n$ sobre $\mathcal{Q}_1 \times \mathcal{Q}_2$, inductivamente de la siguiente manera:

- $p \sim_0 q$ para todo $p \in \mathcal{Q}_1$, $q \in \mathcal{Q}_2$

- $p \sim_n q$ si para todo $l \in \mathcal{L}$:

  i) para todo $p' \in \mathcal{Q}_1$ tal que $p \xrightarrow{l} p'$ existe $q' \in \mathcal{Q}_2$ tal que $q \xrightarrow{l} q'$ y $p' \sim_{n-1} q'$

  ii) para todo $q' \in \mathcal{Q}_2$ tal que $q \xrightarrow{l} q'$ existe $p \in \mathcal{Q}_1$ tal que $p \xrightarrow{l} p'$ y $p' \sim_{n-1} q'$

Decimos que p es *n-bisimilar* a q cuando $p \sim_n q$.

Teorema 1 ([HM85] Teorema 2.1)**.** Sean $S_1 = \langle \mathcal{Q}_1, \mathcal{L}, \rightarrow_1 \rangle$, $S_2 = \langle \mathcal{Q}_2, \mathcal{L}, \rightarrow_2 \rangle$ dos sistemas de transición etiquetados *finitos*. Para cada $p \in \mathcal{Q}_1$, $q \in \mathcal{Q}_2$, $p \sim q$ *si y solo si* $(\forall n \in \mathbb{N}) p \sim_n q$.

Esto es equivalente a decir que $\sim = \bigcap_{n \in \mathbb{N}} \sim_n$.

Notemos que $\sim_0 \supseteq \sim_1 \supseteq \sim_2 \ldots$, y que $\bigcap_{n \in \mathbb{N}} \sim_n$ es el mayor punto fijo en la relación inducida en la Def. 5. Esta caracterización induce un algoritmo sencillo para calcular la bisimulación sobre dos *LTS finitos*, la secuencia de relaciones estratificadas se satura después de un máximo de n pasos para un par de *LTS* de n configuraciones (a partir de un cierto $n \in \mathbb{N}$, no importa cuanto sigamos iterando, siempre obtenemos la misma relación).

Definición 6 (Choreography Automata, [BLT20] Def. 3.1)**.** Sea $\mathcal{M}$ un conjunto finito de mensajes y $\mathcal{B}$ un conjunto finito de participantes. Definimos un *c-automata* como un *FSA* sobre el conjunto de etiquetas $\mathcal{L}_{int}$, descrito como:

$$\mathcal{L}_{int} = \{\, c \to s : m \mid c \neq s \in \mathcal{B} \wedge m \in \mathcal{M} \,\}$$

En un *c-automata*, las etiquetas de la forma $c \to s : m$ representan el envío de un mensaje m, desde un participante c hacia otro s. Definimos $ptp(c \to s : m) := \{\, c, s \,\}$ los participantes descritos en una etiqueta.

Definición 7 (Communicating Finite State Machines, [BLT20] Def. 2.4)**.** Sea $\mathcal{M}$ un conjunto finito de mensajes y $\mathcal{B}$ un conjunto finito de participantes. Definimos una *CFSM* como una *FSA* con etiquetas en $\mathcal{L}_{act}$, descrito como:

$$\mathcal{L}_{act} = \{\, cs!m, cs?m \mid c, s \in \mathcal{B}, m \in \mathcal{M} \,\}$$

En una *CFSM*, las acciones de la forma $cs!m$ representan el envío de un mensaje m, desde un participante c hacia otro participante s. Las acciones de la forma $cs?m$ representan la recepción de un mensaje m, por parte de un participante s, enviado desde otro participante c.

Definición 8 (Eliminación de literales redundantes)**.** Sea $\mathcal{F}$ una fórmula en lógica de primer orden escrita únicamente como una conjunción de literales (entendiendo literal como una fórmula atómica o su negación). Definimos una operación para eliminar literales redundantes de $\mathcal{F}$ de la siguiente manera:

$$\overline{\mathcal{F}} = \begin{cases} L_1 \wedge \cdots \wedge L_n & si \quad \mathcal{F} = L_1 \wedge \cdots \wedge L_n \text{ y } L_i \neq L_j (\forall 1 \leq i, j \leq n) \\ L_1 \wedge \ldots L_i \wedge \ldots L_n & si \quad \mathcal{F} = L_1 \wedge \ldots L_i \wedge \ldots L_j \wedge \ldots L_n \text{ y } L_i = L_j (\forall 1 \leq i, j \leq n) \end{cases}$$

Esta operación nos va a servir al momento de definir las nociones de bisimulación.

2.1. Descriptores de protocolos con restricciones

Siguiendo con la visión explicada en la introducción, la idea es que estamos trabajando en el contexto de una *sesión*. Una sesión consiste en una serie estructurada de intercambios de mensajes entre múltiples participantes [BHTY10]. La estructura de este intercambio se rige siguiendo un protocolo de comunicación, que puede ser descrito utilizando máquinas como los *global graphs* [DY12] o los *choreography automatas* [BZ83]. Nosotros nos vamos a concentrar en estos últimos, mas específicamente, en su versión en donde las aristas (o transiciones) pueden estar decoradas con *assertions* (*asserted choreography automatas*).
Para poder especificar protocolos que abarquen restricciones sobre los datos enviados en un mensaje (el *payload*), extendemos los *c-automata* a *asserted c-automata* (*ac-automata*) y, en consecuencia (como vamos a ver más adelante), las *CFSM* a *asserted CFSM* (*a-CFSM*). Para esto, vamos a reformar la estructura de los mensajes $\mathcal{M}$ descrita en Def. 6 de la siguiente manera:

Sea $\mathcal{M}$ un conjunto de mensajes formado por tuplas etiquetadas de la forma $\mathcal{T}\langle V\rangle$, donde $\mathcal{T}$ es una etiqueta, y $V = t_1v_1, \ldots, t_hv_h$ es una lista de pares conteniendo el tipo y nombre de variable. La idea es que $\mathcal{T}$ representa el nombre del mensaje a intercambiar, y V las variables a intercambiar, cada una acompañada por su *tipo*. El conjunto de variables de V se define como $vars(V) := \{v_1, \ldots, v_h\}$ y, si $\mathcal{B}$ es el conjunto de participantes de $\mathcal{L}_{int}$ (Def. 6) y $c, s \in \mathcal{B}$, entonces

- $vars(m) := vars(V)$
- $vars(c \to s : m) := vars(m)$

son conjuntos de variables de m y $c \to s : m$ respectivamente, donde $m = \mathcal{T}\langle V\rangle$.
Intuitivamente, ahora el mensaje especifica también el tipo de los valores comunicados por el emisor, y las variables "locales" donde el receptor "almacena" esos valores.

Vamos a recuperar de [BHTY10] (con algunos cambios menores) las fórmulas de primer orden para especificar las restricciones sobre los *payload*. El conjunto $\mathcal{A}$ de fórmulas lógicas es derivado de la siguiente gramática:

$$A, B ::= True \mid \phi(e_1, \ldots, e_n) \mid \neg A \mid A \wedge B \mid A \implies B \mid (\exists v : t)A$$

ϕ se extiende sobre predicados atómicos predefinidos con aridad y tipos fijos (ej. *bool*, *int*, *string*, etc.) y $e_1, \ldots, e_n$ denotan expresiones. En lugar de fijar un lenguaje específico, se asume que se abarcan tipos de datos usuales en los lenguajes de programación.
Sea $A \in \mathcal{A}$ un predicado, definimos:

- $vars(A)$ como el conjunto de variables **libres** en A, y
- $bvars(A)$ como el conjunto de variables **ligadas** en A.

A partir de ahora, asumimos que $bvars(A) \cap vars(A) = \emptyset$.

Definición 9 (Asserted Choreography Automata (*ac-automata*), [GLS+22] Def. 4.6)**.** Sea $M = \langle \mathcal{Q}, q_0, \mathcal{L}_{int} \times \mathcal{A}, \to\rangle$ un *FSA* definido sobre el lenguaje $\mathcal{L}_{int} \times \mathcal{A}$, en donde $\mathcal{L}_{int}$ se define sobre el conjunto de mensajes $\mathcal{M}$ introducido en la Sec. 2.1.

Para un estado $q \in \mathcal{Q}$, escribimos $\mathcal{SP}(q)$ para denotar el conjunto de caminos simples que parten de q_0 y llegan a q. Sea $t = (q, (l, A), q') \in \to$ una transición, definimos $vars(t) := vars(l)$ y decimos que:

- t **define** una variable v, si $v \in vars(t)$ y, para todo camino $\mathcal{P} \in \mathcal{SP}(q)$, para toda transición $t' \in \mathcal{P}$ vale $v \notin vars(t')$
- t **redefine** una variable v si, $v \in vars(t)$ y existen un camino $\mathcal{P} \in \mathcal{SP}(q)$ y una transición $t' \in \mathcal{P}$ tal que $v \in vars(t')$

Finalmente, decimos que M es un *ac-automata* si:

1. Para todo par de caminos $\mathcal{P}$, $\mathcal{P}'$ con al menos un estado en común q, si existen $t \in \mathcal{P}$ y $t' \in \mathcal{P}'$ transiciones tal que ambas definen una variable v (en estados anteriores a q), entonces t y t' asignan el mismo tipo a v.
2. El *c-automata* subyacente, obtenido mediante remover las assertions en M, es determinístico.

Usamos la notación $q \xrightarrow[A]{l} q'$ para una transición $(q, (l, A), q') \in \rightarrow$ si $(l, A) \in \mathcal{L}_{int} \times \mathcal{A}$.

La condición (1) evita la confusión respecto del tipo de una variable, cuando esta es definida a lo largo de distintos caminos (no tiene sentido intentar escribir una restricción sobre una variable cuyo tipo no está definido).
Sin perdida de generalidad, vamos a asumir que si $q \xrightarrow[A]{l} q' \in \rightarrow$, entonces $vars(l) \cap bvars(A) = \emptyset$. Esta condición puede ser cumplida simplemente renombrando las variables ligadas en los predicados.

Repasando lo mencionado durante la introducción (Cap. 1), la idea es utilizar los *ac-automatas* como descriptores globales del protocolo de comunicación de nuestro sistema y, a partir de una estrategia *top-down*, obtener el contrato de interoperabilidad del participante (o componente) con el que nos queremos comunicar (en forma de *a-CFSM*).

Definición 10 (Asserted Communicating Finite State Machines (*a-CFSM*))**.** Definimos una *a-CFSM* como una *CFSM* sobre el lenguaje $\mathcal{L}_{act} \times \mathcal{A}$, en donde $\mathcal{L}_{act}$ se define sobre el conjunto de mensajes $\mathcal{M}$ introducido en la Sec. 2.1.

Usamos la notación $q \xrightarrow[A]{l} q'$ para una transición $(q, (l, A), q') \in\!\rightarrow$ si $(l, A) \in \mathcal{L}_{act} \times \mathcal{A}$, y $\rightarrow$ es el conjunto de transiciones.

Definición 11 (Proyección de un *ac-automata*, [GLS+22] Def. 4.16)**.** Sea $M = \langle \mathcal{Q}, q_0, \mathcal{L}_{int} \times \mathcal{A}, \rightarrow \rangle$ un *ac-automata*, y sea $c \in \mathcal{B}$ un participante en M. Definimos a la proyección de una transición $t \in \rightarrow$ sobre c, escrita $t \downarrow_{M,c}$, como:

$$t \downarrow_{M,c} = \begin{cases} q \xrightarrow[A]{cs!m} q & si \quad t = q \xrightarrow[A]{c \rightarrow s:m} q' \\ q \xrightarrow[A]{sc?m} q & si \quad t = q \xrightarrow[A]{s \rightarrow c:m} q' \\ q \xrightarrow[(\exists X):A]{\epsilon} & si \quad t = q \xrightarrow[A]{l} q', p \notin ptp(l) \text{ y } X = \{\, v \in vars(A) \mid t \text{ define } v \,\} \end{cases}$$

La proyección de M sobre c, denotada $M \downarrow_c$, se obtiene determinizando y minimizando a la *a-CFSM* intermedia descrita por:

$$M_c = \left\langle \mathcal{Q}_M, q_{0_M}, \mathcal{L}_{act} \times \mathcal{A}, \{\, (q \xrightarrow[A]{l} q') \downarrow_{AC,c} \mid\ q \xrightarrow[A]{l} q' \in\!\rightarrow \,\} \right\rangle$$

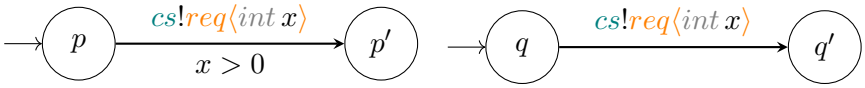
Como se ha mencionado, el objetivo de este trabajo se centra en definir una noción de bisimulación para *a-CFSM*. El hecho de definir un contexto de aplicación para estos autómatas nos permite abordar la problemática desde un punto de vista en donde podemos contar con ciertas garantías respecto de la construcción de estas máquinas. Las *a-CFSM* con las que vamos a trabajar son las que resultan de la proyección sobre algún participante en un *ac-automata*, lo que nos garantiza que las máquinas (por lo menos) van a: 1) ser deterministas y 2) tener variables bien definidas en los caminos hacia un estado (primera condición en la Def. 9).

3. ASSERTED BISIMULATION

En una *CFSM*, una transición $p \xrightarrow{cs!\,m} p'$ representa el intercambio de un mensaje m entre el participante c como emisor y s como receptor. Dado un estado q, la definición de bisimulación en Def. 4 es suficiente para comprobar si q puede simular a p y si p puede simular a q.

En una *a-CFSM*, una transición $p \xrightarrow[A]{cs!\,m} p'$ representa el intercambio de un mensaje m entre el participante c como emisor y s como receptor, en donde las variables en el mensaje m respetan la restricción A. Tenemos ahora que cada transición puede estar decorada por una assertion que, en este caso, restringe las valuaciones posibles para las variables en el mensaje m. Por lo tanto, la Def. 4 resulta insuficiente para comprobar si un estado q puede simular a p, ya que la condición de simulación debiera tener en cuenta esta restricción. De no ser así podríamos terminar afirmando, por ejemplo, que un estado q es bisimilar a otro p, y encontrarnos con trazas disponibles en p pero no en q (lo que no tiene sentido, ya que la equivalencia de trazas es una consecuencia de la bisimulación).

Ejemplo 1 (Restricción de los valores disponibles en un mensaje)**.**

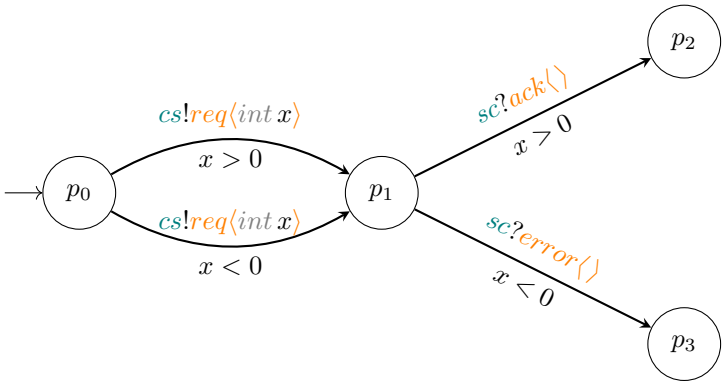


Podemos ver que en la transición desde p, los valores de x válidos para el mensaje req se ven limitados por la assertion $x > 0$. Es claro que q siempre podría simular a p, dado que no se restringen los valores en la transición desde q. Sin embargo, y por este mismo motivo, p no podría simular a q ya que existen trazas como $cs!req\langle\, 0\rangle$ disponible desde q pero no desde p.

Necesitamos entonces una nueva noción de bisimulación para este tipo de autómatas.

Según lo definido para *a-CFSM* en Def. 10, las assertions son escritas como predicados en lógica de primer orden. Sea $\mathcal{P} = p_0 \xrightarrow[A_0]{c_1 s_1!\, m_0} p_1 \to \cdots \to p_n \xrightarrow[A_n]{c_n s_n!\, m_n} p_{n+1}$ un camino en una *a-CFSM*, para llegar hacia el estado p_{n+1} a través de $\mathcal{P}$ fue necesario que se cumplan $A_0, \ldots, A_n$, a las que podemos pensar en términos de cumplimiento de una única restricción, una *fórmula* dada por la conjunción de todas las assertions: $A_0 \wedge \cdots \wedge A_n$.
Dado que $\mathcal{P}$ no necesariamente es el único camino posible para llegar hasta p_{n+1}, y que la posibilidad de avanzar sobre una transición (desde p_{n+1}) depende de las restricciones cumplidas (junto con la restricción impuesta en la propia transición), a la hora de decidir si un estado q puede simular a p_{n+1} vamos a necesitar considerar, además, el contexto en ambos estados. Este contexto va a estar conformado por la *fórmula* correspondiente a las restricciones de un camino posible con el que se llego al estado en cuestión (una fórmula para cada estado, p_{n+1} y q).

Ejemplo 2 (Restricciones posibles en un estado)**.**



Tenemos dos transiciones disponibles para llegar a p_1, una en donde los valores en el mensaje *req* tienen que cumplir la restricción $x > 0$ y otra en donde tienen que cumplir $x < 0$. En el Ej. 1 ya vimos que, para que un estado q_0 pueda simular a p_0, este debe tener en cuenta las restricciones en cada una de las transiciones desde p_0. Sin embargo, no es lo mismo simular p_1 sabiendo que se cumple $x > 0$, ya que nunca se podría mandar el mensaje *error*, y de manera análoga cuando se cumple $x < 0$ para el mensaje *ack*. Por lo tanto, para que un estado q_1 pueda simular a p_1, es necesario saber cuáles fueron las restricciones cumplidas y, por lo tanto, también es necesario saber cuáles fueron cumplidas para llegar a q_1.

La noción de bisimulación, entonces, necesita que una relación $\mathcal{R}$ contenga elementos de la forma: $((p, K_p), (q, K_q))$ donde K_p y K_q son fórmulas correspondientes a las posibles restricciones cumplidas para llegar a p y q respectivamente.

Parte de la problemática a resolver debe considerar el hecho de que los autómatas sobre los que se busca una relación de bisimulación pueden estar escritos utilizando distintos nombres para sus participantes, mensajes o variables. No obstante, en primer lugar vamos a concentrarnos en resolver el caso en donde los autómatas comparten un lenguaje, es decir, que fueron escritos utilizando los mismos nombres para las partes en sus respectivas etiquetas (participantes, mensajes, variables). Posterior a esto, extenderemos lo presentado para abarcar el problema original.

3.1. Autómatas finitos con assertions

Como mencionamos al comienzo del capítulo, el motivo por el que no podemos utilizar una definición de bisimulación "clásica" para las *a-CFSM* se debe a que, al momento de verificar si se cumple una simulación entre dos estados (además de la equivalencia de etiquetas entre las transiciones), debemos considerar el cumplimiento de las restricciones acumuladas a lo largo de los caminos para llegar hacia un estado. Estas restricciones predican en términos de las variables fijadas a lo largo de las transiciones correspondientes a estos caminos, y la equivalencia entre los participantes (emisor y receptor) se verifica al momento de validar la equivalencia entre las etiquetas (de las transiciones, en una simulación entre dos estados). Es decir que si tenemos $t = q \xrightarrow[A]{cs!m\langle t_1\, v_1, \ldots, t_n\, v_n\rangle} q'$ una transición en una *a-CFSM*, la información de las etiquetas que nos interesa tener en cuenta para definir la simulación de assertions (en una nueva noción de bisimulación) se reduce a $m\langle t_1\, v_1, \ldots, t_n\, v_n\rangle$, el mensaje y su contenido (el *payload*).

Dicho esto, podemos abstraernos de las *a-CFSM* hacia un tipo de autómata que modele **solo** un protocolo de envío de mensajes, dejando de lado a los participantes involucrados en la conversación; es decir autómatas en donde las transiciones sean de la forma $q \xrightarrow[A]{m\langle t_1\, v_1, \ldots, t_n\, v_n\rangle} q'$.

Lo siguiente entonces será: 1) definir a este tipo de autómatas, 2) definir propiedades excluyentes (para garantizar el cumplimiento de nuestra definición), 3) definir una noción de bisimulación para el nuevo tipo de autómatas, 4) extenderla hacia las *a-CFSM*.

Definición 12 (Assertable Finite State Machine (*a-FSM*))**.** Sea $\mathcal{M}$ un conjunto de mensajes y $\mathcal{A}$ un conjunto de fórmulas en lógica de primer orden, ambos los definidos para *a-CFSM* (Def. 10)

Una *assertable-FSM* (*a-FSM*) es un sistema de transición etiquetado **determinístico** de la forma $M = \langle Q, q_0, \mathcal{M} \times \mathcal{A}, \rightarrow \rangle$ donde

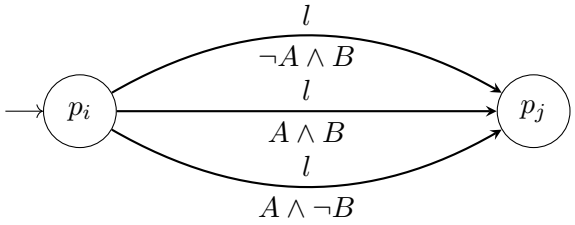
- Q es el conjunto de estados *finito*.
- q_0 es el estado inicial.
- $\mathcal{M} \times \mathcal{A}$ es un conjunto *finito* de etiquetas,
- $\rightarrow$ un conjunto *finito* de transiciones.

Como se describe en la sintaxis presentada en la Sec. 2.1, las assertions obtenidas a partir de $\mathcal{A}$ son limitadas a fórmulas escritas en conjunciones de literales. Esta limitación, sin embargo, no restringe la expresividad de los autómatas, ya que podemos representar operaciones como el *or* ($\vee$) utilizando distintas transiciones. A continuación se muestran ejemplos de esto.

Ejemplo 3 (Reescritura de fórmulas de primer orden utilizando transiciones)**.** Sea M una *a-FSM* y $t = p_i \xrightarrow[F]{k} p_j$ una transición en M. Si $F = A \vee B$, entonces F es verdadera cuando:

- A y B son verdaderas,
- A es falsa y B verdadera, o,
- A es verdadera y B falsa.

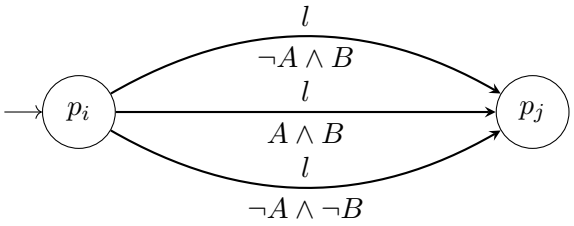
Luego, se puede definir el siguiente autómata.



En cambio, si $F = A \implies B$, recordando que $A \implies B \approx \neg A \vee B$, entonces F es verdadera cuando:

- A y B son verdaderas,
- A es falsa y B verdadera, o,
- A y B son falsas.

Luego, se puede definir el siguiente conjunto de transiciones.



Como veremos más adelante, al trabajar con máquinas deterministas solo es posible avanzar por una de las transiciones.

Observación 1. Cabe mencionar que las variables (libres) de una restricción se interpretan cuantificadas universalmente. Es decir que si tenemos la transición $q \xrightarrow[x>0]{f\langle int\, x\rangle} q'$, la assertion se interpreta como $(\forall x)x > 0$.

Vamos a recuperar los conceptos de **definición** y **redefinición** de una variable introducidos en la Def. 9, extendiéndolos de manera trivial a las *a-FSM*.

Las restricciones con las que trabajamos son fórmulas de primer orden, en donde sus variables están contenidas en el conjunto de variables usadas en los intercambios de mensajes. Dado que esperamos que estas fórmulas prediquen sobre propiedades de esas variables, vamos a presentar definiciones que nos ayuden a limitar el alcance de las mismas a lo largo de una ejecución, con la finalidad de que los autómatas con los que trabajemos estén bien definidos.

Definición 13 (Variables conocidas)**.** Sea M una *a-FSM* y $\mathcal{P} = p_0 \xrightarrow[A_0]{l_0} p_1 \to \cdots \to p_n \xrightarrow[A_n]{l_n} p_{n+1}$ un camino en M. Definimos al conjunto de **variables conocidas** en $\mathcal{P}$, escrito $\mathcal{V}(\mathcal{P})$, como el conjunto conformado por las variables en los labels $l_0, \ldots, l_n$. Formalmente

$$\mathcal{V}(\mathcal{P}) := \bigcup_{i=1,\ldots,n} \{\, v \mid v \in vars(l_i) \,\}$$

Decimos que v **es conocida** en $\mathcal{P}$ si $v \in \mathcal{V}(\mathcal{P})$.

Dada una transición $t = p \xrightarrow[A]{l} p'$ en una *a-FSM*, la assertion A solo debería poder predicar en términos de las variables conocidas en cada uno de los caminos existentes para llegar hasta p. Notemos que si $v \in vars(A)$ y $v \notin vars(l)$, no alcanza con que v sea conocida en **algún** camino hacia p, ya que de existir un camino $\mathcal{P} = p_0 \to \cdots \to p$ tal que $v \notin \mathcal{V}(\mathcal{P})$, existirían trazas para las que t no está bien definida (tenemos una variable con valor indefinido).
Definamos entonces la siguiente propiedad.

Definición 14 (Sensitividad a la historia (en *a-FSM*))**.** Sea M una *a-FSM*, $t = p \xrightarrow[A]{l} p'$ una transición en M y v una variable. Análogo a la Def. 13, decimos que v **es conocida** en t si:

1. t define a v, o
2. $(\forall \mathcal{P})\mathcal{P} \in \mathcal{SP}(p) \implies v \in \mathcal{V}(\mathcal{P})$

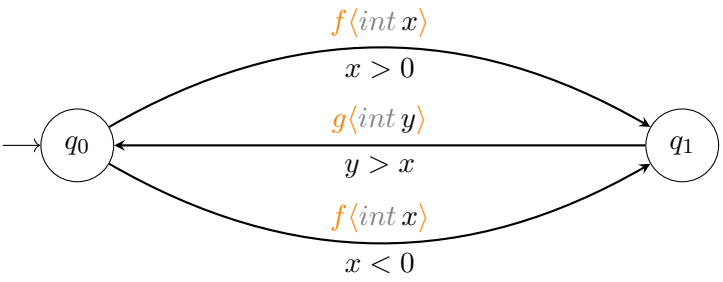
Decimos que M es **sensible a la historia** si toda variable en A es conocida en t, es decir si

$$(\forall v)v \in vars(A) \implies v \text{ es conocida en } t$$

Esta condición garantiza que la assertion de una transición **no puede** predicar sobre variables no definidas.

Por otro lado sabemos que para construir una relación de bisimulación entre dos *a-FSM* es necesario tener en consideración las restricciones (o assertions) acumuladas a lo largo de los caminos existentes para llegar a un estado. Vamos a ver que, el hecho de tener redefiniciones, influye sobre dichas restricciones.

Una redefinición puede darse, por ejemplo, si tenemos un ciclo como el siguiente:.



En este autómata observamos que la transición $t = q_0 \xrightarrow[x>0]{f\langle int\, x\rangle} q_1$ redefine a la variable x al avanzar hacia q_1 por segunda vez, como puede ser el caso de un camino $\mathcal{P} = q_0 \xrightarrow[x>0]{f\langle int\, x\rangle} q_1 \xrightarrow[y>x]{g\langle int\, y\rangle} q_0 \xrightarrow[x<0]{f\langle int\, x\rangle} q_1$.

Analicemos la conjunción de las assertions acumuladas a lo largo del camino $\mathcal{P}$.

1. Ninguna transición: $True$ (vale todo, porque todavía no se agregó ninguna restricción)
2. Después de la primera transición: $x > 0$
3. Después de la segunda transición: $x > 0 \wedge y > x$
4. Después de la tercera transición: $x > 0 \wedge y > x \wedge x < 0$

La fórmula resultante al avanzar con la tercera transición es la correspondiente al estado q_1, después de haber atravesado un ciclo del autómata y haber avanzado sobre la transición restante (la correspondiente a la restricción $x < 0$). Esto nos indicaría que para haber llegado hasta q_1 después de haber pasado por el ciclo tiene que existir un entero x que cumpla $x > 0 \wedge x < 0$; es evidente que esto no es posible. Esta contradicción proviene de haber redefinido a la variable x en la última transición de $\mathcal{P}$, y no haber tenido ninguna consideración sobre las restricciones que acumulamos respecto de x.

El tipo de autómata con el que estamos trabajando **no tiene memoria**, es decir que no es posible que las assertions prediquen en términos de encarnaciones o instancias de una variable (no se puede hablar en términos de x_n, el $n - esimo$ valor que tomó x). Sin embargo, sabemos que la validez de una restricción depende solo del **último** valor que haya podido tomar cada una de las variables que la componen (a lo largo de una ejecución). Por lo tanto, al momento de redefinir una variable x es necesario descartar aquellas aserciones correspondientes a definiciones previas de la variable.

Esta idea de "conjunción de assertions disponibles para un camino" va a ser clave a la hora de definir una noción de bisimulación, por lo tanto, pasemos a definirla formalmente de la siguiente manera.

Definición 15 (Conocimiento)**.** Sea M una *a-FSM* y $\mathcal{P} = p_0 \xrightarrow[A_0]{l_0} p_1 \to \cdots \to p_{n-1} \xrightarrow[A_{n-1}]{l_{n-1}} p_n \xrightarrow[A_n]{l_n} p_{n+1}$ un camino en M. Definimos al **conocimiento** de $\mathcal{P}$, escrito $\mathcal{K}(\mathcal{P})$, como:

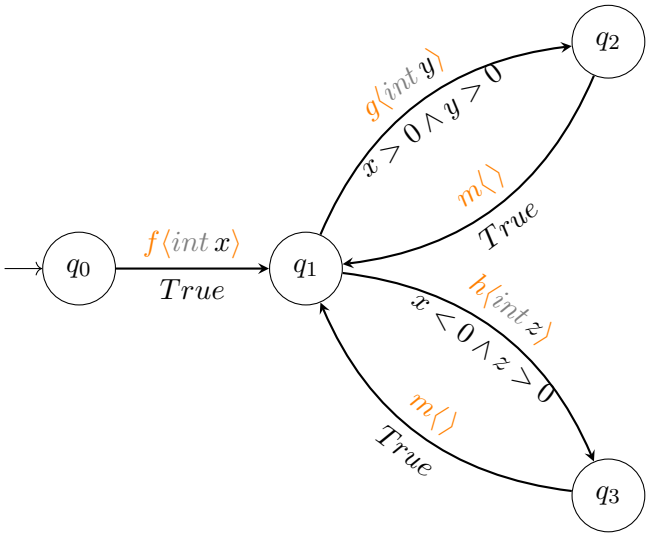
$$\mathcal{K}(\mathcal{P}) := \begin{cases} True & \text{si } n = 0 \\ \mathcal{K}(p_0 \xrightarrow[A_0]{l_0} p_1 \to \cdots \to p_{n-1} \xrightarrow[A_{n-1}]{l_{n-1}} p_n) \setminus l_n \wedge A_n & \text{si } n > 0 \end{cases}$$

donde

$$K \setminus l := \begin{cases} True & \text{si } K = True \\ K' \setminus l & \text{si } K = p \wedge K' \text{ y } vars(l) \cap vars(p) \neq \emptyset \\ p \wedge K' \setminus l & \text{si } K = p \wedge K' \text{ y } vars(l) \cap vars(p) = \emptyset \end{cases}$$

con K una conjunción de literales.

Ejemplo 4 (Conocimientos para los distintos caminos hacia un estado)**.**



Si tenemos

$$\mathcal{P}_0 = q_0 \xrightarrow[True]{f\langle int\, x\rangle} q_1$$

$$\mathcal{P}_1 = \mathcal{P}_0 \xrightarrow[x>0 \wedge y>0]{g\langle int\, y\rangle} q_2$$

$$\mathcal{P}_2 = \mathcal{P}_1 \xrightarrow[True]{m\langle\rangle} q_1$$

$$\mathcal{P}_3 = \mathcal{P}_2 \xrightarrow[x>0 \wedge y>0]{g\langle int\, y\rangle} q_2$$

el conocimiento en cada camino nos queda

$$\begin{aligned}
\mathcal{K}(\mathcal{P}_0) &= \mathcal{K}(q_0) \setminus f\langle int\, x\rangle \wedge True = True \\
\mathcal{K}(\mathcal{P}_1) &= \mathcal{K}(\mathcal{P}_0) \setminus g\langle int\, y\rangle \wedge x > 0 \wedge y > 0 = x > 0 \wedge y > 0 \\
\mathcal{K}(\mathcal{P}_2) &= \mathcal{K}(\mathcal{P}_1) \setminus m\langle\,\rangle \wedge True = x > 0 \wedge y > 0 \\
\mathcal{K}(\mathcal{P}_3) &= \mathcal{K}(\mathcal{P}_2) \setminus g\langle int\, y\rangle \wedge x > 0 \wedge y > 0 \\
&= (x > 0 \wedge y > 0) \setminus g\langle int\, y\rangle \wedge x > 0 \wedge y > 0 \\
&= (x > 0) \wedge x > 0 \wedge y > 0 = x > 0 \wedge y > 0
\end{aligned}$$

En el camino $\mathcal{P}_3$, la variable y es redefinida, por lo tanto los literales correspondientes a esa variable son borrados del conocimiento.
Notemos además que el camino $\mathcal{P}_1 \xrightarrow[x<0 \wedge z>0]{h\langle int\, z\rangle} q_3$ no es accesible, ya que la transición nos indica que solo se puede avanzar cuando $x < 0$, y por $\mathcal{K}(\mathcal{P}_1)$ sabemos que vale $x > 0$.

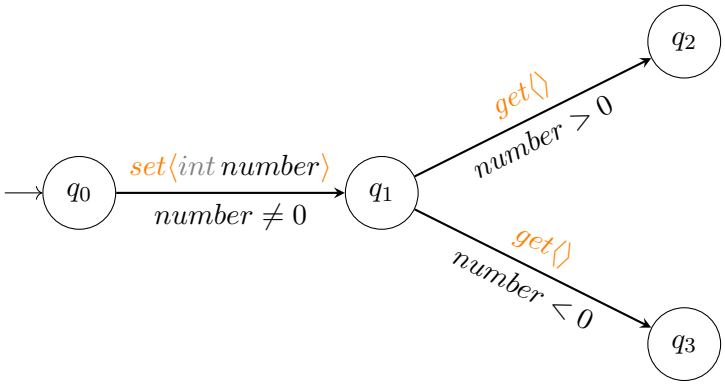
Es importante remarcar que vamos a trabajar con *a-FSM* determinísticas, y dado que las transiciones en estos autómatas pueden estar acompañadas por assertions, el determinismo pasa a ser más amplio que solo el hecho de que las transiciones salientes de un estado deban estar etiquetadas distintas.

Definición 16 (Determinismo en una *a-FSM*)**.** Sea M una *a-FSM*. Decimos que M es determinística, si para todo par de transiciones $t_1 = q \xrightarrow[A_1]{l} q'$, $t_2 = q \xrightarrow[A_2]{l} q''$ de M se cumple que:

$$t_1 \neq t_2 \implies ((\forall \sigma)\sigma \models A_1 \implies \sigma \not\models A_2)$$

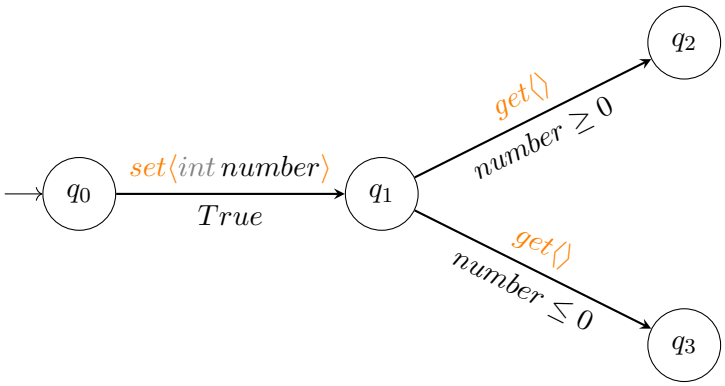
Es decir que los conjuntos de valores que satisfacen a cada assertion son disjuntos.

Ejemplo 5 (*a-FSM* determinística)**.**



La transición $q_0 \xrightarrow[number \neq 0]{set\langle int\, number\rangle} q_1$ restringe las valuaciones posibles para la variable *number* a los distintos a cero. $q_1 \to q_2$ restringe a los mayores a cero, mientras que $q_1 \to q_3$ lo hace con los menores a cero. Estas últimas dos transiciones presentan un mismo label, sin embargo trabajan sobre conjuntos de valuaciones disjuntos, por lo que el autómata es determinista.

Ejemplo 6 (*a-FSM* NO determinística)**.**



En este caso la transición $q_0 \xrightarrow[True]{set\langle int\ number\rangle} q_1$ no restringe las valuaciones para la variable *number* (son todos los enteros posibles). Las transiciones $q_1 \xrightarrow[number \geq 0]{get\langle\rangle} q_2$ y $q_1 \xrightarrow[number \leq 0]{get\langle\rangle} q_3$ representan dos conjuntos de valores, uno en donde los enteros son mayores o iguales a cero, y otro en donde son menores o iguales a cero (respectivamente). La intersección de estos conjuntos tiene como único elemento al $x = 0$, lo que deja a la traza $set\langle 0\rangle.get\langle\rangle$ con dos caminos válidos y, por ende, el autómata no puede ser determinista.

Notar que, según la Def. 16, el autómata es **no** determinístico aún si cambiamos la assertion en la transición $q_0 \xrightarrow[True]{set\langle int\ number\rangle} q_1$ por $x \neq 0$.

A continuación vamos a pasar a definir, no una, sino tres nociones alternativas de bisimulación. La idea será iterar a partir de las limitaciones de una primera definición (que podríamos denominar como "trivial"), siendo esta la más restrictiva, encontrar una segunda; iterar nuevamente sobre sus respectivas limitaciones y encontrar una tercera y última definición, más flexible que las anteriores. Con esta última definición abordaremos el problema de *matching* de nombres, tanto a nivel teórico como práctico.

Por último, cabe mencionar que construiremos las definiciones de bisimulación para autómatas que cumplan las propiedades de *determinismo* y *sensibilidad a la historia*.

3.2. Bisimulación sensible a las transiciones

Para construir una noción de *bisimulación* (para *a-FSM*), previamente necesitamos construir una idea de *simulación* (ya que podemos definir una bisimulación a partir de una definición de simulación). Repasemos la idea de simulación en la Def. 4. Decimos que $\mathcal{S}$ es una relación de simulación si, para todo par $p\,\mathcal{S}\,q$ vale que: para toda transición $p \xrightarrow{l} p'$, existe una transición $q \xrightarrow{l} q'$ y $p'\,\mathcal{S}\,q'$.

Veamos como podemos extender esta idea para abarcar las *a-FSM*. Como vimos al comienzo del capítulo (y terminamos de definir en la Def. 15) necesitamos considerar al conocimiento ($\mathcal{K}$) de un estado como parte de la relación. Esto quiere decir que necesitamos poder comparar si el conocimiento de un estado simula correctamente al de otro. A tales efectos, introduciremos lo que llamaremos **condición de simulación del conocimiento** (cuya definición posponemos por el momento).

Decimos que $\mathcal{S}$ es una relación de simulación *sensible a las transiciones* si, para todo par $(p, K_p)\,\mathcal{S}\,(q, K_q)$ vale que: para toda transición $p \xrightarrow[A]{l} p'$, existe una transición $q \xrightarrow[B]{l} q'$ tal que se cumple la *condición de simulación del conocimiento* y $(p', K_p \setminus l \wedge A)\,\mathcal{S}\,(q', K_q \setminus l \wedge B)$.

Para un par $(p, K_p)\,\mathcal{S}\,(q, K_q)$ decimos que q, con conocimiento K_q **simula** a p con conocimiento K_p; o bien que p con conocimiento K_p **es simulado** por q con conocimiento K_q.

La *condición de simulación del conocimiento* debe asegurarnos que todos los valores que cumplen la restricción $K_q \setminus l \wedge B$ son **por lo menos** los mismos que cumplen la restricción $K_p \setminus l \wedge A$. De esta manera el estado que simula (q en este caso) permitiría que ocurran como mínimo las mismas transiciones existentes desde el estado a simular (p en este caso).

Sean σ y ρ valuaciones, formalmente lo que buscamos es una condición que nos asegure que lo siguiente se cumple:

$$(\forall\sigma)(\exists\rho)\,\sigma \models K_p \wedge \rho \models K_q \wedge$$
$$\sigma[vars(l) \mapsto \overline{v}] \models (K_p \setminus l \wedge A) \implies \rho[vars(l) \mapsto \overline{v}] \models (K_q \setminus l \wedge B)$$

De este requerimiento podemos extraer que la condición de simulación buscada puede ser:

$$K_p \setminus l \wedge A \implies K_q \setminus l \wedge B \tag{3.1}$$

Finalmente, podemos definir la siguiente noción de bisimulación para *a-FSM*

Definición 17 (Bisimulación entre *a-FSM* (sensible a las transiciones))**.** Sean $M_1 = \langle Q_{M_1}, q_{0_{M_1}}, \mathcal{M} \times \mathcal{A}, \rightarrow_{M1} \rangle$, $M_2 = \langle Q_{M_2}, q_{0_{M_2}}, \mathcal{M} \times \mathcal{A}, \rightarrow_{M_2} \rangle$ dos *a-FSM deterministas* y *sensibles a la historia*. Una relación $\mathcal{R} \subseteq ((Q_{M_1} \times \mathcal{A}) \times (Q_{M_2} \times \mathcal{A}))$ es una simulación si

para todo par $(p, K_p)\,\mathcal{R}\,(q, K_q)$, para toda etiqueta l y para toda assertion A vale que:

$$(p \xrightarrow[A]{l} p') \implies (\exists q')(\exists B)$$
$$q \xrightarrow[B]{l} q' \wedge$$
$$(K_p \setminus l \wedge A \implies K_q \setminus l \wedge B) \wedge$$
$$(p', \overline{K_p \setminus l \wedge A})\,\mathcal{R}\,(q', \overline{K_q \setminus l \wedge B})$$

M_1 y M_2 son bisimilares, escrito $M_1 \approx M_2$, si existe una simulación $\mathcal{R}$ tal que: $(q_{0_{M_1}}, True)\,\mathcal{R}\,(q_{0_{M_2}}, True)$, y $\mathcal{R}^{-1}$ es una simulación.

Notemos la definición anterior elimina la redundancia en las fórmulas que componen a los conocimientos, al momento de verificar la última condición de simulación (es decir, cuando pedimos que se cumpla $(p', \overline{K_p \setminus l \wedge A})\,\mathcal{R}\,(q', \overline{K_q \setminus l \wedge B})$), evitando de esta manera que la definición se vuelva infinita. Ilustremos esto, por ejemplo, mediante el siguiente ciclo escrito como un camino:

$$\mathcal{P}^0 = p_0 \xrightarrow[A]{l} p_1 \xrightarrow[B]{l'} p_0$$

al recorrerlo (como lo haría la definición) nos queda: $\mathcal{K}(\mathcal{P}^0) = True \wedge A \wedge B$.
Y si queremos recorrerlo una segunda vez, mediante

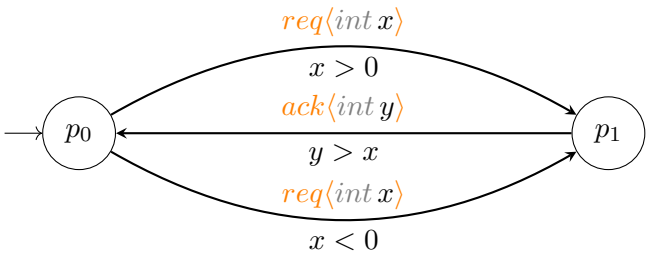
$$\mathcal{P}^1 = \mathcal{P} \rightarrow \mathcal{P}$$

nos queda: $\mathcal{K}(\mathcal{P}^1) = True \wedge A \wedge B \wedge A \wedge B$.
Si queremos recorrerlo por $n - esima$ vez, mediante

$$\mathcal{P}^n = \mathcal{P}^{n-1} \rightarrow \mathcal{P}$$

nos queda: $\mathcal{K}(\mathcal{P}^n) = True \bigwedge_{0,\ldots,n} A \wedge B$.
Al ser Def. 17 recursiva, solo una relación $\mathcal{R}$ infinita podría contener elementos de esa forma. Con lo cual, eliminar la redundancia nos elimina (a su vez) este problema.

Ejemplo 7 (Debe ser bisimilar a si mismo)**.**

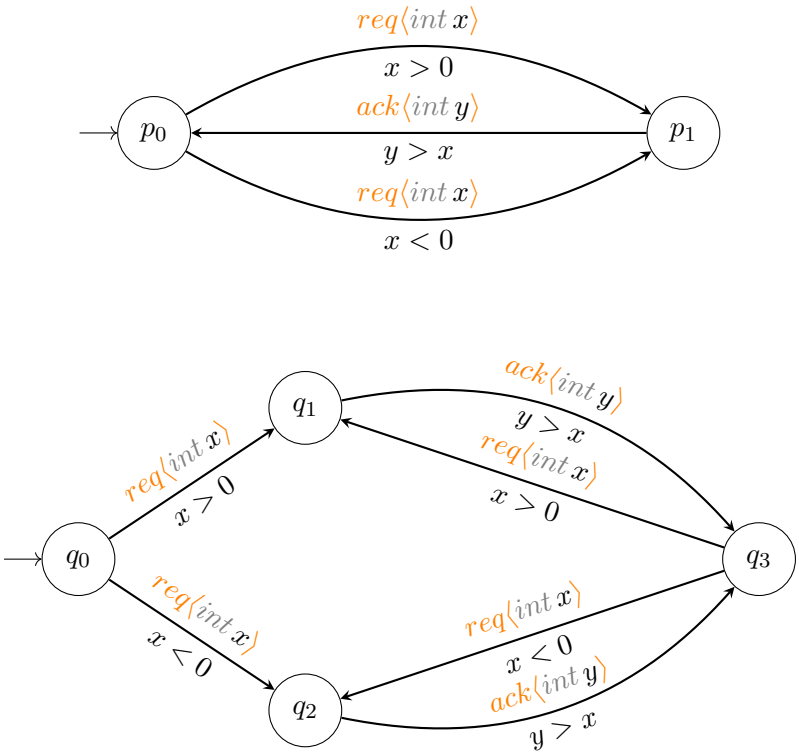


$$\begin{aligned}
\mathcal{R} = \{ & \\
& ((p_0, True), (p_0, True)), \\
& ((p_1, True \wedge x < 0), (p_1, True \wedge x < 0)), \\
& ((p_0, True \wedge x < 0 \wedge y > x), (p_0, True \wedge x < 0 \wedge y > x)), \\
& ((p_1, True \wedge x > 0), (p_1, True \wedge x > 0)), \\
& ((p_0, True \wedge x > 0 \wedge y > x), (p_0, True \wedge x > 0 \wedge y > x)) \\
\}
\end{aligned}$$

Si $K_p = x > 0 \wedge y > x$ en p_0, al avanzar sobre $req\langle int\, x\rangle$ (en cualquiera de las dos transiciones desde p_0) se pierden los literales que predicaban sobre x:

$$K_p \setminus l = x > 0 \wedge y > x \setminus req\langle int\, x\rangle = True$$

Ejemplo 8 (Autómatas bisimilares)**.**

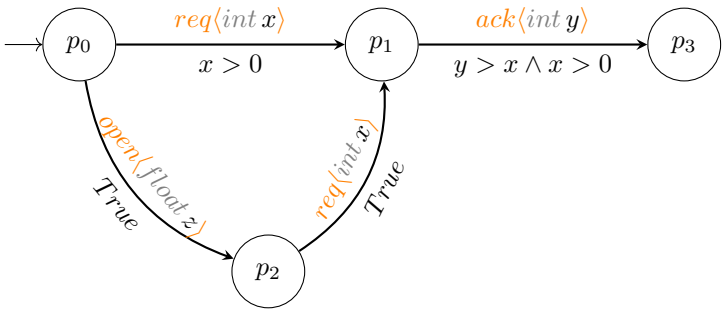


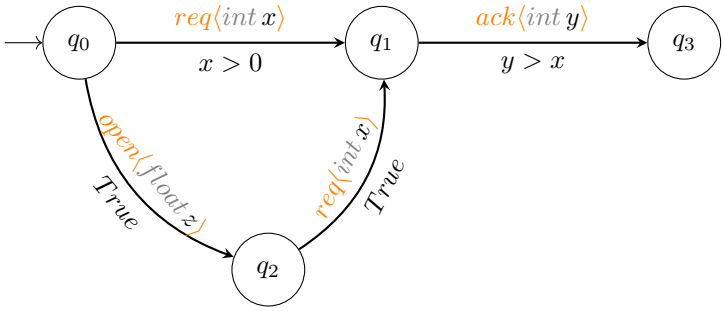
$$\mathcal{R} = \{$$

$$((p_0, True), (q_0, True)),$$
$$((p_1, True \wedge x < 0), (q_1, True \wedge x < 0)),$$
$$((p_0, True \wedge x < 0 \wedge y > x), (q_0, True \wedge x < 0 \wedge y > x)),$$
$$((p_1, True \wedge x > 0), (q_1, True \wedge x > 0)),$$
$$((p_0, True \wedge x > 0 \wedge y > x), (q_0, True \wedge x > 0 \wedge y > x))$$

$$\}$$

Ejemplo 9 (Autómatas no bisimilares)**.**





Sabemos que para que exista una relación $\mathcal{R}$ que describa una bisimulación entre dos autómatas, se tiene que cumplir que:

1. $(p_0, True)\,\mathcal{R}\,(q_0, True)$

lo cual, siguiendo la definición presentada, nos lleva a necesitar que se cumpla:

2. $(p_2, True)\,\mathcal{R}\,(q_2, True)$
3. $(p_1, True)\,\mathcal{R}\,(q_1, True)$
4. $(p_3, True \wedge y > x \wedge x > 0)\,\mathcal{R}\,(q_3, True \wedge y > x)$

pero, para que se cumpla (4), se tiene que cumplir:

4.1. $y > x \wedge x > 0 \implies y > x$

4.2. $y > x \implies y > x \wedge x > 0$

Es claro que (4.1) se cumple, ya que $y > x$ se puede deducir de $y > x \wedge x > 0$, pero como $x > 0$ no se puede deducir de $y > x$ entonces (4.2) no se cumple. Luego, no puede existir una relación $\mathcal{R}$ para este par de autómatas y, por lo tanto, no son bisimilares.

Otra forma de evidenciar esto puede ser mediante, por ejemplo, una traza $T = open\langle 1\rangle.req\langle 0\rangle.ack\langle 1\rangle$. Notemos que T es una traza disponible en el segundo autómata, pero no en el primero, ya que la transición que va desde p_1 a p_3 requiere que los valores de la variable x (obtenidos mediante la función req) sean estrictamente mayores a cero.

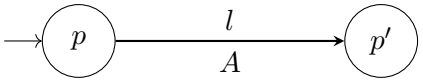
3.2.1. Límites de la definición

Supongamos que tenemos un estado p en una *a-FSM* M_1, con conocimiento K_p, en donde existen transiciones desde p de la forma $p \xrightarrow[A_i]{l} p'$ con $i = 1, \ldots, n$. Supongamos también otro estado q en una *a-FSM* M_2, con conocimiento K_q en donde existen transiciones desde q de la forma $q \xrightarrow[B_j]{l} q'$ con $j = 1, \ldots, m$. Asumamos M_1, M_2 deterministas, con lo cual las assertions en las transiciones de cada estado deben ser disjuntas.
Si existe una relación $\mathcal{R}$ tal que $M_1 \approx M_2$, y suponemos que $(p, K_p)\,\mathcal{R}\,(q, K_q)$, entonces para cada $p \xrightarrow[A_k]{l} p'$ existe un $q \xrightarrow[B_h]{l} q'$ que lo simula, con k, h fijos, $k \in \{1, \ldots, n\}$ y $h \in \{1, \ldots, m\}$. Como hay bisimilaridad, también vale la vuelta (las transiciones desde p simulan a las transiciones desde q). Lo que esto nos muestra, es que debe existir una biyección entre las transiciones desde p y las transiciones

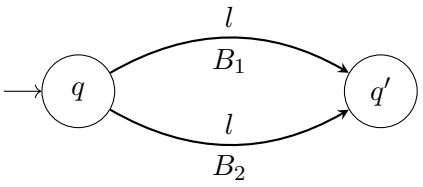
desde q y que, por lo tanto, $m = n$, ya que cada transición desde un estado puede simular solo a **una** de las que salen desde el otro estado.

Al pedir, de manera indirecta, que exista esta biyección, dejamos afuera de la definición casos como el siguiente:

Ejemplo 10 (Autómatas no bisimilares debido a los límites en la definición)**.**



(a) M_1



(b) M_2

Si vale $A \iff B_1 \vee B_2$, entonces el comportamiento de M_1 es equivalente al de M_2. Sin embargo, la bisimulación entre este par de autómatas no logra ser capturada por la Def. 17. Al verificar si q es simulable por p, llegamos a la condición $B_1 \implies A$ lo cual no puede ser cierto, ya que B_1 es mas fuerte que A.

Encontramos como límite, el hecho de no considerar bisimilitud en casos en donde los valores que cumplen los conocimientos son equivalentes, pero están representados de manera distinta en cada autómata.

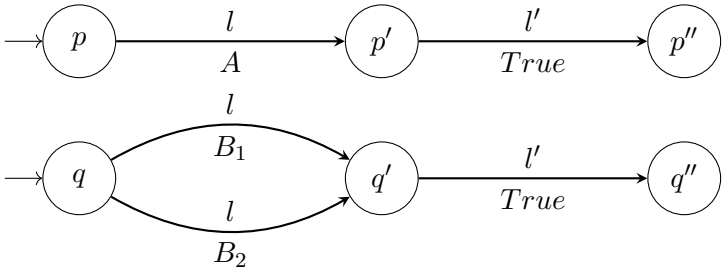
3.3. Bisimulación insensible a particiones en assertions

Vamos a construir una nueva condición de simulación, junto con una nueva condición de simulación del conocimiento, que permita que una transición pueda ser simulada por un conjunto de otras transiciones (en lugar de una sola). Decimos que $\mathcal{S}$ es una relación de simulación *insensible a particiones en assertions* si, para todo par $(p, K_p)\,\mathcal{S}\,(q, K_q)$ vale que: para toda transición $p \xrightarrow[A]{l} p'$, existe un *conjunto* $T = \{\, q \xrightarrow[B_i]{l} q' \mid i = 1, \ldots, n \,\}$ tal que se cumple la *condición de simulación del conocimiento* (formalizada mas abajo) y $(p', K_p \setminus l \wedge A)\,\mathcal{S}\,(q', K_q \setminus l \wedge B_i)$ para todo $i = 1, \ldots, n$.

La discusión para la búsqueda de la *condición de simulación del conocimiento* es similar a la establecida en la Sec. 3.2, en donde ahora el estado q, quien simula, lo hace utilizando n transiciones en lugar de una sola. Para abarcar esto, vamos a pedir que esta condición asegure que todos los valores que cumplen la restricción $K_q \setminus l \wedge \bigvee_{i=1,\ldots,n} B_i$ sean **por lo menos** los mismos que cumplen la restricción $K_p \setminus l \wedge A$, es decir que se cumpla:

$$K_p \setminus l \wedge A \implies K_q \setminus l \wedge \bigvee_{i=1,\ldots,n} B_i \tag{3.2}$$

Si bien esto parece una extensión natural de la condición descrita en la sección anterior, resulta no ser suficiente para este caso. Veamos el siguiente ejemplo:



Supongamos que vale $A \iff B_1 \vee B_2$, es claro que la transición $p \xrightarrow[A]{l} p'$ puede ser simulada mediante las transiciones $q \xrightarrow[B_1]{l} q'$ y $q \xrightarrow[B_2]{l} q'$. Sin perdida de generalidad, centrémonos en la transición cuya assertion es B_1. Sabemos que también debe valer que $(p', K_p \setminus l \wedge A)\,\mathcal{S}\,(q', K_q \setminus l \wedge B_1)$. Por lo descrito en la ecuación 3.2, necesitaríamos que valga:

$$K'_p \setminus l' \wedge True \implies K'_q \setminus l' \wedge True$$

es decir: $A \implies B_1$. Sin embargo esto no puede ser cierto, ya que B_1 es una condición más *fuerte* por hipótesis.

Para que esto funcione lo que nos está faltando es restringir el conocimiento simulado, al conocimiento que simula. Luego, la nueva condición de simulación del conocimiento nos queda determinada de la siguiente manera:

$$(K_p \wedge K_q) \setminus l \wedge A \implies K_q \setminus l \wedge \bigvee_{i=1,\ldots,n} B_i \tag{3.3}$$

Finalmente, podemos llegar a la siguiente definición de bisimulación.

Definición 18 (Bisimulación entre *a-FSM* (insensible a particiones en assertions))**.** Sean $M_1 = \langle Q_{M_1}, q_{0_{M_1}}, \mathcal{M} \times \mathcal{A}, \rightarrow_{M_1} \rangle$, $M_2 = \langle Q_{M_2}, q_{0_{M_2}}, \mathcal{M} \times \mathcal{A}, \rightarrow_{M_2} \rangle$ dos *a-FSM deterministas* y *sensibles a la historia*. Una relación $\mathcal{R} \subseteq ((Q_{M_1} \times \mathcal{A}) \times (Q_{M_2} \times \mathcal{A}))$ es una simulación si

para todo par $(p, K_p)\,\mathcal{R}\,(q, K_q)$, para toda etiqueta l y para toda assertion A vale que:

$$
\begin{aligned}
&(p \xrightarrow[A]{l} p') \implies (\exists q')(\exists T) \\
&\quad T = \{\, q \xrightarrow[B_i]{l} q' \,\}_{i \in I \neq \emptyset} \subseteq \rightarrow_{M_1} \ \wedge \\
&\quad \Big((K_p \wedge K_q) \setminus l \wedge A \implies K_q \setminus l \wedge \bigvee_{i \in I} B_i \Big) \wedge \\
&\quad (\forall i \in I)(p', \overline{K_p \setminus l \wedge A})\,\mathcal{R}\,(q', \overline{K_q \setminus l \wedge B_i})
\end{aligned}
$$

M_1 y M_2 son bisimilares, escrito $M_1 \approx M_2$, si existe una simulación $\mathcal{R}$ tal que: $(q_{0_{M_1}}, True)\,\mathcal{R}\,(q_{0_{M_2}}, True)$, y $\mathcal{R}^{-1}$ es una simulación.

Ejemplo 11 (Bisimulación de una transición utilizando más de una)**.** Veamos un ejemplo de un par de *a-FSM* en donde los valores que pueden tomar las variables en algunas transiciones son simulados utilizando más de una transición.

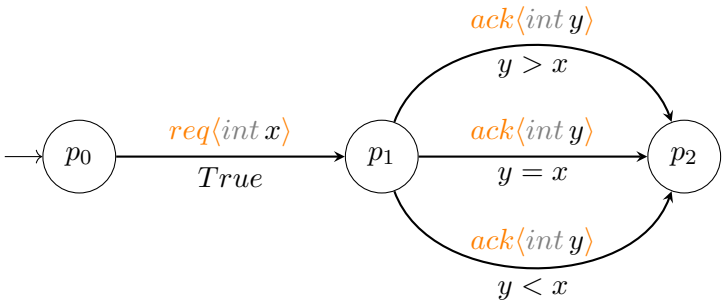


Fig. 3.2: M_1

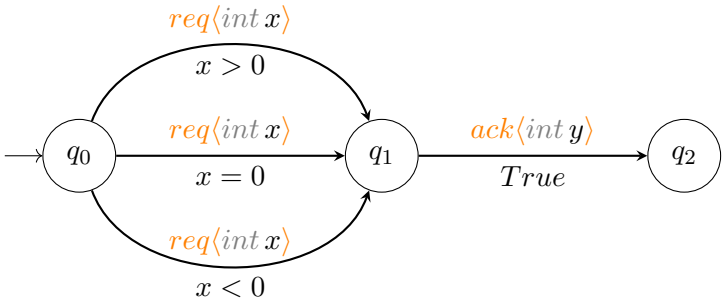


Fig. 3.3: M_2

Notemos que $True \iff x > 0 \vee x = 0 \vee x < 0$ y $True \iff y > x \vee y = x \vee y < x$, con lo cual las valuaciones entre las transiciones de los autómatas son equivalentes. La siguiente relación muestra que $M_1 \approx M_2$ según la definición presentada.

$\mathcal{R} = \{$

$$
\begin{aligned}
&((p_0, True), (q_0, True)),\\
&((p_1, True), (q_1, x > 0)),\\
&\quad ((p_2, True \wedge y > x), (q_1, x > 0 \wedge True)),\\
&\quad ((p_2, True \wedge y = x), (q_1, x > 0 \wedge True)),\\
&\quad ((p_2, True \wedge y < x), (q_1, x > 0 \wedge True)),\\
&((p_1, True), (q_1, x = 0)),\\
&\quad ((p_2, True \wedge y > x), (q_1, x = 0 \wedge True)),\\
&\quad ((p_2, True \wedge y = x), (q_1, x = 0 \wedge True)),\\
&\quad ((p_2, True \wedge y < x), (q_1, x = 0 \wedge True)),\\
&((p_1, True), (q_1, x < 0)),\\
&\quad ((p_2, True \wedge y > x), (q_1, x < 0 \wedge True)),\\
&\quad ((p_2, True \wedge y = x), (q_1, x < 0 \wedge True)),\\
&\quad ((p_2, True \wedge y < x), (q_1, x < 0 \wedge True))
\end{aligned}
$$

}

Tomemos por ejemplo el par relacionado $(p_2, True \wedge y > x)\,\mathcal{R}\,(q_2, x > 0 \wedge True)$, cuyos conocimientos representan haber avanzado sobre los siguientes caminos:

$$p_0 \xrightarrow[True]{req\langle int\, x\rangle} p_1 \xrightarrow[y>x]{ack\langle int\, y\rangle} p_2$$

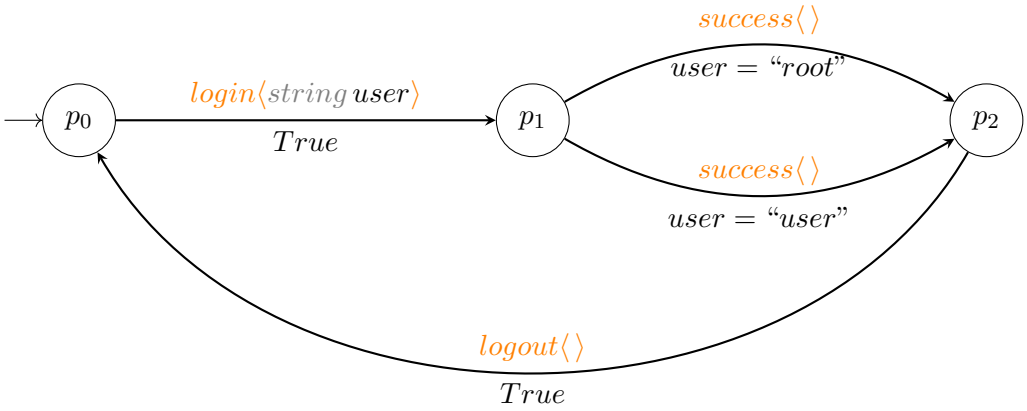
$$q_0 \xrightarrow[x>0]{req\langle int\, x\rangle} q_1 \xrightarrow[True]{ack\langle int\, y\rangle} q_2$$

Estos caminos contemplan solo las valuaciones en las cuáles x es positivo e y es mayor a x, y, lo que ocurre en este caso es que:

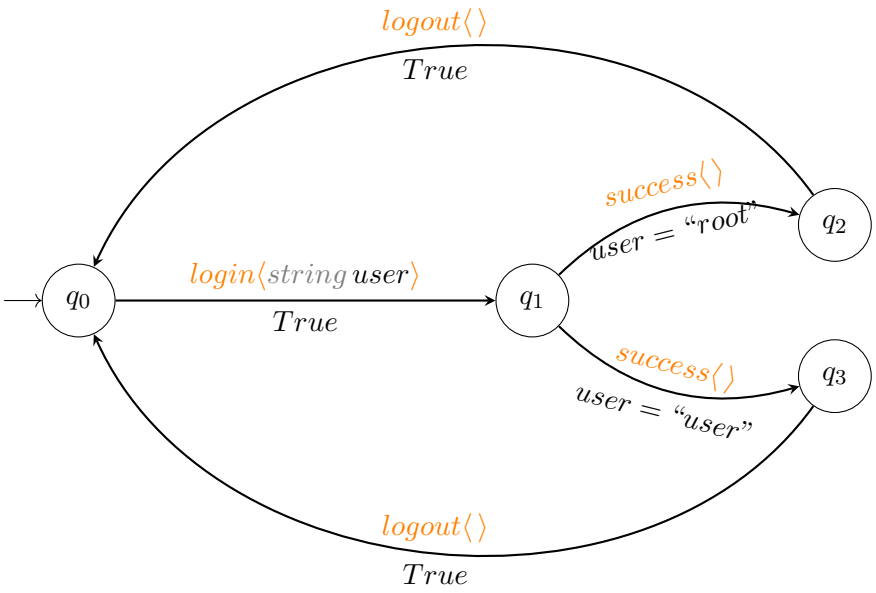
1. q_0 simula, de el conjunto de todos los enteros x producido por la transición en p_0, solo el subconjunto de los enteros $x > 0$ (y el resto mediante otras transiciones), y
2. q_1 simula, **por lo menos**, el subconjunto dado por todos los enteros $y > x$ producido por la transición del camino desde p_1, del conjunto de todos los enteros y (dado por $True$).

3.3.1. Límites de la definición

Supongamos que necesitamos un servicio (muy sencillo) de control de acceso, y tenemos un contrato especificado como el siguiente *a-FSM*



En nuestro repositorio de servicios, tenemos solo una implementación disponible, que cumple con el siguiente contrato:



La única diferencia entre la implementación y nuestro requerimiento, se encuentra en que, quien implementó el servicio, decidió separar el cómputo del usuario *root* de el de los usuarios "normales". Sin embargo, a pesar de que el comportamiento es equivalente, la Def. 18 no logra capturar bisimilitud entre este par de autómatas. El problema surge al no considerar la posibilidad de que uno de los contratos pueda dividir un cómputo en distintas rutinas, dejando como resultado dos (o más) estados.

Dada la utilidad práctica que se busca para este trabajo, permitir que contratos con estas características puedan ser bisimilares podría aportar un gran valor.

3.4. Bisimulación flexible

El salto a partir de la definición anterior no es demasiado complejo. Lo que necesitamos es re-escribir brevemente la condición de simulación presentada en Def. 18, pero manteniendo la misma idea para la condición de simulación del conocimiento. Entonces, decimos que $\mathcal{S}$ es una relación de simulación *flexible* si, para todo par $(p, K_p)\,\mathcal{S}\,(q, K_q)$ vale que: para toda transición $p \xrightarrow[A]{l} p'$, existe un *conjunto* $T = \{\, q \xrightarrow[B_i]{l} q_i \mid i = 1, \ldots, n \,\}$ tal que $(K_p \wedge K_q) \setminus l \wedge A \implies K_q \setminus l \bigvee_{i=1,\ldots,n} B_i$ se cumple y $(p', K_p \setminus l \wedge A)\,\mathcal{S}\,(q_i, K_q \setminus l \wedge B_i)$ para todo $i = 1, \ldots, n$.

Es decir que ahora, permitimos a los estados de llegada en las transiciones de T ser distintos. Notemos que cuando $q_i = q_j$ para todo $i, j = 1, \ldots, n$ tenemos exactamente la definición anterior.

Luego, la definición de bisimulación más flexible se extiende de manera natural de la Def. 18 de la siguiente manera.

Definición 19 (Bisimulación entre *a-FSM* (flexible))**.** Sean $M_1 = \langle Q_{M_1}, q_{0_{M_1}}, \mathcal{M} \times \mathcal{A}, \rightarrow_{M1} \rangle$, $M_2 = \langle Q_{M_2}, q_{0_{M_2}}, \mathcal{M} \times \mathcal{A}, \rightarrow_{M_2} \rangle$ dos *a-FSM deterministas* y *sensibles a la historia*. Una relación $\mathcal{R} \subseteq ((Q_{M_1} \times \mathcal{A}) \times (Q_{M_2} \times \mathcal{A}))$ es una simulación si

para todo par $(p, K_p)\,\mathcal{R}\,(q, K_q)$, para toda etiqueta l y para toda assertion A vale que:

$$
\begin{aligned}
&(p \xrightarrow[A]{l} p') \implies (\exists T) \\
&\quad T = \{\, q \xrightarrow[B_i]{l} q_i \,\}_{i \in I \neq \emptyset} \subseteq \rightarrow_{M_1} \ \wedge \\
&\quad \left((K_p \wedge K_q) \setminus l \wedge A \implies K_q \setminus l \wedge \bigvee_{i \in I} B_i \right) \wedge \\
&\quad (\forall i \in I)(p', \overline{K_p \setminus l \wedge A})\,\mathcal{R}\,(q_i, \overline{K_q \setminus l \wedge B_i})
\end{aligned}
$$

M_1 y M_2 son bisimilares, escrito $M_1 \approx M_2$, si existe una simulación $\mathcal{R}$ tal que: $(q_{0_{M_1}}, True)\,\mathcal{R}\,(q_{0_{M_2}}, True)$, y $\mathcal{R}^{-1}$ es una simulación.

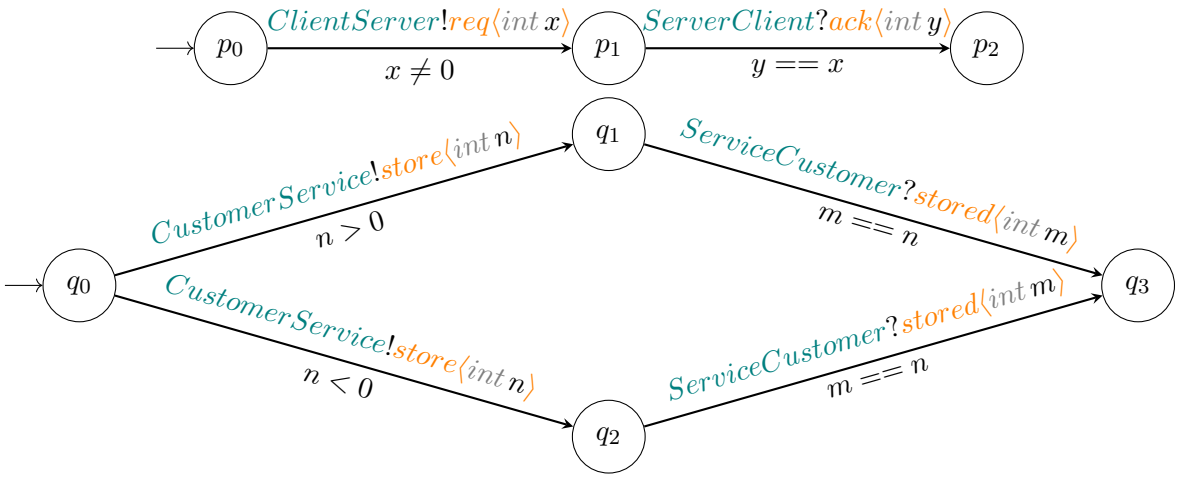
3.5. Extendiendo la definición de bisimulación a *a-CFSM*

Es fácil ver que tanto la propiedad de *sensibilidad a la historia* como las definiciones de bisimulación presentadas para las *a-FSM* son trivialmente extensibles a las *a-CFSM*. En ambos casos solo necesitamos hablar en términos de las variables correspondientes a los mensajes enviados (o intercambiados, en caso de las *a-CFSM*) y de los caminos en los que se envían (o intercambian) estos mensajes. El único momento en el que los participantes de una *a-CFSM* deben ser considerados es al momento de buscar si existe una, o un conjunto, de transiciones desde el estado que simula cuya etiqueta sea la misma que la del estado y la transición a simular. Esto último queda cubierto de manera sencilla mediante una relación de equivalencia entre las etiquetas, es decir, igualdad entre "strings".

Observación 2. Vemos entonces que, si consideramos las etiquetas de las *a-CFSMs* como *strings* atómicos, estas pueden ser consideradas *a-FSM*, y por lo tanto los resultados obtenidos son válidos en este tipo de autómatas.

Como se mencionó durante la introducción (Cap. 1), el paradigma de trabajo asume que el contrato que expresa el requerimiento y la provisión no son (necesariamente) escritos por el mismo agente, y por lo tanto no puede asumirse que utilicen la misma terminología.

Ejemplo 12 (*a-CFSM*s bisimilares, con nombres distintos)**.**



El comportamiento presentado por estos autómatas es equivalente, y podría ser capturado por la Def. 19 si pudiésemos determinar: 1) un *matching* de nombres y 2) una relación de bisimulación, considerando este *matching*.

- $Customer \equiv Client$
- $Service \equiv Server$
- $store \equiv req$
- $stored \equiv ack$
- $n \equiv x$
- $m \equiv y$

$$\begin{aligned}\mathcal{R} = \{ & \\ & ((p_0, True), (q_0, True)), \\ & ((p_1, True \wedge x \neq 0), (q_1, True \wedge n > 0)), \\ & ((p_2, True \wedge x \neq 0 \wedge y == x), (q_1, True \wedge n > 0 \wedge m == n)), \\ & ((p_1, True \wedge x \neq 0), (q_1, True \wedge n < 0)), \\ & ((p_2, True \wedge x \neq 0 \wedge y == x), (q_1, True \wedge n < 0 \wedge m == n)), \\ \} & \end{aligned}$$

A nivel teórico, lo único que necesitamos pedir es que, dadas dos *a-CFSM*, exista una relación de equivalencia o una biyección entre los conjuntos de participantes, mensajes y variables de ambos autómatas. Luego, podemos adaptar la Def. 19 para contemplar el *matching* de nombres de la siguiente manera.

Definición 20 (Bisimulación entre *a-CFSMs*, modulo un *matching* de nombres)**.** Sean $M_1 = \langle Q_{M_1}, q_{0_{M_1}}, \mathcal{L}_{act} \times \mathcal{A}, \rightarrow_{M1} \rangle$, $M_2 = \langle Q_{M_2}, q_{0_{M_2}}, \mathcal{L}_{act} \times \mathcal{A}, \rightarrow_{M_2} \rangle$ dos *a-CFSM deterministas* y *sensibles a la historia*.

Sean $\mathcal{B}_{M_1}$, $\mathcal{M}_{M_1}$, $\mathcal{V}_{M_1}$, $\mathcal{B}_{M_2}$, $\mathcal{M}_{M_2}$, $\mathcal{V}_{M_2}$ los conjuntos de: participantes, mensajes y variables de los autómatas M_1 y M_2 respectivamente.

Decimos que existe una relación de equivalencia entre los participantes, mensajes y variables de M_1 y M_2 si existe una función Φ de la forma:

$$\begin{aligned}
\Phi(p_{M_1}) = p_{M_2} \quad &\text{si} \quad p_{M_1} \in \mathcal{B}_{M_1}, p_{M_2} \in \mathcal{B}_{M_2} \quad \text{y} \quad (\forall p'_{M_1} \in \mathcal{B}_{M_1}) p_{M_1} \neq p'_{M_1} \implies \Phi(p'_{M_1}) \neq p_{M_2} \\
\Phi(m_{M_1}) = m_{M_2} \quad &\text{si} \quad m_{M_1} \in \mathcal{M}_{M_1}, m_{M_2} \in \mathcal{M}_{M_2} \quad \text{y} \quad (\forall m'_{M_1} \in \mathcal{M}_{M_1}) m_{M_1} \neq m'_{M_1} \implies \Phi(m'_{M_1}) \neq m_{M_2} \\
\Phi(v_{M_1}) = v_{M_2} \quad &\text{si} \quad v_{M_1} \in \mathcal{V}_{M_1}, v_{M_2} \in \mathcal{V}_{M_2} \quad \text{y} \quad (\forall v'_{M_1} \in \mathcal{V}_{M_1}) v_{M_1} \neq v'_{M_1} \implies \Phi(v'_{M_1}) \neq v_{M_2}
\end{aligned}$$

y esta es biyectiva.

Una relación $\mathcal{R} \subseteq ((Q_{M_1} \times \mathcal{A}) \times (Q_{M_2} \times \mathcal{A}))$ es una simulación si

existe una función Φ entre M_1 y M_2 que muestra equivalencia entre sus conjuntos de participantes, mensajes y variables,

para todo par $(p, K_p)\,\mathcal{R}\,(q, K_q)$, para toda etiqueta l y para toda assertion A vale que:

$$
\begin{array}{l}
(p \xrightarrow[A]{l} p') \implies (\exists T) \\
\quad T = \{\, q \xrightarrow[B_i]{\Phi(l)} q_i \,\}_{i \in I \neq \emptyset} \subseteq \rightarrow_{M_1} \ \wedge \\
\quad \left(\Phi(K_p \setminus l) \wedge (K_q \setminus \Phi(l)) \wedge \Phi(A) \implies K_q \setminus \Phi(l) \wedge \bigvee_{i \in I} B_i \right) \wedge \\
\quad (\forall i \in I)(p', \overline{K_p \setminus l \wedge A})\,\mathcal{R}\,(q_i, \overline{K_q \setminus \Phi(l) \wedge B_i})
\end{array}
$$

M_1 y M_2 son bisimilares, escrito $M_1 \approx M_2$, si existe una simulación $\mathcal{R}$ tal que: $(q_{0_{M_1}}, True)\,\mathcal{R}\,(q_{0_{M_2}}, True)$, y $\mathcal{R}^{-1}$ es una simulación y Φ^{-1} es su función de equivalencia entre los conjuntos de participantes, mensajes y variables para M_2 y M_1. Además, extendemos la función Φ de la siguiente manera (para etiquetas y fórmulas):

$$
\begin{array}{rl}
\Phi(cs!m\langle t_1\, v_1, \ldots, t_n\, v_n\rangle) = \Phi(c)\Phi(s)!\Phi(m\langle t_1\, v_1, \ldots, t_n\, v_n\rangle) & \text{si} \quad cs!m\langle t_1\, v_1, \ldots, t_n\, v_n\rangle \in \mathcal{L}_{act} \\
\Phi(cs?m\langle t_1\, v_1, \ldots, t_n\, v_n\rangle) = \Phi(c)\Phi(s)?\Phi(m\langle t_1\, v_1, \ldots, t_n\, v_n\rangle) & \text{si} \quad cs?m\langle t_1\, v_1, \ldots, t_n\, v_n\rangle \in \mathcal{L}_{act} \\
\Phi(A) = A[v \mapsto \Phi(v)] \quad (\forall v) v \in var(A) & \text{si} \quad A \in \mathcal{A}
\end{array}
$$

4. CÓMPUTO DE LA RELACIÓN ASSERTED BISIMULATION

Como complemento a los resultados obtenidos se desarrolló una herramienta con la finalidad de poner en práctica el resultado capturado mediante la Def. 20. Se trata de un programa escrito utilizando *python* como lenguaje, en su versión *3.6.5*, que aprovecha las capacidades ofrecidas por la biblioteca de *z3-solver* para modelar y probar la veracidad de predicados en lógica de primer orden. La herramienta se compone de tres módulos (principalmente): 1) modelo de *a-CFSM*, 2) resolución de *matching* de nombres, 3) algoritmo de bisimulación (a partir de los dos puntos anteriores).

En este capítulo repasaremos el algoritmo de bisimulación utilizado por la herramienta, encargado de generar el *matching* de nombres a medida que construye una relación de bisimulación. Para esto primero haremos un resumen de las estructuras utilizadas para modelar las *a-CFSM* y a continuación un pseudocódigo a modo de descripción (del algoritmo). Además realizaremos un conjunto de pruebas con el fin de evaluar el rendimiento de la herramienta, con alguna medida de rendimiento a definir.

Es importante aclarar que este trabajo no se enfoca en la optimización del código, sino en el hecho de que este sea funcional al problema a resolver. Aún así fue necesario avanzar sobre algunas mejoras de rendimiento, ya que de otra forma el algoritmo se volvía impracticable en cuanto los casos comenzaban a escalar ligeramente. Estas optimizaciones se centran en el cálculo de la relación inicial, foco en el que detectamos que surgía el principal problema de rendimiento; nos explayaremos sobre esto más adelante.

El código fuente es libre, y puede encontrarse en `https://github.com/diegosenarruzza/bisimulation`.

4.1. Modelo de autómatas

Se define un modelo para las *a-CFSM*, en donde la estructura de los autómatas se compone de: un conjunto estados, un estado inicial, un conjunto transiciones por estado y una lista con los participantes. Al mismo tiempo, las transiciones se componen de: un estado fuente, un estado objetivo, una etiqueta (label) y una assertion. Los labels son modelados como acciones, identificando cada una de las partes que la componen: emisor (string), receptor (string), etiqueta de mensaje (string) y *payload* (array de variables de *z3*).

4.2. Algoritmo de bisimulación

Dado que la finalidad en este capítulo es la de poner en práctica la Def. 20, para el desarrollo de este algoritmo decidimos basarnos en la técnica de estratificación descrita en Def. 5. A partir de esta idea, extendemos la noción de relación inicial ($\sim_0$) para abarcar candidatos de la forma (estado, conocimiento).

4.2.1. Relación inicial

El algoritmo de estratificación parte de una relación en donde se toman todos los elementos candidatos a relacionar de cada uno de los autómatas, y se combinan de todas las formas posibles para obtener una relación inicial. En el caso de los autómatas finitos convencionales, estos candidatos constan solo de los estados de cada autómata. En nuestro caso, necesitamos que los candidatos estén compuestos por un estado, y un posible conocimiento como resultado de haber llegado a dicho estado. Dado que en principio no conocemos todos los caminos posibles para llegar a un estado, y por lo tanto no sabemos

cuáles pueden ser estos conocimientos asociados, tomamos para el cálculo de la relación inicial a todas las combinaciones de assertions disponibles en un autómata. Es decir, si tenemos un estado q y dos assertions A_1, A_2 (en el autómata que contiene a q), los candidatos estarían dados por el resultado del producto cartesiano entre el estado y las assertions: $(q, \emptyset), (q, \{ A_1 \}), (q, \{ A_2 \}), (q, \{ A_1, A_2 \})$. Luego, la relación inicial se calcula como el producto cartesiano entre los candidatos de ambos autómatas.

Más adelante analizaremos la complejidad asociada a generar de esta manera la relación inicial.

Notemos que estamos utilizando conjuntos para representar al conocimiento. Manejar esta estructura facilita el manejo de las assertions desde el punto de vista de la implementación, en lugar de mantener una única conjunción de todas ellas.

4.2.2. Matching de nombres

Recordemos que la idea es asumir que los autómatas a evaluar no tienen porque tener nombres en común, así que iremos construyendo un *matching* de nombres a medida que calculamos una posible relación de bisimulación. Supongamos que queremos comprobar si un elemento (p, K_p) puede ser simulado por otro (q, K_q), más específicamente tenemos que existe una transición $p \xrightarrow[A]{l} p'$ y queremos ver si es simulable desde q. Para esto, como los autómatas no comparten lenguaje, necesitamos conseguir un análogo a l en el autómata correspondiente a q. Supongamos que M_1 es el autómata que contiene a p y M_2 el que contiene a q, nuestra estrategia para realizar el *matching* de nombres será la siguiente:

1. Sabemos que l se compone de emisor, receptor y un mensaje (que a su vez tiene una etiqueta y un payload)
2. Si el emisor ya tiene un *match* realizado previamente, devolvemos ese. Sino, tomamos como candidatos a todos los participantes **no emparejados** de M_2, asignamos uno y lo devolvemos.
3. Si el receptor ya tiene un *match* realizado previamente, devolvemos ese. Sino, tomamos como candidatos a todos los participantes **no emparejados** de M_2, asignamos uno y lo devolvemos.
4. Si el mensaje ya tiene un *match* realizado previamente, devolvemos ese. Sino, tomamos las acciones en M_2 cuyos mensajes **no hayan sido emparejados**, y nos quedamos con aquellas cuyo emisor y receptor sean los emparejados. Asignamos uno de estos y lo devolvemos.
5. Todas las decisiones de emparejamiento tomadas son almacenadas, junto con los candidatos no tomados.
6. Devolvemos una nueva acción con las componentes emparejadas.

La idea es que las decisiones tomadas conforman un árbol de *backtracking*, por lo tanto si al finalizar la ejecución no conseguimos construir una relación de bisimulación, volvemos los pasos sobre la última decisión, tomamos al siguiente candidato, y reintentamos. Si no hay más decisiones que tomar y aún así no encontramos una relación, es que esta no existía (y los autómatas evaluados no son bisimilares).

```
class Bisimulation:

    def init(M_1 = ⟨Q_{M_1}, q_{0_{M_1}}, L_act × A, →_{M_1}⟩, M_2 = ⟨Q_{M_2}, q_{0_{M_2}}, L_act × A, →_{M_2}⟩):
        symmetricMode = SymmetricMode(False)
        matcher = ActionMatcher(M_1, M_2, symmetricMode)

    def execute():
        ∼ = calcularBisimulacion()
        while ¬esUnaRelacionValida(∼) ∧ matcher.quedanDecisionesPorTomar?:
                # Backtracking hasta que no haya mas decisiones para tomar.
                matcher.tomarSiguienteDecision()
                ∼ = calcularBisimulacion()

        # Si no se pudo, devuelve una relaci n y un match vac o
        if ¬esUnaRelacionValida(∼):
            return ∅, matcher.emptyMatch()

        return ∼, matcher.matches()

    def calcularBisimulacion():
        n = 0
        ∼_n= optimizar((Q_1 × P(M_1.assertions)) × (Q_2 × P(M_2.assertions)))
        try:
            do:
                ∼_{n+1}= ∅
                for ((p, K_p), (q, K_q)) ∈∼_n:
                    symmetricMode.disable()
                    if esSimulacion((p, K_p), (q, K_q), matcher):
                        symmetricMode.enable()
                        if esSimulacion((q, K_q), (p, K_p), matcher):
                            agregar ((p, K_p), (q, K_q)) en ∼_{n+1}
                n++
            while ∼_n≠∼_{n−1}
        except:
            # Si algo no matcheo, invalida el resultado actual para volver a intentar
            ∼_n= ∅

        return ∼_n

    def esSimulacion((p, K_p), (q, K_q)):
        for p --action/A--> p':
            matchedAction = matcher.match(action)
            caeEnLaRelacionActual?, simulaElConocimiento? = False

            for T ∈ P({ q --matchedAction/B--> q' }) ∧ ¬(simulaElConocimiento? ∨ caeEnLaRelacionActual?):
                K'_p = transcribirVariables(K_p \ action)
                A' = transcribirVariables(A)
                K'_q = K_q \ matchedAction

                simulaElConocimiento? = K'_p ∧ K'_q ∧ A' ⟹ K'_q ∧ ⋁_{q --matchedAction/B--> q'∈T} B

                candidato = ((p', K_p \ action), (q', K_q \ matchedAction))
                if symmetricMode.enable?:
                    candidato = ((q', K_q \ matchedAction), (p', K_p \ action))

                 caeEnLaRelacionActual? = candidato ∈∼_n
```

```
57
58              # Si no consigui  simular la transici n actual, no es una simulaci n
59              if ¬simulaElConocimiento ∨ ¬caeEnLaRelacionActual:
60                  return False
61
62          # Si sale del ciclo sin retornar False previamente, quiere decir que logro
                simular todas las transiciones desde p
63          return True
```

```
1  class Matcher:
2
3      def init(M1, M2, symmetricMode):
4          particintesCandidatos = SymmetricList(M2.participantes, M1.participantes,
               symmetricMode)
5          accionesCandidatas = SymmetricList(M2.acciones, M1.acciones, symmetricMode)
6          match = SymmetricDict(symmetricMode)
7          decisiones = []
8
9      def match(emisor, receptor, mensaje⟨payload⟩):
10         matchedEmisor = matchParticipante(emisor)
11         matchedReceptor = matchParticipante(receptor)
12         matchedMensaje = matchMensaje(mensaje, matchedEmisor, matchedReceptor)
13
14         return Accion(matchedEmisor, matchedReceptor, matchedMensaje)
15
16     def matchParticipante(participante):
17         if ¬match.definido?(participante):
18             if participantesCandidatos.hayCandidatosPara?(participante):
19                 candidato = participantesCandidatos.sacarUno()
20                 match.definir(participante, candidato)
21
22                 decisiones.push( ⟨participante, participantesCandidatos, symmetricMode⟩ )
23             else:
24                 raise(Error('No quedan candidatos'))
25
26         return match[participante]
27
28     def matchMensaje(mensaje, matchedEmisor, matchedReceptor):
29         if ¬match.definido?(mensaje):
30             # Son validas aqueyas cuyo emisor y receptor coinciden con los
                   matcheados, y comparte aridad y tipo de las variables en el payload
31             accionesValidas = {
32                 accion.mensaje | accion ∈ accionesCandidatas ∧
33                 accion.emisor == matchedEmisor ∧ accion.receptor == matchedReceptor ∧
34                 accion.mensaje.tienePayloadCompatibleCon(mensaje)
35             }
36             if ¬accionesValidas.empty?():
37                 candidato = accionesValidas.sacarUno()
38                 accionesCandidatas = accionesValidas \ {candidato}
39                 match.definir(mensaje, candidato.mensaje)   # matchea tanto el
                       nombre de mensaje como los nombres de variables
40
41                 decisiones.push( ⟨mensaje, accionesCandidatas.mensajes, symmetricMode⟩ )
42             else:
43                 raise(Error('No quedan candidatos'))
44
45         return match[mensaje]
46
```

```
47      def tomarSiguienteDecision():
48          if decisiones.size() > 0:
49              ultimaDecision = decisiones.pop()
50              if ultimaDecision.quedanCandidatos?():
51                  ultimaDecision.borrarMatchYDevolverCandidatoUsado()
52                  ultimaDecision.siguienteCandidato():
53                  decisiones.push(ultimaDecision)
54              else:
55                  tomarSiguienteDecision()
```

4.2.3. Elementos simétricos

Recordemos que, para que un elemento $((p, K_p), (q, K_q)) \in \sim_n$ sea incluido en $\sim_{n+1}$ necesitamos verificar si: 1) (p, K_p) es simulable por (q, K_q), y si 2) (q, K_q) es simulable por (p, K_p). Sin embargo en la relación $\sim_n$ mantenemos elementos pertenecientes al conjunto $(Q_1 \times M_1.assertions) \times (Q_2 \times M_2.assertions)$ por lo que, en principio, solo sería posible verificar *(1)*. Sumado a esto, el *matcher* necesita determinar de alguna manera que candidatos debe tomar en cada caso y de que forma debe guardar los nombres que decide *matchear*. Si $p \in \mathcal{Q}_{M_1}$ esta siendo simulado, entonces debe tomar candidatos de M_2, análogamente debe tomar de M_1 si $p \in \mathcal{Q}_{M_2}$.

Con la finalidad de determinar hacia que lado de la bisimulación estamos "mirando", construimos unas colecciones de datos a las que denominamos *Symmetric*. Estas se componen de dos colecciones internas: la actual y la simétrica; y un objeto al que denominamos *symmetricMode*, encargado de conservar el estado de simulación actual (hacia donde estamos "mirando") y es generado desde la clase *Bisimulación*, encargándose de intercambiar los modos según si M_1 simula a M_2 o viceversa (mediante los métodos *disable* y *enable*). De este tipo de colecciones tenemos dos:

1. *SymmetricList*: usada para guardar los candidatos a participantes y acciones (para los mensajes). Como inicialmente el *symmetricMode* está deshabilitado (en *False*), entonces quien está simulando es M_2, y son sus candidatos los correspondientes a la colección actual de la lista simétrica.

2. *SymmetricDict*: usada para guardar los *matches* de nombres.

Dado que una decisión puede ser tomada en cualquier momento de la bisimulación, y que al hacer *backtracking* necesitamos conocer ese momento (para saber a que colección devolver los candidatos, y de que manera borrar los *matches*), se guarda junto con esta al *symmetricMode* "actual".

4.2.4. Emparejar participante principal

Al comienzo de este trabajo mencionamos que trabajamos en un contexto en donde las *a-CFSM* son obtenidas mediante proyecciones de un *ac-automata* (Def. 11). Lo que vamos a hacer entonces es: 1) determinar al participante sobre el que está proyectada la *a-CFSM* al momento de construir el modelo y, 2) emparejar a los participantes principales de ambos autómatas, al momento de comenzar el cálculo de la bisimulación.

4.2.5. Emparejar conocimiento

Para poder determinar si un conocimiento K_q puede simular a otro K_p, primero necesitamos transcribir este último en términos del primero, lo que significa llevar los nombres de sus variables en

términos del autómata que está simulando. El problema con esto es que no hay un orden preestablecido para comprobar si los elementos de la relación $\sim_n$ están en $\sim_{n+1}$, además de que $\sim_0$ puede contener elementos $((p', K'_p), (q', K'_q))$ tal que K'_p (o K'_q) se compone de assertions de cualquier parte del autómata, inclusive predicando sobre variables que quizás nunca son accesibles desde p' (o q').

Para solucionar este problema, planteamos en la herramienta la siguiente solución mediante el método *transcribirVariables*:

1. Nos quedamos con el conjunto de variables de (K_q).
2. Filtramos por las que no estén en el *matcher*.
3. Buscamos en el autómata que esté simulando actualmente (si *symmetricMode* está deshabilitado es M_1, sino es M_2) las transiciones que definan estas variables.
4. Emparejamos las acciones de esas transiciones, y en consecuencia los mensajes que definen las variables y las propias variables.
5. Buscamos los emparejamientos de las variables de K_q y los usamos para renombrarlas.

4.2.6. Backtracking

El algoritmo construye un *matching* de nombres a medida que va necesitando emparejar participantes y mensajes para poder avanzar sobre la construcción de una relación de bisimulación, y existen dos situaciones en donde vamos a decidir hacer *backtracking* sobre las decisiones de emparejamiento tomadas para el *matching*.

La primera ocurre cuando la relación que se está construyendo encuentra un punto fijo, es decir si $\sim_n = \sim_{n-1}$, entonces no se sigue iterando. Esta relación construida va a ser válida, si y solo si, contiene a los elementos iniciales de ambos autómatas, es decir $((p_0, \emptyset), (q_0, \emptyset))$, de no ser así se procede a: 1) deshacer la última decisión de emparejamiento tomada, 2) tomar la siguiente posible, y 3) volver a intentar construir una relación a partir de este nuevo emparejamiento.

La segunda ocurre al intentar realizar un emparejamiento para un participante o mensaje, y no quedan más candidatos disponibles. En este caso lo que se hace es: 1) elevar una excepción desde el *matcher* avisando que hubo un error en los emparejamientos, 2) capturarla desde el algoritmo de bisimulación y 3) proceder de la misma forma que en el caso anterior (volviendo los pasos sobre la última decisión tomada).

En ambos casos puede ocurrir que no existan más decisiones a tomar, es decir que se haya recorrido todo el árbol de *backtracking*, lo que significa que no existe una relación de bisimulación posible entre los autómatas evaluados, para ninguno de los *matching* de nombres posibles. El resultado para esto es devolver una relación vacía, y un *matching* de nombres vacío.

4.2.7. Costo computacional

Hagamos un breve análisis del costo computacional del algoritmo, para esto fijemos una *a-CFSM* $M = \langle \mathcal{Q}, q_0, \mathcal{L}_{act} \times \mathcal{A}, \rightarrow \rangle$ y separemos el costo en dos partes.

La primer parte es el cálculo de la bisimulación. Notemos que se verifica la existencia de una simulación para cada uno de los elementos en la relación inicial, lo que nos deja una cantidad de operaciones proporcional al tamaño de la misma. Dijimos que la relación inicial se calcula como el producto entre los candidatos de los autómatas a evaluar, y cada conjunto de candidatos se calcula como el producto entre todos los estados de un autómata, por todas las combinaciones de assertions que haya en este. En el peor de los casos existe una assertion por transición, lo que significa que si $|\mathcal{Q}|$

es la cantidad de estados y $|\to|!$ es la cantidad de combinaciones de todas las assertions (en el peor caso), entonces la cantidad de candidatos para M está en el orden de: $\mathcal{O}(|Q| \times |\to|!)$. Si asumimos que M es el autómata con más candidatos de los dos evaluados, entonces la cantidad de elementos en la relación inicial está en el orden de: $\mathcal{O}((|Q| \times |\to|!)^2)$

La segunda parte es el costo del matching. Dado que utilizamos un algoritmo basado en *backtracking*, sabemos que el peor caso es en el que tenemos que recorrer todo el árbol de posibilidades para conseguir un *match* (o no conseguirlo, en caso de que no exista), y recorrer todo el árbol significa que se probaron todas las combinaciones de emparejamiento posibles. La cantidad de combinaciones en el peor caso se da cuando todas las transiciones tienen un mensaje distinto, lo que implica realizar un emparejamiento para cada una. Si además todas son compatibles entre si (comparten aridad y tipos de datos) entonces el árbol se compone de todas las combinaciones de mensajes posibles, y se encuentra en el orden de: $\mathcal{O}(|\to|!)$.

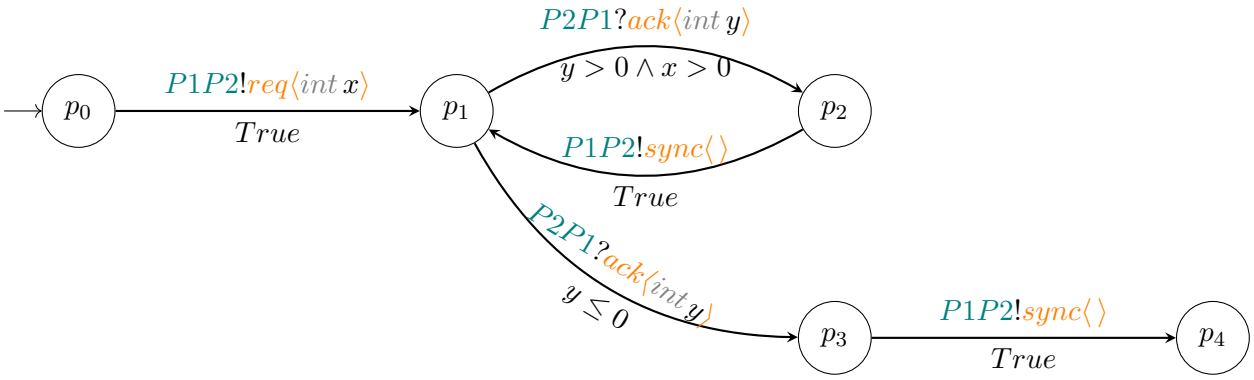
Finalmente, el costo computacional de todo el algoritmo (en un peor caso) se encuentra en el orden de: $\mathcal{O}((|Q| \times |\to|!)^2 \times |\to|!)$.

Es cierto también que este costo asume peores casos quizás irreales, dado que no parece probable que exista una *a-CFSM* en donde todos los mensajes tengan misma aridad y tipo de variables por ejemplo, o en donde todas las transiciones sean decoradas con assertions. Sin embargo, si es cierto que la manera en la que realizamos el cálculo inicial escala demasiado rápido hacia una cantidad de elementos para nada práctica, dejando imposibilitada una ejecución real.

4.3. Optimizaciones sobre el cálculo de la relación inicial

Si bien mencionamos que este trabajo no se centra en la optimización, al momento de ejecutar nos dimos cuenta que el algoritmo resulta impracticable en cuanto los autómatas comenzaban a crecer en tamaño, dada la cantidad de elementos en la relación inicial (que vimos, está directamente relacionada con el costo computacional). Teniendo esto en cuenta, vamos a repasar tres estrategias utilizadas con el fin de reducir la cantidad de elementos a evaluar, a partir de la relación inicial (reduciendo los candidatos de cada uno de los autómatas a evaluar). Estas estrategias son aplicadas en el orden presentado.

Ejemplo 13.



Es claro que, por ejemplo, el único camino hacia p_0 es el mismo estado (un camino sin transiciones). Por lo tanto, el único conocimiento asociable válido es un conjunto vacío. Sin embargo, el algoritmo

descrito construye un candidato por cada combinación de assertions posible:

$$\{ (p_0, \emptyset), (p_0, \{ y > 0, x > 0 \}), (p_0, \{ y \leq 0 \}), (p_0, \{ y > 0, x \leq 0 \}) \}$$

Esta misma combinación de assertions es aplicable a p_2, estado al que nunca se podría haber llegado mediante el cumplimiento de la restricción $y \leq 0$. Esta cantidad de combinaciones (que desde un principio observamos como innecesarias) es la que termina en una relación inicial con una cantidad de elementos demasiado grande como para que el algoritmo sea computacionalmente viable.

Partiendo del Ej. 13 podemos avanzar sobre la siguiente premisa: una assertion solo puede ser conocida por los estados alcanzables desde la transición decorada por dicha assertion. Y, a partir de esta premisa, surge la siguiente (primer) estrategia:

1. Sea $t = q_i \xrightarrow[A]{l} q_j$ una transición y $\mathcal{Q}$ los estados del mismo autómata.

2. Sea $\mathcal{Q}' \subseteq \mathcal{Q}$ los estados alcanzables desde q_j.

3. Dado que existe un camino entre q_j y todo estado $q' \in \mathcal{Q}'$, la assertion A pude ser una restricción a cumplir a partir de q', entonces la dejamos como posible conocimiento.

Para el cálculo de los estados alcanzables utilizamos un algoritmo *DFS*, si este retorna algún camino, entonces es alcanzable.

De esta manera, en el ejemplo anterior se reducen los candidatos del estado inicial p_0 a solo uno: $\{ (p_0, \emptyset) \}$.

Otro detalle que por ahora no hemos mencionado, está en las transiciones que redefinen una variable. Sigamos con el Ej. 13 y supongamos el siguiente camino: $\mathcal{P} = p_0 \to p_1 \to p_2 \to p_1 \to p_3$, y su conocimiento asociado $\mathcal{K}(\mathcal{P}) = x > 0 \wedge y \leq 0$. Dado que la transición desde p_1 a p_3 redefine la variable y, la restricción $y > 0$ es removida. Ahora bien si por ejemplo quisiéramos verificar que el autómata ejemplificado es bisimilar a si mismo, el algoritmo va a necesitar verificar que el candidato $(p_3, \{ x > 0, y \leq 0 \})$ pertenezca a la relación $\sim_n$, por lo que necesitamos que la redefinición mencionada se vea reflejada en la relación inicial.

El único caso que resulta problemático, son las assertions cuya expresión es una conjunción, dado que necesitamos remover solo la parte correspondiente a la variable redefinida (en el ejemplo, de $x > 0 \wedge y > 0$ solo queremos remover $y > 0$ cuando redefinimos y). Partiendo del resultado obtenido por la estrategia descrita anteriormente, optamos por la siguiente solución como segunda estrategia:

1. Identificamos las transiciones $t = q \xrightarrow[A]{l} q'$ en donde se redefinen variables sobre alguna assertion B, candidata de q (el estado *fuente* de t), tal que B es una conjunción.

2. Agregamos el resultado de $B \setminus l$ como assertion candidata a todos los estados alcanzables desde q'.

La tercera estrategia consta de descartar conocimientos que reconocemos como inválidos. Un conocimiento resulta inválido cuando este no es *satisfacible*. En el Ej. 13, las estrategias anteriores pueden dejarnos con un conocimiento en p_3 de la forma: $K = \{ x > 0, y > 0, y \leq 0 \}$, en donde la conjunción de las restricciones en K es trivialmente no satisfacible. Esta estrategia descarta este tipo de conocimiento.

4.4. Pruebas de rendimiento

En esta sección realizaremos un conjunto de pruebas de rendimiento **preliminares** sobre el algoritmo que brinda la herramienta, con el objetivo de determinar alguna noción de despeño a medida que crecen los autómatas a evaluar. La idea es realizar pruebas que fuercen el uso de la estructura de *bactracking* y medir cuanto tiempo demora el algoritmo en terminar, en casos en donde los autómatas evaluados son bisimilares.

A partir del análisis del costo computacional del algoritmo hecho anteriormente, sabemos que necesitamos centrarnos en medir los tiempos de ejecución en función de la cantidad de elementos en la relación inicial. Esta relación crece, a su vez, en función del crecimiento en la cantidad de estados y la cantidad de transiciones en el autómata. Vamos a definir entonces una medida de *tamaño* para los autómatas a evaluar, calculada como el producto de dos números: la cantidad de estados, multiplicados por la cantidad de transiciones.

Si bien mencionamos que el peor caso respecto del costo computacional no parece presentarse en casos reales, se espera que los tiempos de ejecución acompañen la forma de una función factorial. Además, dado que apoyamos la complejidad del algoritmo en la cantidad de elementos, esperamos que para las distintas estrategias los tiempos de ejecución sean equivalentes (esperamos no observar una distancia significativa entre los tiempos, para autómatas de mismo tamaño).

La idea para estas pruebas va a ser, a partir de un autómata base, incrementarlo en tamaño agregando estados y transiciones, utilizando distintas estrategias de incremento. Más específicamente, la estructura de las pruebas va a ser la siguiente:

1. Se fabrica una *a-CFSM* de un *tamaño* inicial.
2. Se fabrican otras 10 *a-CFSM*, del mismo *tamaño* y bisimilares a la primera.
3. Se mide el tiempo del algoritmo al calcular una bisimulación entre la primera y cada una de las otras 10, y se toma el promedio.
4. Se hace crecer al primer autómata y se repite el proceso hasta llegar a uno de tamaño n.

Vamos a realizar un conjunto de 3 pruebas, variando la manera en la que hacemos crecer al autómata en el punto (4) de la siguiente forma:

i. Se eligen aleatoriamente 2 estados, y a cada uno se le agrega una transición hacia un nuevo estado. Estas transiciones tienen mensajes con entre 1 y 3 parámetros, y una restricción para cada uno, siendo la assertion correspondiente una conjunción de estas restricciones.

ii. Se elige aleatoriamente un estado y se agregan 2 transiciones hacia un mismo estado. Estas transiciones tienen un mismo mensaje, con entre 1 y 3 parámetros, más un entero sobre el que se impone una restricción. Cada transición es decorada con una assertion con dicha restricción, siendo estas siempre disjuntas.

iii. Se procede igual al punto (*ii*), pero las transiciones van hacia estados distintos.

Además para la construcción de una nueva acción se selecciona un participante de manera aleatoria, vinculándolo mediante una comunicación con el participante principal.

Para el punto (2), dado que el proceso de *matching* de nombres siempre asume que los autómatas a evaluar nunca comparten nombres, lo que vamos a hacer es clonar al autómata generado en el punto

(1) y mezclar las transiciones desde cada estado de manera aleatoria, de forma que el algoritmo se vea obligado a realizar *backtracking*. El hecho de utilizar un *tamaño* límite (en el punto (1)) en lugar de una cantidad de iteraciones se debe a que la proporción de incremento varía entre las estrategias, de querer usar una misma proporción la cantidad de transiciones y estados a agregar por iteración debería ser mayor, lo que nos deja una menor granuralidad entre cada etapa.

Con el fin de que el *z3-solver* no influya el costo computacional, vamos a limitarnos a mensajes cuyos parámetros sean del tipo *enteros*, *booleanos* y *strings*, predicando siempre por igualdad o desigualdad en las restricciones. Las pruebas se realizan partiendo de un autómata de *tamaño* 36, y para la selección del n (en el punto (1)) realizamos una pre-selección en donde medimos los tiempos de ejecución, para cada estrategia, buscando que estos no superen (aproximadamente) una hora. El tamaño límite queda configurado en $n = 1200$.

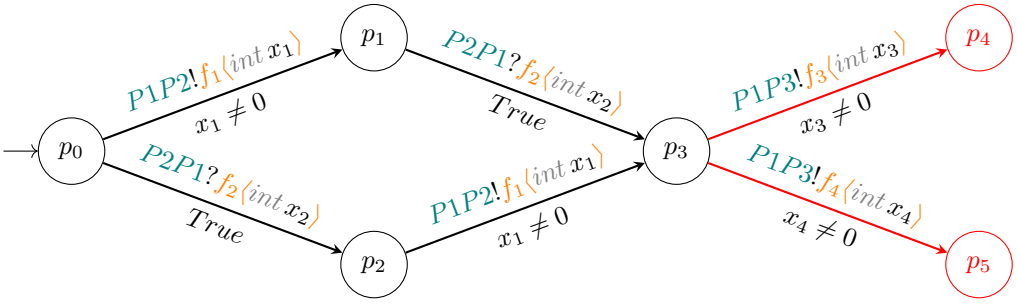


Fig. 4.1: Ejemplo de incremento con la estrategia (i)

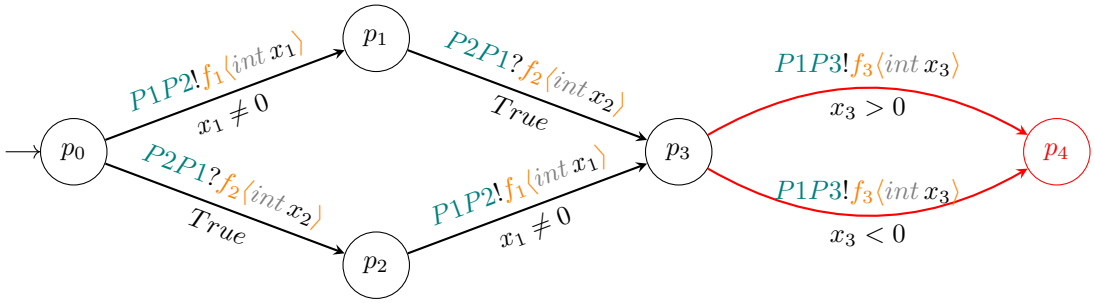


Fig. 4.2: Ejemplo de incremento con la estrategia (ii)

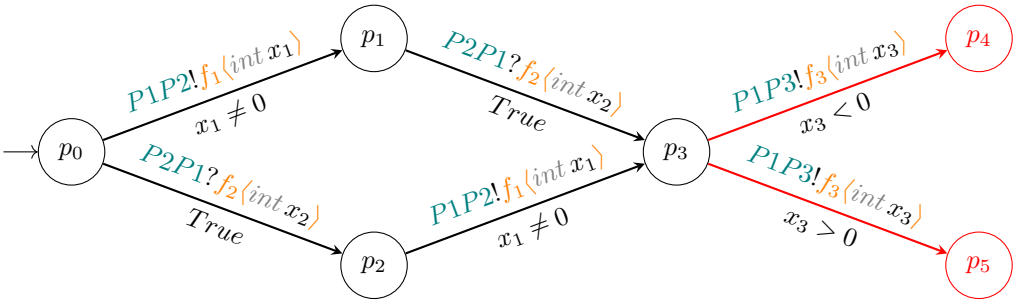


Fig. 4.3: Ejemplo de incremento con la estrategia (iii)

El siguiente es un gráfico en representación de los resultados obtenidos.

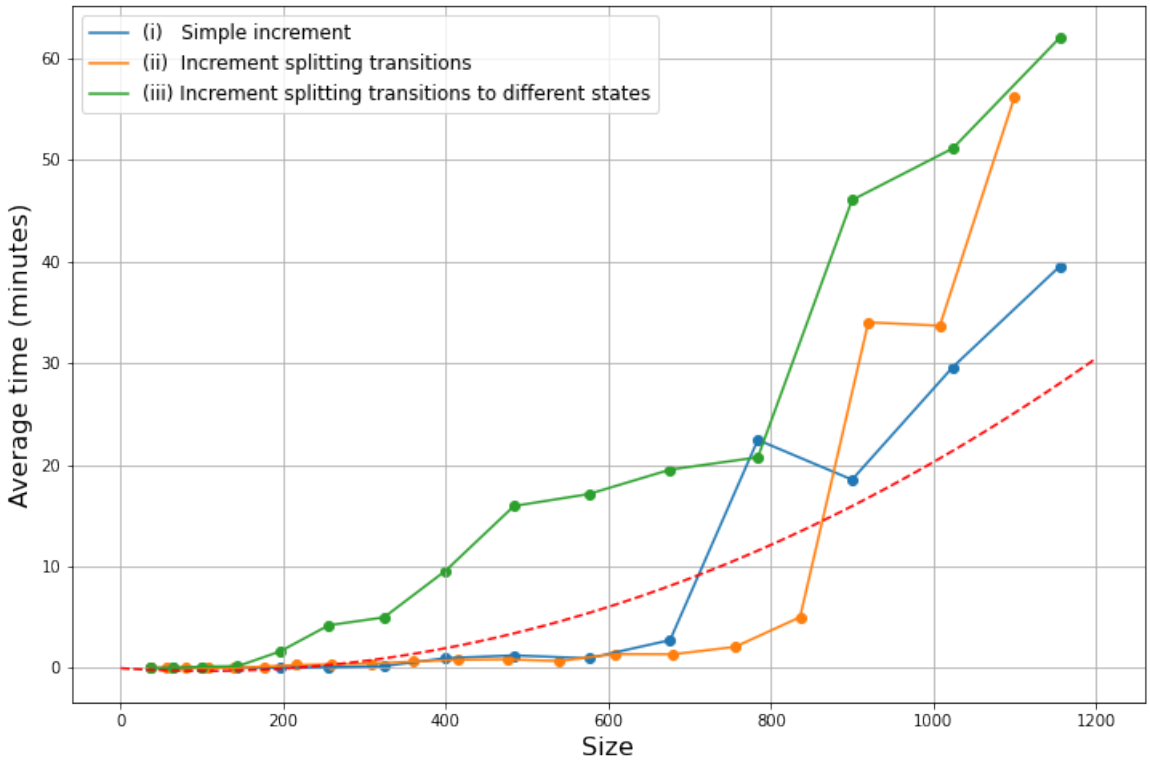


Fig. 4.4: Promedio de tiempo del cálculo de bisimulación por *tamaño* del autómata

Lo primero que podríamos destacar es que tanto la estrategia (i) como la (ii) parecen seguir una tendencia similar en autómatas con *tamaño* menor a $n = 600$, en donde los tiempos de ejecución son de aproximadamente 1 minuto. A pesar de que la estrategia (iii) se separa en estos primeros casos, a partir de un tamaño de $n = 800$ parece cumplirse nuestra suposición de una similitud entre los tiempos de ejecución de las tres estrategias (tendiendo hacia una misma función de crecimiento).

Ahora bien, no podemos decir lo mismo acerca de la hipótesis en donde se esperaban tiempos de ejecución con crecimiento factorial, ya que el gráfico muestra quedarse muy lejos una función como

esta. Mas aún, la linea roja y punteada mostrada en el gráfico representa una función cuadrática de la forma $y = ax^2 + bx$ en donde los coeficientes fueron ajustados a los puntos mediante la función *curve_fit* del paquete *scipy.optimize* de *python*. Por ende, podríamos decir que las funciones de tiempo por *tamaño* para las tres estrategias se asemejan a un polinomio de grado 2.

Es posible que los casos estudiados no sean suficientes para determinar la naturaleza de la función de tiempos de los ejecución, o bien que la suposición inicial en donde la complejidad algorítmica esta en el orden factorial esté siendo demasiado inocente en el cálculo (exhibiendo en la práctica un comportamiento más eficiente). Para poder tener mejores resultados, es necesario profundizar en un análisis de complejidad del algoritmo de bisimulación, así como del cálculo de la función inicial (y seguramente del algoritmo de *matching* de nombres).

5. CONCLUSIONES Y TRABAJO A FUTURO

5.1. Conclusiones

El objetivo en esta tesis era el de abordar la problemática de definir una noción de bisimulación para las *a-CFSM*, necesaria para poder trabajar en una arquitectura que permita realizar una reconfiguración dinámica transparente de artefactos de software (Cap. 1), a partir de utilizar estos autómatas como descriptores de protocolos locales. Para realizar esta tarea definimos un tipo de autómata más general, al que denominamos *a-FSM*, lo que nos permitió definir una noción de bisimulación concentrándonos exclusivamente en un intercambio de mensajes con restricciones, independientemente de los participantes. A partir de este resultado pudimos extenderlo a *a-CFSM* mediante la Def. 20, permitiéndonos incorporar a la solución un *matching* de nombres entre los autómatas a evaluar. La solución encontrada, además, resulta ser flexible en términos de los autómatas que encuentra bisimiliares, dado que considera bisimilaridad sobre pares de estados en donde el cómputo de un mensaje resulta equivalente, pero no necesariamente igual (nos referimos al hecho destacado en la Def. 19, en donde pares de estados pueden tener transiciones hacia una cantidad de estados distintos, siempre y cuando los valores que cumplen con las restricciones sean los mismos).

Todo esto desarrollado en un contexto en el que las *a-CFSM* son obtenidas a partir de la proyección de un participante, desde una descripción global de un protocolo de comunicación dada por un *ac-autómata*. Partir de este contexto nos proporciona ciertas garantías, por ejemplo si este descriptor global cumple la propiedad de *consistencia* [GLS+22] (Def. 4.13) sabemos que su proyección cumple con la propiedad de *sensibilidad a la historia* (Def. 14), necesaria para la definición de bisimulación presentada (Sec. 3.1).

Con el fin de poner en práctica los resultados obtenidos, desarrollamos una herramienta compuesta de: un algoritmo de bisimulación (utilizando una técnica de estratificación) y una algoritmo de *matching* de nombres (utilizando una estrategia de *backtracking*). Aplicamos (y detallamos) un conjunto de optimizaciones necesarias para el funcionamiento de la herramienta y el algoritmo de bisimulación elegido, debido al alto costo computacional original. Por último, exploramos preliminarmente los límites la implementación presentada, estudiando los tiempos de demora a medida que crecen los autómatas a evaluar.

El conjunto de estos resultados proporcionan una solución tanto teórica como práctica del problema descrito originalmente; siendo quizás la parte práctica una solución preliminar, pero pudiendo servir como punto de partida para futuros trabajos.

5.2. Trabajo a futuro

En esta sección vamos a discutir algunas líneas de investigación a futuro, en las cuáles encontramos cuestiones netamente prácticas del algoritmo, y por otro lado tenemos temas teóricos asociados a la elección del *compliance* para el vínculo entre los contratos de *requerimiento* y *provisión*.

Como mencionamos anteriormente, uno de los objetivos de este trabajo era el de desarrollar una herramienta que pusiera en práctica definición de bisimulación presentada, considerando el *matching* de nombres. El algoritmo (como parte de la herramienta) encargado de la construcción de la bisimulación comienza definiendo una relación inicial mas bien *naive*, por lo que se optó por optimizar ante la

imposibilidad de ponerlo en practica. Sin embargo y a pesar de estas mejoras de rendimiento realizadas, el algoritmo sigue teniendo un costo computacional muy elevado para poder ser puesto en marcha sobre autómatas reales, en situaciones como la explicada en la introducción de este trabajo.

Si uno quisiera mantener el uso de este algoritmo y mejorar su rendimiento, un buen punto de partida podría ser el de realizar un *benchmarking*, tanto para encontrar focos de complejidad como para detectar aspectos a ser pre-computados. Por ejemplo, durante el cálculo de la bisimulación no disponemos de un orden de ejecución para los elementos en la relación y entonces, al momento de querer emparejar una assertion para la que todavía no emparejamos sus variables, esto resulta en tener que buscar las transiciones en las que se definen esas variables, y emparejarlas. Si supiésemos previamente cuáles son los conocimientos reales disponibles en cada uno de los estados, podríamos ordenar los elementos en la relación inicial de manera que **siempre** que se verifique una simulación, todas las assertions prediquen en términos de variables ya emparejadas. El cálculo de estos conocimientos podría ser costoso, ya que habría que conseguir todos los caminos que alcanzan a cada uno de los estados (inclusive teniendo en cuenta los ciclos), desde el estado inicial. Sin embargo en la práctica estos autómatas resultarían ser estáticos (no se modifican en el tiempo), con lo cual los conocimientos para cada estado, y por lo tanto los candidatos para construir una relación inicial, podrían ser pre-computados al momento de poner en funcionamiento el servicio que cumple con el protocolo descrito mediante la *a-CFSM* (y tenerlo ya disponible al momento de calcular la bisimulación).

Por otro lado, un algoritmo basado en estratificación podría considerarse poco óptimo en si mismo, por lo que podríamos optar por definir algún conjunto de algoritmos candidatos a partir de los existentes en la literatura. Mediante este conjunto, lo que se puede hacer es estudiar el funcionamiento de cada uno, intentar adaptarlo a la noción de bisimulación descrita en la Def. 20, implementarlo haciendo uso (quizás) del *Matcher* provisto por la herramienta descrita en este trabajo, y finalmente realizar un *benchmark* para determinar al que "mejor" cumple la tarea a realizar (tener una *configuración dinámica transparente* Cap. 1).

5.2.1. Reconsiderando la noción de satisfacibilidad entre contratos

Si nos remitimos al paper original [IVT16], parecería razonable poner en discusión el hecho de utilizar una bisimulación como *compliance* para determinar si uno contrato de *provisión* satisface a no de *requerimiento*.

Tomemos un contrato de requerimiento R y un contrato de provisión PR. En principio, la lógica indicaría que una simulación es suficiente para determinar si el contrato PR satisface R, ya que nos asegura que PR puede hacer **por lo menos** lo mismo que R. Parte del contexto de asumir que esto es correcto, pertenece al hecho de que los autómatas correspondientes a los contratos solo intercambian etiquetas, y que las ejecuciones son simplemente sucesiones de estas etiquetas. Entonces, a la hora de hablar de equivalencia, podríamos reducirlo a términos de equivalencia de ejecuciones. Ahora bien, cuando las etiquetas tienen una semántica esto deja de ser así. La semántica de las etiquetas en las *a-CFSMs* corresponde al envió y recepción de mensajes, lo que determina roles en la comunicación. Estos roles pueden ser: *activo*, cuando un participante envía un mensaje y *pasivo*, cuando un participante recibe un mensaje.

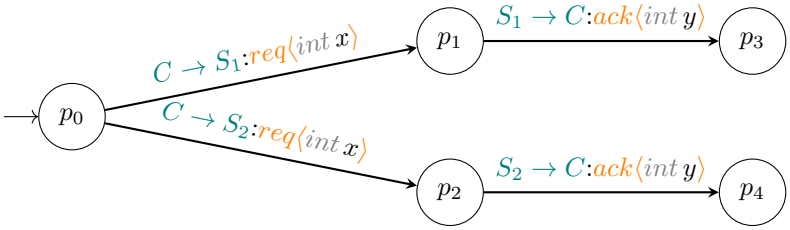
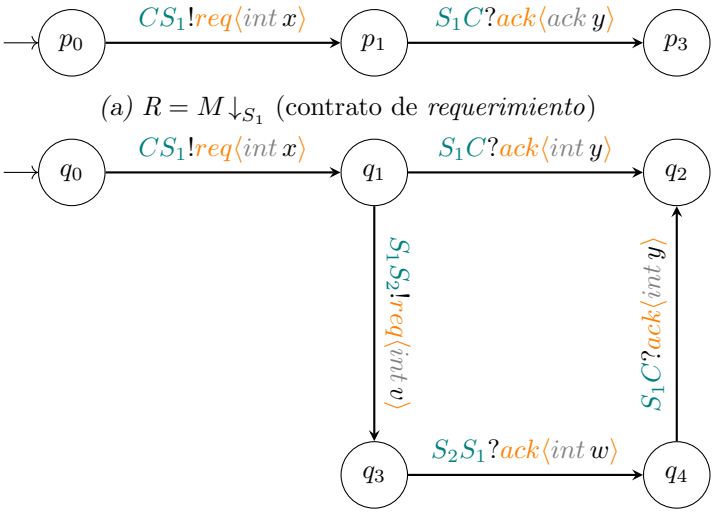


Fig. 5.1: ac-automata M (descriptor global)

Supongamos que el *compliance* se trata de una simulación entre PR (quien simula) y R (el simulado). Esto quiere decir que el participante p, correspondiente al servicio contratado, podría hacer (quizás) más cosas de las esperadas; es decir, podría tener más acciones que las detalladas en el descriptor global. Si alguna de estas acciones está ligada a un rol *pasivo* por parte de p, entonces este sabe como responder a mensajes que el rol activo no conoce, por lo que no existe un problema. Ahora bien, si alguna de estas acciones está ligada a un rol *activo* por parte de p, entonces nos encontramos con una acción que podría no ser aceptable por el comportamiento del canal, ya que el descriptor global no da garantías de que el participante correspondiente al rol *pasivo* pueda contestar apropiadamente.

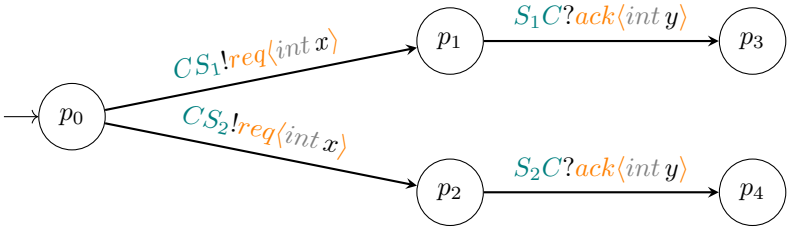


(a) $R = M{\downarrow_{S_1}}$ (contrato de *requerimiento*)

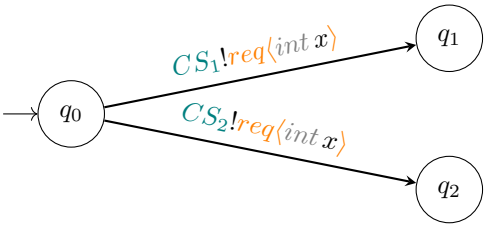
(b) PR (contrato de *provisión*)

Fig. 5.2: Ejemplo de simulación de R por parte de PR. El participante S_1 juega un rol *activo* en acciones sobre las que M no puede dar garantía, en este caso, que S_2 sepa responder apropiadamente.

Supongamos ahora un *compliance* inverso, una simulación entre R (quien simula) y PR (el simulado). El resultado es opuesto a lo descrito anteriormente, en donde el participante p, correspondiente al servicio contratado, sabe hacer **a lo sumo** todo lo esperado; esto quiere decir que puede tener menos acciones que las detalladas en el descriptor global. Si alguna de estas acciones está ligada a un rol *activo* por parte de p, entonces desconoce como enviar ciertos mensajes que el rol *pasivo* sabe contestar, por lo que no existe un problema. Si alguna de estas acciones está ligada a un rol *pasivo* por parte de p, ahora es este quien podría no saber responder apropiadamente a un mensaje.



(a) $R = M{\downarrow_C}$ (contrato de *requerimiento*)



(b) PR (contrato de *provisión*)

Fig. 5.3: Ejemplo de simulación de PR por parte de R. El participante C juega un rol *pasivo* en acciones sobre las que no puede garantizar saber responder apropiadamente.

Luego, surgen problemas en ambas direcciones, por lo que una simulación resulta no ser suficiente. Pero entonces, ¿Es posible hallar un *compliance* sin recurrir a una bisimulación? Si bien no tenemos una respuesta, una posibilidad es la de estudiar una noción de simulación que cambie la dirección dependiendo de si el rol es *activo* o *pasivo*. Sin embargo no es claro como se definiría esto y, si bien existen nociones de simulación en sistemas de I/O, el uso del término "simulación" podría considerarse incorrecto para este caso. Para evitar confusiones, sería preferible hablar en términos de una noción no direccional de *compliance*.

Bibliografía

[BHTY10] Laura Bocchi, Kohei Honda, Emilio Tuosto, and Nobuko Yoshida. A theory of design-by-contract for distributed multiparty interactions. In Paul Gastin and François Laroussinie, editors, *CONCUR 2010 - Concurrency Theory*, pages 162–176, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[BLT20] Franco Barbanera, Ivan Lanese, and Emilio Tuosto. Choreography automata. In Simon Bliudze and Laura Bocchi, editors, *Coordination Models and Languages*, pages 86–106, Cham, 2020. Springer International Publishing.

[BZ83] Daniel Brand and Pitro Zafiropulo. On communicating finite-state machines. *J. ACM*, 30(2):323–342, apr 1983.

[DY12] Pierre-Malo Deniélou and Nobuko Yoshida. Multiparty session types meet communicating automata. In Helmut Seidl, editor, *Programming Languages and Systems*, pages 194–213, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[GLS+22] Lorenzo Gheri, Ivan Lanese, Neil Sayers, Emilio Tuosto, and Nobuko Yoshida. Design-by-contract for flexible multiparty session protocols. In Karim Ali and Jan Vitek, editors, *36th European Conference on Object-Oriented Programming (ECOOP 2022)*, volume 222 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 8:1–8:28, Dagstuhl, Germany, 2022. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[HM85] Matthew Hennessy and Robin Milner. Algebraic laws for nondeterminism and concurrency. *J. ACM*, 32(1):137–161, jan 1985.

[IVT16] Carlos Gustavo Lopez Pombo Ignacio Vissani and Emilio Tuosto. Communicating machines as a dynamic binding mechanism of services. *Electronic Proceedings in Theoretical Computer Science*, 203:85–98, feb 2016.

[LTY15] Julien Lange, Emilio Tuosto, and Nobuko Yoshida. From communicating machines to graphical choreographies. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, page 221–232, New York, NY, USA, 2015. Association for Computing Machinery.

[Mil99] Robin Milner. *Communicating and Mobile Systems: The pi-Calculus*. Cambridge University Press, USA, 1999.

[MLa] Agustín Eloy Martinez Suñé and Carlos G. Lopez Pombo. Automatic quality-of-service evaluation in service-oriented computing. pages 221–236.

[MLb] Agustín Eloy Martinez Suñé and Carlos G. Lopez Pombo. Quality of service ranking by quantifying partial compliance of requirements. pages 181–189.

[ŢF13] IonuŢ ŢuŢu and José Luiz Fiadeiro. A logic-programming semantics of services. In Reiko Heckel and Stefan Milius, editors, *Algebra and Coalgebra in Computer Science*, pages 299–313, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[YZF21] Nobuko Yoshida, Fangyi Zhou, and Francisco Ferreira. Communicating finite state machines and an extensible toolchain for multiparty session types. In *Fundamentals of Computation Theory: 23rd International Symposium, FCT 2021, Athens, Greece, September 12–15, 2021, Proceedings*, page 18–35, Berlin, Heidelberg, 2021. Springer-Verlag.