Una semántica ejecutable en lógica de reescritura para programación espacial concurrente por restricciones (SCCP)

Miguel Ángel Romero González

Director
Dr. Camilo Rocha

Programa de Ingeniería de Sistemas
Escuela Colombiana de Ingeniería *Julio Garavito*
Bogotá, Colombia
Enero, 2017

Resumen

Este documento presenta una semántica ejecutable en lógica de reescritura para programación espacial concurrente por restricciones (del, inglés, *Spatial Concurrent Constraint Programming*, abreviado SCCP). SCCP es un modelo computacional utilizado para razonar acerca de información espacial y conocimiento distribuido entre los procesos de un sistema. En SCCP, los procesos simulan declaración de conocimiento común y consulta de información global, a su vez, cada agente en SCCP tiene su propio espacio de información, y entidades epistémicas tal como conocimiento o creencias son representadas en un sistema de restricciones paramétricas. La semántica ejecutable de SCCP se hace disponible a los usuarios y programadores a través de una herramienta que integra un lenguaje de programación y la especificación en Maude. Desde esta herramienta, además de simular programas en este lenguaje, también es posible verificar propiedades especificadas en lógica lineal temporal.

Agradecimientos

Quiero agradecer al Profesor Camilo Rocha por ser mi director de proyecto, por guiarme en este proceso y permitirme trabajar con él. Le agradezco su apoyo y tiempo, lo que me ha permitido aprender de su experiencia y conocimientos. Le agradezco todas las enseñanzas que me ayudaron en mi formación personal y profesional.

A mis padres quienes me han apoyado incondicionalmente para alcanzar mis metas. A las personas que hicieron parte de estos años en la Escuela como parte de un proyecto de vida.

Índice general

Capítulo 1

Introducción

Programación concurrente por restricciones (del inglés, *Concurrent Constraint Programming*, abreviado CCP) [**17, 18, 16**] es un modelo de concurrencia que combina el sistema operacional tradicional del cálculo de procesos con una idea declarativa basada en la lógica. La cuestión fundamental en CCP es la especificación de sistemas por medio de restricciones que representan información parcial sobre algunas variables. El modelo CCP es paramétrico en el sistema de restricciones especificando la estructura e interdependencias de la información parcial que los procesos pueden solicitar y publicar en un banco compartido. En el modelo CCP los procesos pueden ser considerados transmisores y el estado del sistema esta determinado por el banco de información (i.e. una restricción) que se refina monotónicamente por procesos que agregan nueva información.

La noción de sistema de restricciones se ha fortalecido recientemente con la inclusión de funciones espaciales al lenguaje de restricciones, dando como resultado un sistema espacial por restricciones (del inglés, *Spatial Constraint System*, abreviado SCS) [**8**]. Las funciones espaciales pueden considerarse como operadores topológicos y de clausura que permiten la especificación de información espacial y epistémica como parte de la estructura de restricciones. La programación espacial concurrente por restricciones (del inglés, *Spatial Concurrent Constraint Programming*, abreviado SCCP) es un modelo de concurrencia resultado de parametrizar CCP con un sistema espacial por restricciones [**8**]. El modelo SCCP es ideal para modelar y razonar acerca de agentes que consultan y comparten información en la presencia de jerarquías espaciales. Algunos ejemplos de gestión de acceso a información por jerarquias incluyen círculos de amigos y publicación de mensajes en redes sociales, carpetas compartidas en la nube, y compartimentación de la ejecución de procesos en el sistema operativo. De estos campos surgen cuestiones como lo es el diseño de políticas para predecir y prevenir problemas de privacidad, por ejemplo.

Este documento presenta una semántica en lógica de reescritura que agrupa la semántica operacional estructurada de SCCP [**8**]. La semántica en lógica de reescritura de SCCP es una teoría ejecutable en lógica de reescritura [**11**]. La especificación en lógica de reescritura puede ser ejecutada en Maude [**3**], un lenguaje e implementación de lógica de reescritura de alto desempeño. La semántica en lógica de reescritura de SCCP hace parte de un esfuerzo para brindar técnicas deductivas y algorítmicas para razonar acerca de las propiedades espaciales y epistémicas de los sistemas concurrentes. Específicamente, la semántica ejecutable en lógica de reescritura presentada en este documento puede ser utilizada para ser ejecutada y verificar algoritmicamente propiedades de alcanzabilidad de sistemas de agentes dentro de una estructura espacial jerárquica y con información localizada.

Finalmente, para poder poner los conceptos espaciales y epistémicos al alcance de los programadores, este documento introduce un prototipo de lenguaje de programación con una sintaxis simple que soporta sistemas SCCP, tal como consultas y modificación de bancos de información indexando por el espacio de un agente. Los programas escritos en este lenguaje pueden ser introducidos en una herramienta que se comunica con el sistema de Maude y puede ser ejecutada en la semántica en lógica de reescritura de SCCP.

El documento está organizado de la siguiente manera:

1. El Capitulo 2 incluye algunos conceptos requeridos para la lectura del documento.
2. El Capítulo 3 presenta la descripción del modelo SCCP y su semántica operacional.
3. El Capítulo 4 presenta la especificación formal donde se modelan los agentes, procesos y sus transiciones.
4. El Capítulo 5 presenta el lenguaje de programación propuesto basado en el modelo SCCP.
5. El Capítulo 6 presenta el ambiente desarrollado para el lenguaje de programación.
6. El Anexo A incluye los módulos de sistema y funcionales requeridos para la especificación del modelo SCCP.

Capítulo 2

Preliminares

2.1. Lógica de reescritura

Lógica de reescritura [**11**] es una lógica de cambios concurrentes. Las reglas de la lógica de reescritura son patrones generales para acciones básicas que pueden ocurrir concurrentemente con otras acciones en un sistema concurrente. Por lo tanto, la lógica de reescritura permite razonar sobre cambios complejos en un sistema, teniendo en cuenta que los cambios corresponden a las acciones básicas axiomatizadas por las reglas de reescritura.

2.1.1. Teoría ecuacional. Una *signatura* ordanada por tipos Σ es una tupla $\Sigma = (S, \leq, F)$ con un conjunto parcialmente ordenado de tipos $(S, \leq)$ y un conjunto de símbolos de función F. La relación binaria $\equiv_{\leq}$ denota la relación de equivalencia generada por $\leq$ sobre S y su extensión a cadenas en S^*.

La colección de variables X es una familia S-indexada $X = \{X_s\}_{s \in S}$ de conjuntos de variables disyuntos con cada X_s infinito contable. $T_\Sigma(X)_s$ es el *conjunto de términos de tipo s* y el *conjunto de términos simples de tipo s* se denota por $T_{\Sigma,s}$. Las expresiones $\mathcal{T}_\Sigma(X)$ y $\mathcal{T}_\Sigma$ denotan las correspondientes algebras de Σ-términos ordenadas por tipos.

Una Σ-*ecuación* es una pareja $t = u$ con $t \in T_\Sigma(X)_{s_t}$, $u \in T_\Sigma(X)_{s_u}$ y $s_t \equiv_{\leq} s_u$. Una Σ-*ecuación condicional* es una ecuación condicional $t = u$ **if** γ con $t = u$ una Σ-ecuación y γ una conjunción finita de Σ-ecuaciones. Un *tipo al tope* en Σ es un tipo $s \in S$ tal que si $s' \in S$ y $s \equiv_{\leq} s'$, entonces $s' \leq s$.

Una *teoría ecuacional* es un par (Σ, E), donde Σ es una signatura y E es una colección finita de ecuaciones, posiblemente condicionales. Una teoría ecuacional $\mathcal{E} = (\Sigma, E)$ induce la relación de congruencia $=_{\mathcal{E}}$ sobre $T_\Sigma(X)$ definida por $t =_{\mathcal{E}} u$, con $t, u \in T_\Sigma(X)$, sí y sólo sí $\mathcal{E} \vdash t = u$ por las reglas de deducción para lógica ecuacional ordenada por tipos en [**12**], sí y sólo sí, en [**12**] $t = u$ es válido en todos los modelos de $\mathcal{E}$.

Las expresiones $\mathcal{T}_{\Sigma/E}(X)$ y $\mathcal{T}_{\Sigma/E}$ corresponden a las algebras de cocientes inducidas por $=_{\mathcal{E}}$ sobre las algebras de términos $\mathcal{T}_{\Sigma}(X)$ y $\mathcal{T}_{\Sigma}$. El algebra $\mathcal{T}_{\Sigma/E}$ es llamado el *algebra inicial* de (Σ, E).

Las Σ-ecuaciones están divididas en un conjunto A de axiomas estructurales (tales como asociatividad, conmutatividad y/o identidad) y el conjunto E de ecuaciones.

2.1.2. Teoría de reescritura. Una *teoría de reescritura* es una tupla $\mathcal{R} = (\Sigma, E, R)$ con una teoría ecuacional $\mathcal{E}_{\mathcal{R}} = (\Sigma, E)$ y un conjunto finito de Σ-reglas R. Una *teoría de reescritura al tope* es una teoría de reescritura $\mathcal{R} = (\Sigma, E, R)$, tal que cada regla $t \to u$ **if** $\gamma \in R$ es tal que $l, r \in T_{\Sigma}(X)_s$ para algún $s = [s]$ en Σ, $l \notin X$, y no hay operadores en Σ que tengan al tipo s como argumento.

Una teoría de reescritura $\mathcal{R} = (\Sigma, E, R)$ induce la relación de reescritura $\to_{\mathcal{R}}$ sobre $T_{\Sigma}(X)$ definida por $t \to_{\mathcal{R}} u$, con $t, u \in T_{\Sigma}(X)$, sí y sólo sí una demostración de reescritura de un paso de $\mathcal{R} \vdash t \to u$ puede ser obtenida por las reglas de deducción para teorías de reescritura ordenadas por tipos en [**1**], sí sólo sí, en [**1**] $t \to u$ es válido en todos los modelos de $\mathcal{R}$. La expresión $\mathcal{T}_{\mathcal{R}} = (\mathcal{T}_{\Sigma/E}, \to_{\mathcal{R}}^{*})$ denota el modelo de alcanzabilidad inicial de $\mathcal{R} = (\Sigma, E, R)$ [**1**], donde $\to_{\mathcal{R}}^{*}$ representa la clausura transitiva-reflexiva de $\to_{\mathcal{R}}$.

2.2. Maude

Maude [**3**] es un lenguaje declarativo. Un programa en Maude es una teoría lógica y un cálculo en Maude es una deducción lógica que utiliza los axiomas especificados en la teoría o el programa, según corresponde, bajo el sistema deductivo de la lógica de reescritura [**11**].

Maude cuenta con dos tipos de módulos: funcional y de sistema.

2.2.1. Módulos funcionales. Los módulos funcionales corresponden a teorías ecuacionales y definen tipos de datos y operaciones sobre estos tipos de datos.

Los módulos funcionales de Maude suponen la propiedad de que la ecuaciones son consideradas reglas de simplificación que se usan únicamanete de izquierda a derecha y su repetida aplicación reduce un término a su forma canónica, la cual no depende del orden en el que se usen las ecuaciones. Este tipo de simplificación canónica es posible si la teoría ecuacional asociada a un módulo funcional es Church-Rosser [**6**] y terminante [**10**].

Los módulos funcionales pueden contener: ecuaciones con o sin atributos, pertenencias y operadores con sus respectivos atributos. Las ecuaciones y pertenencias pueden ser condicionales o incondicionales.

Un módulo funcional se define con la palabra clave `fmod`. Para más información sobre conceptos básicos y sintaxis de Maude para módulos funcionales se refiere al lector a [**3**].

2.2.2. Módulos de sistema. Un módulo de sistema especifica una teoría de reescritura. Una teoría de reescritura contiene tipos de datos, clases de equivalencia de tipos de datos, operadores, y ecuaciones, pertenencias y reglas de reescritura, las cuales pueden ser condicionales. De lo anterior se puede afirmar que toda teoría de reescritura tiene una teoría ecuacional subyacente. Un módulo de sistema se declara con la palabra clave `mod`. Para más información sobre conceptos básicos y sintaxis de Maude para módulos de sistema se refiere al lector a [**3**].

El conjunto de Σ-reglas R es coherente con respecto a las ecuaciones E módulo A, si Maude puede ejecutar un módulo de sistema admisible [**3**] usando la estrategia de primero simplificar un término t a su forma E/A-canónica y luego usar una regla con R módulo A para lograr el efecto de reescribir con R módulo $E \cup A$.

2.3. Retículo algebraico completo

Esta sección presenta la definición de un *Retículo Algebraico Completo*, para más información se refiere al lector a [**2**].

Una relación binaria $\leq$ definida en un conjunto A es un *orden parcial* del conjunto A si se mantienen las condiciones *C1*, *C2* y *C3*.

C1 Reflexividad. $a \leq a$
C2 Antisimetría. $a \leq b$ y $b \leq a$ implica $a = b$
C3 Transitividad. $a \leq b$ y $b \leq c$ implica $a \leq b$

Un conjunto no vacío con orden parcial es llamado un *conjunto parcialmente ordenado* o más brevemente un *poset* (del inglés, *partially ordered set*).

Un conjunto parcialmente ordenado L es un *retículo* si y solo si para cada a, b en L existen $\sup\{a, b\}$ e $\inf\{a, b\}$. La expresión $\sup\{a, b\}$ denota la *menor cota superior* o *supremo*, e $\inf\{a, b\}$ la *mayor cota inferior* o *ínfimo*.

Un conjunto parcialmente ordenado P es *completo* si para cada subconjunto A de P existen $\sup A$ e $\inf A$. Todos los conjuntos parcialmente ordenados completos son retículos. Un retículo L que es completo como conjunto parcialmente ordenado es un *retículo completo*.

Por ejemplo, si L es un conjunto de proporciones, $\vee$ y $\wedge$ denotan la disjunción y la conjunción, respectivamente (i.e. "or" y "and"), entonces las identidades *L1* a *L4* son propiedades conocidasde la lógica

proposicional. Un ejemplo de *retículo completo* son los números naturales, ordenados por divisibilidad. El supremo esta dado por el mínimo común múltiplo y el ínfimo por el máximo común divisor. Para conjuntos infinitos el supremo e ínfimo son el 0 y 1, respectivamente. Si se remueve el 0 del conjunto anteriormente descrito, ya no es un *retículo completo* puesto que no tiene supremo para todos los subconjuntos. Aunque sigue siendo un *retículo*.

2.4. *SMT-Solving*

Satisfacibilidad es el problema de determinar si una fórmula que expresa una restricción tiene un modelo o una solución. Un gran número de problemas pueden ser descritos en términos de satisfacibilidad (i.e., problemas de grafos, rompecabezas como Sudoku, verificación de software y hardware, optimización, generación de casos de prueba). Muchos de estos problemas pueden ser representados por fórmulas Booleanas y resueltos utilizando un solucionador de satisfacibilidad Booleana (del inglés, *Boolean Satisfiability Solver*, abreviado SAT-Solver). Otros problemas requieren la expresividad de la aritmética, cuantificadores, operadores de tipos de datos, entre otros. Esta clase de problemas pueden ser manejados por teorías de satisfacibilidad módulo (del inglés, *Satisfiability Modulo Theories*, abreviado SMT) [**5**]. En este caso, las fórmulas de interés pueden ser expresadas en lógica de primer orden, lo cual hace que *SMT-Solving* sea más general que SAT-Solving. Sin embargo, el problema de decisión asociado deja de ser decidible: este es, básicamente, el precio que se paga por contar con más expresividad en el lenguaje de fórmulas.

El término SMT es usado para describir el problema de satisfacibilidad para una teoría arbitraria o combinaciones de estas. El programa que soluciona este problema es denominado *SMT-solver* [**9**]. En particular, un SMT-Solver es un programa de computador que integra procedimientos de decisión y semi-decisión para fragmentos específicos de la lógica de primer orden y sus teorás respectivas. En este trabajo se usan procedimientos de decisión y semi-decisión sobre los enteros para codificar un sistema de restricciones sobre enteros.

Maude, en sus versiones más recientes utiliza CVC4 como SMT-Solver suministrando funcionalidades para reescritura módulo fórmulas o restricciones en la sintaxis de teorías soportadas por su lenguaje. La extensión de lógica de reescritura a reesritura mótulo SMT es una apuesta reciente para modelar y analizar sistemas abiertos de estados infinitos [**15**].

Capítulo 3

SCCP

La Programación Concurrente por Restricciones (del inglés, *Concurrent Constraint Programming*, abreviado CCP) es un modelo computacional en el cual agentes (v.gr., procesos, usuarios) interactúan sobre un banco de información compartido, el cual puede contener información parcial. Este modelo hace uso de dos operaciones básicas, la operación **ask** intenta inferir si es posible obtener información de los procesos, mientras que la operación **tell** agrega información parcial al banco de los procesos. La notación clásica de las operaciones *read* and *write* es reemplazada en este modelo por la notación **ask** and **tell**.

Las características de CCP lo identifican como un formalismo declarativo que permite la implementación de diferentes sistemas reactivos. En esta área CCP se ha extendido para modelar diferentes tipos de concurrencia y abordar problemas que surgen en la modelación de sistemas reactivos. Dichas extensiones incluyen no-determinismo [**13**], movilidad y comportamiento sincronizado. La extensión mas reciente agrega modalidades epistémicas y espaciales que permiten modelar comportamiento distribuido, que no era posible modelar con las extensiones anteriormente mencionadas.

Este capítulo presenta algunas definiciones particulares requeridas para posteriormente definir las extensiones del modelo CCP. Para más información se refiere el lector a [**8**]. La Sección 3.1 presenta el Sistema de Restricciones Simples. Las secciones 3.2 y 3.3 presentan el sistema espacial y epistémico de restricciones, respectivamente, y sus propiedades. La sección 3.4 presentan SCCP y ECCP, dos variaciones del modelo CCP. Finalmente, la Sección 3.5 presenta la semántica operacional estructurada para espacios y conocimiento en procesos.

3.1. Sistema de restricciones simple

El modelo CCP es paramétrico en un sistema de restricciones (del inglés, *Constraint System*, abreviado *CS*). Esto significa que los parámetros del modelo de procesos son restricciones en algún lenguaje

de primer orden. El *CS* es la estructura que especifica las interdependencias de la información usada en el banco compartido por medio de fórmulas.

De acuerdo con [**4**] las restricciones del sistema se pueden ver como un retículo algebraico completo. Este retículo es provisto de un orden parcial $\sqsubseteq$. Los detalles de un *CS* se introducen en la Definición 3.1.

DEFINICIÓN 3.1. Un *sistema de restricciones* es un retículo algebraico completo

$$C = (Con, Con_0, \sqsubseteq, \sqcup, true, false)$$

donde Con es el conjunto de restricciones, Con_0 es el conjunto de elementos compactos de Con y está ordenado con respecto a $\sqsubseteq$, $\sqcup$ es la operación del límite inferior definido en subconjuntos, y $true$ y $false$ son los elementos mínimo y máximo de Con, respectivamente.

3.2. Sistema espacial de restricciones

Para SCCP es necesario definir un sistema espacial de restricciones, los cuales corresponden a una extensión de CS. El objetivo de SCCP es modelar un sistema distribuido de múltiples agentes en el que cada agente tiene su propio espacio de computación. Esto se obtiene agregando la noción de espacio para un agente, denotado por una función $[c]_i$, que significa que una restricción c esta contenida en el espacio del agente i. La definición de un *sistema espacial de restricciones de n agentes* se presenta en la Definición 3.2.

DEFINICIÓN 3.2. Un *sistema espacial de restricciones de n agentes* (del inglés, *Spatial Constraint System*, abreviado n-SCS) es un sistema de restricciones $\mathbf{C}$ provisto con n funciones $([\cdot]_1, \ldots, [\cdot]_n)$ sobre el conjunto de restricciones *Con*. Cada función $[\cdot]_i : Con \to Con$ debe satisfacer las siguientes propiedades:

S.1 $[true]_i = true$
S.2 $[c \sqcup d]_i = [c]_i \sqcup [d]_i$
S.3 $c \sqsubseteq d \Rightarrow [c]_i \sqsubseteq [d]_i$

Formalmente un n-SCS puede ser denotado

$$C = (Con, Con_0, \sqsubseteq, \sqcup, true, false, [\cdot]_1, \ldots, [\cdot]_n),$$

en donde $(Con, Con_0, \sqsubseteq, \sqcup, true, false)$ es un retículo completo.

La Propiedad S.1 expone que tener un banco local sin información es igual a no tener información globalmente. La Propiedad S.2 permite unir diferente información que está en el mismo espacio. Finalmente, la Propiedad S.3 indica que si la información c puede ser derivada de d, entonces cualquier agente puede derivar esto en su propio espacio.

En un modelo n-SCS nada previene que haya información inconsistente entre agentes. Esto es conocido como *confinamiento de inconsistencias* [**8**]. Es decir, algunos espacios de agentes pueden contener información inconsistente sin que esto invalide la información de otros agentes.

Existe otra propiedad en los sistemas espaciales de restricciones llamada *preservación de diferencia*. Esta propiedad se refiere al hecho de que en algunos casos puede suceder $[c]_i = [d]_i$ con $c \neq d$. Esto puede ser interpretado como la imposibilidad del agente i de distinguir entre c y d. Sin embargo, en algunas aplicaciones esta propiedad puede ser necesaria. En este trabajo no consideramos este tipo de distinciones.

Por último, se debe introducir el concepto de *información compartida* e *información global*. Informalmente, *información compartida* sobre un grupo $\mathbf{G}$ se refiere a la información que es compartida entre agentes de dicho grupo. Mientras que *información global* se refiere al hecho de que información c esta presente en cada espacio anidado.

3.3. Sistema epistémico de restricciones

Mientras que la información en un n-SCS representa lo que puede ser entendido como creencias, el objetivo de un sistema epistémico de restricciones es modelar conocimiento local y global (i.e., verdades y hechos). En este orden de ideas, la información c es un hecho que un agente i conoce y esto se denota como $[c]_i$ por medio de los operadores de clausura del n-SCS. De esta forma la información global del sistema corresponde al cúmulo de todas las informaciones locales del sistema.

Es importante aclarar que en el *Sistema Epistémico de Restricciones* no se presenta el concepto de información inconsistente, puesto que no no es posible que un agente tenga conocimiento inconsistente. En otras palabras, todo conocimiento c de un agente debe ser cierta y por lo tanto no se aplica el concepto de inconsistencia.

Definición 3.3. Un *sistema epistémico de restricciones de n agentes* (del inglés, *Epistemic Constraint System*, abreviado n-ECS) es un sistema n-SCS cuyas funciones de espacio $[\cdot]_1, \ldots, [\cdot]_n$ también son operadores de clausura y, ademas de las propiedades S.1, S.2 y S.3, estas funciones satisfacen:

S.4 $c \sqsubseteq [c]_i$
S.5 $[[c]_i]_i = [c]_i$

La Propiedad S.4 expone que si un agente conoce cierta información, entonces esa información debe ser cierta. Esto significa que $[c]_i$ contiene por lo menos tanta información como c. De otro lado, la Propiedad S.5

se refiere al hecho de que un agente es consciente con respecto a la información que conoce.

3.4. Espacios y conocimiento en procesos

Ahora se presentan dos variantes del modelo CCP llamados programación espacial concurrente de restricciones y programación epistémica concurrente de restricciones (SCCP y ECCP, por sus siglas en inglés). El primero se refiere a un cálculo que solo permite ejecutar procesos dentro del espacio de un agente, posiblemente anidado, mientras que el segundo extiende este comportamiento a la interacción entre agentes solicitando y procesando conocimiento dentro de la distribución espacial de información [8]. Cada extensión utiliza un SCS y ECS, respectivamente.

3.4.1. Sintaxis de procesos. A continuación se presenta la sintaxis para los procesos en SCCP y ECCP.

Definición 3.4. Los términos de ambos lenguajes es presentada por la siguiente sintaxis:

$$P, Q \ldots ::= \mathbf{0} \mid \mathbf{tell}(c) \mid \mathbf{ask}(c) \rightarrow P \mid P \parallel Q \mid [P]_i \,.$$

Para SCCP el sistema de restricción es n-SCS, y para ECCP es n-ECS. A continuación se describe brevemente el significado de cada operador:

- **0** representa la inactividad de un proceso, es decir, que no hace nada.
- El proceso **tell(c)** agrega información al banco.
- El proceso **ask(c)** $\rightarrow P$ verifica si la restricción c es una implicación del banco actual y después ejecuta el proceso P.
- El proceso $P \parallel Q$ es para la ejecución en paralelo de procesos.
- El proceso $[P]_i$ ejecuta P dentro del espacio de computación del agente i.

3.5. Semántica operacional estructurada

En la semántica operacional estructurada (del inglés, *Structural Operational Semantics*, abreviado SOS) la reducción es hecha por medio de configuraciones de la forma $\langle P, d\rangle \rightarrow \langle P', d'\rangle$, donde P, P' denotan procesos, y c, c' denotan el banco de los procesos. Las reglas comunes entre ambos lenguajes se expresan a continuación. El simbolo γ representa un grupo de configuraciones.

Las reglas mostradas en la Figura 3.1 presentan la semántica operacional estructurada para SCCP. Por ejemplo, en la regla $\mathrm{R}_{\mathrm{TELL}}$ la

$$\frac{}{\langle \mathbf{tell}(c); d\rangle \longrightarrow \langle \mathbf{0}; d \sqcup c\rangle}\ \mathrm{R_{TELL}} \qquad \frac{c \sqsubseteq d}{\langle \mathbf{ask}\ c\ \rightarrow\ P; d\rangle \longrightarrow \langle P; d\rangle}\ \mathrm{R_{ASK}}$$

$$\frac{\langle P; d\rangle \longrightarrow \langle P'; d'\rangle}{\langle P \parallel Q; d\rangle \longrightarrow \langle P' \parallel Q; d'\rangle}\ \mathrm{R_{PAR}} \qquad \frac{\langle P; c^i\rangle \longrightarrow \langle P'; c'\rangle}{\langle [P]_i\,; c\rangle \longrightarrow \langle [P']_i\,; c \sqcup [c']_i\rangle}\ \mathrm{R_{SP}}$$

FIGURA 3.1. SOS para SCCP.

$$\frac{\langle P; c\rangle \longrightarrow \langle P'; c'\rangle}{\langle [P]_i\,; c\rangle \longrightarrow \langle [P']_i \parallel P'; c\rangle}\ \mathrm{R_{SP}}$$

FIGURA 3.2. SOS para ECCP.

restricción c se agrega al banco y luego el proceso se reduce a $\mathbf{0}$. En el caso de la regla $\mathrm{R_{ASK}}$, si la guarda c es implicación del banco, entonces el proceso se reduce y se ejecuta P. $\mathrm{R_{PAR}}$ muestra que se puede ejecutar P o Q. Por último, el proceso que es ejecutado dentro del espacio de un agente, la restricción se agrega dentro del espacio de dicho agente. Estas reglas funcionan para las reducciones comunes en SCCP y ECCP. Sin embargo, en ECCP hay una regla adicional la cual permite modelar el hecho de que la información conocida por un agente i se convierte en un hecho. Esta regla se define a en la Figura 3.2.

Después de definir el lenguaje que se va a usar, se concluye este capítulo con un ejemplo que será usado posteriormente.

3.6. Ejemplo

Se puede considerar un modelo SCCP y ECCP como un árbol. La raíz del árbol corresponde al espacio o agente más exterior, denominado como el *super agente*. Cada nodo del árbol corresponde a la información contenida en el espacio de cada agente. Las aristas definen la jerarquía de los agentes. Este ejemplo se desarrolla utilizando formulas aritméticas sin cuantificadores como sistemas de restricciones.

La Figura 3.3 corresponde al estado inicial del SCS

$$d \stackrel{\mathtt{def}}{=} (V > 10) \sqcup [W = 3 \sqcup [X = 17]_3]_1 \sqcup [Y > 4 \sqcup [Z = 20]_1]_2.$$

Se puede utilizar un proceso de la forma $[\mathbf{tell}(m)]_i$ para representar un mensaje m para un agente i. De forma que, el proceso $[\mathbf{tell}(T = 51)]_1$ agregara esa restricción al banco de información del agente 1. Un proceso de la forma $[\mathbf{ask} Z > 10 \rightarrow S > 3]_2$ no puede evolucionar debido a que no es posible derivar $Z > 10$ del banco de información del agente 2 (i.e., $Y < 0$). Mientas que el proceso $[[\mathbf{ask} Z > 10 \rightarrow S > 3]_1]_2$ si puede evolucionar puesto que $Z > 10 \sqsubseteq Z = 20$. Un proceso de

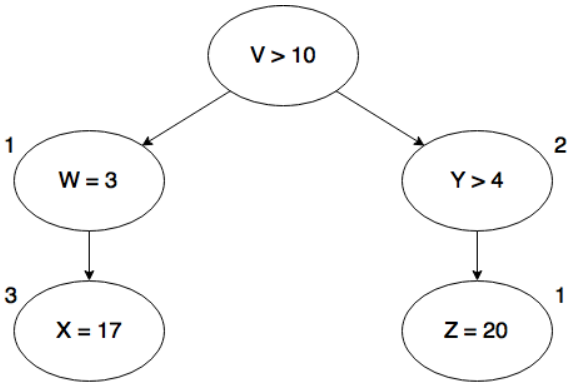


FIGURA 3.3. Jerarquía de agentes

la forma $[[\mathbf{ask} Z > 10 \rightarrow S > 3]_1]_2 \| [\mathbf{tell}(T = 51)]_1$ ejecuta ambos procesos al mismo tiempo, de forma que el resultado es el descrito anteriormente.

Si se define,

$R \stackrel{\text{def}}{=} [V = 42]_3 \parallel T = 8$,

$P \stackrel{\text{def}}{=} \mathbf{ask}\ Y > 0 \ \rightarrow\ Z > 10$,

$Q \stackrel{\text{def}}{=} \mathbf{ask}\ T > 7 \ \rightarrow\ [S \neq 2]_2$.

entonces, el proceso $S = [R]_1 \parallel [P]_2 \parallel [Q]_1$ debe evolucionar tal como lo describe la Figura 3.4.

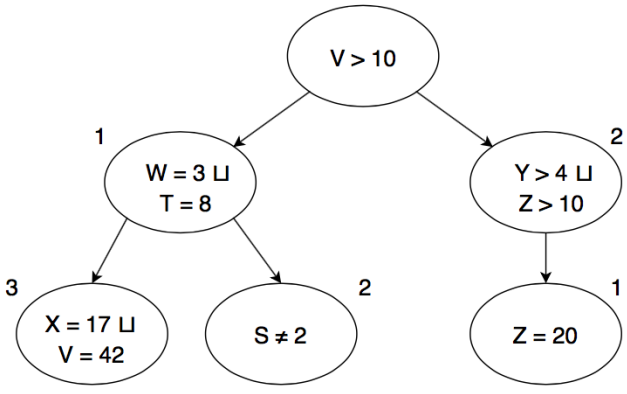


FIGURA 3.4. Evolución del sistema

En el Capítulo 4 se muestra esta derivación utilizando la especificación formal.

Capítulo 4

Una teoría de reescritura para SCCP

Este capítulo presenta una especificación formal del modelo SCCP en la sintaxis de Maude **[3]**. La especificación formal consta de un módulo funcional que define el identificador de un agente y que a su vez indica su ubicación en la jerarquía de procesos; un módulo funcional que define la sintaxis y los tipos requeridos para especificar los diferentes tipos de comandos u operadores disponibles en el modelo; y un módulo funcional que define la sintaxis y los tipos requeridos para especificar los estados del modelo. Este módulo incluye, entre otros, la definición de un agente y un proceso. La transferencia de información en SCCP esta formalizada a partir de de reglas de reescritura que son parte de un módulo de sistema.

La especificación formal utiliza el modulo `SMT`, una técnica innovadora que combina el capacidad de la reescritura de términos, algoritmos de emparejamiento (*matching*), y un SMT-solver. Por medio del módulo `SMT` las restricciones son codificadas como formulas en una teoría con relación de satisfacibilidad determinada por el SMT-solver.

Adicionalmente, este capítulo presenta ejemplos de algunas simulaciones del modelo que usan la especificación formal y el comando `rewrite` de Maude.

Las secciones 4.1, 4.2 y 4.3 presentan los módulos funcionales `AGENT-ID`, `SCCP-SYNTAX` y `SCCP-STATE`, respectivamente. La Sección 4.4 presenta el módulo de sistema `SCCP` que especifica las transiciones del modelo. Finalmente, la Sección 4.5 presenta algunos ejemplos de simulación del modelo usando el comando `rewrite` de Maude.

4.1. Modelamiento formal del identificador

El identificador de un agente está representado por el sort `Aid` definido en el modulo funcional `AGENT-ID` de la siguiente manera:

```
sort Aid .
pr NAT .
subsort Nat < Aid .
```

```
op root : -> Aid .
op _._ : Aid Aid -> Aid [assoc left id: root] .
```

El identificador está definido como una secuencia de números naturales, representados por `NAT`, concatenados con punto. El identificador es asociativo y tiene un elemento identidad izquierdo. En la sintaxis de Maude, estos atributos se agregan incluyendo las palabras clave `assoc` y `left id` en la declaración de la operación.

La identidad izquierda de un identificador está representada por `root`, de forma que `root.1` es equivalente a `1`. Los números de izquierda a derecha indican el espacio y subespacio al cual pertenece el agente, es decir, el identificador `4.2.1` significa que el agente pertenece al espacio `4`, dentro de este al subespacio `2`, tal como se indica en la Figura 4.1.

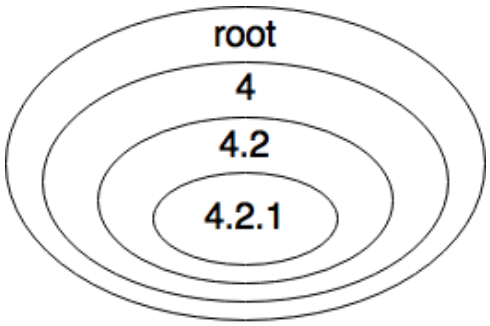


FIGURA 4.1. Identificador de un agente

4.2. Modelamiento formal de los operadores

Los operadores descritos en la Definición 3.4 están representados por el sort `SCCPCmd` en el modulo funcional `SCCP-SYNTAX` tal como se muestra a continuación:

```
sort SCCPCmd .

op 0 : -> SCCPCmd .
op tell_ : Boolean -> SCCPCmd .
op ask_->_ : Boolean SCCPCmd -> SCCPCmd .
op ask<_>_->_ : Aid Boolean SCCPCmd -> SCCPCmd .
op _||_ : SCCPCmd SCCPCmd -> SCCPCmd [assoc gather (e E) ] .
op <_>[_] : Aid SCCPCmd -> SCCPCmd .
```

El argumento del operador `tell_` es la restricción que se va a adicionar al banco de información de un agente, representado por un elemento

de tipo `Boolean`. El operador `ask_->_` tiene como argumentos la condición a ser evaluada dentro del banco de información del agente y el proceso a llevarse a cabo. Cada uno de estos argumentos está representado por elementos del tipo `Boolean` y `SCCPCmd`, respectivamente.

El operador `ask_->_` permite verificar la satisfacibilidad de la restricción dentro del banco de información de un agente, en este modelo se presenta una extensión a este operador mediante la cual se puede verificar la satisfacibilidad de la restricción en un agente de menor jerarquía (i.e., el agente con identificador 4 puede preguntar al agente con identificador 4.2 o 4.3.1). Este operador esta definido como `ask<_>_->_` y adicionalmente tiene como argumento el identificador del agente dentro de su jerarquía, representado por un elemento de tipo `Aid`.

El operador `_||_` tiene como argumentos los procesos que se van a ejecutar en paralelo, representados por elementos de tipo `SCCPCmd`. Finalmente, los argumentos del operador `<_>[_]` son el identificador del agente y el proceso a ejecutar, representados por elementos del tipo `Aid` y `SCCPCmd`, respectivamente.

4.3. Modelamiento formal de agentes y procesos

Los agentes y procesos están representados como objetos por el tipo `Obj` definido en el modulo funcional `SCCP-STATE` de la siguiente manera:

```
sorts Cid Obj Cnf Sys .
subsorts Obj < Cnf .

ops store process : -> Cid .
op [_,_,_] : Cid Aid Boolean -> Obj [ctor] .
op [_,_,_] : Cid Aid SCCPCmd -> Obj [ctor] .
op mt : -> Cnf [ctor] .
op __ : Cnf Cnf -> Cnf [ctor assoc comm id: mt] .
op {_} : Cnf -> Sys [ctor] .
```

El tipo `Cid` representa el identificador de clase, en este caso agentes y procesos designados como `store` y `process`, respectivamente. Un agente está representado como un objeto de la forma `[_,_,_]` cuyos argumentos son el identificador de clase (i.e., `store`), el identificador del agente y su banco de información. Cada uno de estos argumentos está representado por elementos del tipo `Cid`, `Aid` y `Boolean`, respectivamente.

Los argumentos de un proceso son el identificador de clase (i.e., `process`), el identificador del agente y el proceso a ejecutar dentro

del espacio de dicho agente. Estos argumentos están representados por elementos del tipo `Cid`, `Aid` y `SCCPCmd`, respectivamente.

Los estados del sistema están representados por el tipo `Sys` definido como una sopa de agentes y procesos representada por el tipo `Cnf`; dicha sopa es conmutativa, asociativa y tiene elemento identidad `mt`. En la sintaxis de Maude, estos atributos se agregan incluyendo las palabras clave `comm`, `assoc` y `id` en la declaración de las operaciones.

4.4. Modelamiento formal de las transiciones

Las transiciones concurrentes del sistema están modeladas por seis reglas de reescritura en la sintaxis de Maude y que definen la semántica de los operadores descritos en la Sección 4.2. En las transiciones, los procesos se ejecutan y el banco de información de los agentes puede ser modificado. Las regla de reescritura aplican sobre los estados del sistema, es decir, sobre un elemento de tipo `Sys` definido e la Sección 4.3. La ecuación `exists-store?` en la regla `ext-ask1` verifica si en el sistema existe un agente a partir de su identificador, cada uno de estos argumentos está representado por elementos del tipo `Cnf` y `Aid`, respectivamente. Las reglas de reescritura se especifican a continuación:

```
 rl [tell] :
    { [ store, L, B] [process, L, tell B0 ] X }
 => { [ process, L, 0 ] [ store, L, B and B0] X } .

crl [ask] :
    { [ store, L, B ] [ process, L, ask B0  -> C0 ] X }
 => { [ store, L, B ] [ process, L, C0 ] X }
 if check-unsat(B and not(B0)) .

crl [ask-ext0] :
    { [ store, L0 . L1, B ]
      [ process, L0, ask< L1 > B0  -> C0 ] X }
 => { [ store, L0 . L1, B ] [ process, L0, C0 ] X }
 if check-unsat(B and not(B0)) .

crl [ext-ask1] :
    { [ process, L0, ask< L1 > B0  -> C0 ] X }
 => { [ process, L0, C0 ] X }
 if exists-store?(X, L0 . L1) == false
 /\ check-unsat(not(B0)) .

 rl [parallel] :
```

```
   { [ process, L, C0 || C1 ] X }
=> { [ process, L, 0 ] [ process, L, C0 ]
     [ process, L, C1 ] X } .

rl [space] :
   { [ process, L0, < L1 >[ C0 ] ] X }
=> { [ process, L0, 0 ] [ process, L0 . L1, C0]
     [ store, L0 . L1, true ] X } .
```

Las reglas `[ask]`, `[ask-ext0]` y `[ext-ask1]` son reglas condicionales, es decir, requieren que una condición especifica se cumpla para poder ser ejecutadas, dicha condición debe ser de tipo `Boolean`. En particular, un llamado

`check-unsat(`ϕ`)`

en el cual ϕ es una restricción, resuelta en una invocación al SMT-Solver disponible en Maude. Esta invocación resulta en un valor Booleano indicando si la fórmula ϕ es satisfacible o no. Note que ϕ no es satisfacible siempre y cuando ϕ es una fórmula válida en el modelo sobre el cual opera el SMT-Solver. De esta manera, las tres reglas mencionadas anteriormente definen transiciones en los cuales la restricción sobre la cual se consulta al SMT-Solver es válida, tal y como se define la semántica de SCCP en la Sección 3.2.

Adicionalmente, se incluyen dos ecuaciones correspondientes a transiciones concurrentes no observables. La primer ecuación reemplaza un proceso inactivo (i.e., que no hace nada) por el elemento identidad del sistema `mt`, y la segunda ecuación une dos agentes con el mismo identificador mediante una conjunción lógica de sus bancos de información. Estas ecuaciones se especifican a continuación:

```
eq [ process, L, 0 ]
 = mt .
eq [ store, L, B0 ] [ store, L, B1 ]
 = [ store, L, B0 and B1 ] .
```

4.5. Ejemplo

Esta sección muestra cómo especificar y ejecutar el sistema de la Sección 3.6 usando la especificación de los operadores en la Sección 4.2, los agentes y procesos en la Sección 4.3 y las reglas de reescritura de la Sección 4.4.

En el ejemplo de la Sección 3.6, el estado inicial d puede ser escrito en la sintaxis de la especificación Maude de la siguiente manera:

```
{ [store, root, (V:Integer > 10)]
  [store, 1, (W:Integer === 3)]
  [store, 1 . 3, (X:Integer === 17)]
  [store, 2, (Y:Integer > 4)]
  [store, 2 . 1, (Z:Integer === 20)] }
```

Los procesos R, P y Q se definen respectivamente cómo:

```
[process, 1, ((< 3 >[tell(V:Integer === 42)]) ||
              (tell (T:Integer === 8)))]
[process, 2, (ask(Y:Integer > 0) -> tell(Z:Integer > 10))]
[process, 1, (ask(T:Integer > 7) ->
              < 2 >[tell(S:Integer =/== 2)])]
```

Por lo tanto, el proceso $S = [R]_1 \parallel [P]_2 \parallel [Q]_1$ puede ser escrito como se muestra a continuación::

```
[process, root, (((< 1 >[(< 3 >[tell(V:Integer === 42)]) ||
  (tell (T:Integer === 8))]) ||
  (< 2 >[ask(Y:Integer > 0) -> tell(Z:Integer > 10)])) ||
  (< 1 >[ask(T:Integer > 7) -> < 2 >[tell(S:Integer =/== 2)]]))]
```

El proceso S se incluye en el estado inicial del sistema d y se ejecuta en el ambiente de Maude mediante el comando de reescritura `rewrite`, con el fin de conocer la evolución del sistema luego de aplicar todas las reglas de reescritura posibles.

```
rew { [store, root, (V:Integer > 10)]
  [store, 1, (W:Integer === 3)]
  [store, 1 . 3, (X:Integer === 17)]
  [store, 2, (Y:Integer > 4)]
  [store, 2 . 1, (Z:Integer === 20)]
  [process, root, (((< 1 >[(< 3 >[tell(V:Integer === 42)]) ||
     (tell (T:Integer === 8))]) ||
     (< 2 >[ask(Y:Integer > 0) -> tell(Z:Integer > 10)])) ||
     (< 1 >[ask(T:Integer > 7) ->
      < 2 >[tell(S:Integer =/== 2)]]))] } .
```

El estado final del sistema alcanzado por la especificación coincide con lo descrito en la Figura 3.4. En el resultado se observan los 6 agentes con sus respectivos identificadores y bancos de información. El resultado de la reescritura es el siguiente:

```
{ [store,root,V:Integer > (10).Integer]
  [store,(1).NzNat,W:Integer === (3).Integer
```

```
          and T:Integer === (8).Integer]
  [store,(2).NzNat,Y:Integer > (4).Integer
          and Z:Integer > (10).Integer]
  [store,1 . 2,S:Integer =/== (2).Integer]
  [store,1 . 3,X:Integer === (17).Integer
          and V:Integer === (42).Integer]
  [store,2 . 1,Z:Integer === (20).Integer] }
```

Finalmente, se presenta un ejemplo de la extensión del operador **ask** descrito en la Sección 4.2. Considere el mismo estado inicial d, y el siguiente proceso:

```
[process, root, (ask< 1 >(W:Integer < 15) ->
   < 2 . 1 >[tell(M:Integer >= 4)])]
```

Este proceso verifica la satisfacibilidad de la restricción `W:Integer <15` dentro del banco de información del Agente `1` y posteriormente ejecuta el proceso `<2 . 1 >[tell(M:Integer >= 4)]`, el cual agrega la restricción `M:Integer >= 4` al banco de información del Agente `2.1`, tal como se muestra a continuación:

```
{ [store,root,V:Integer > (10).Integer]
  [store,(1).NzNat,W:Integer === (3).Integer]
  [store,(2).NzNat,Y:Integer > (4).Integer]
  [store,1 . 3,X:Integer === (17).Integer]
  [store,2 . 1,Z:Integer === (20).Integer
          and M:Integer >= (4).Integer] }
```

Capítulo 5

Lenguaje de programación

Este capítulo presenta una propuesta de un lenguaje de programación basado en el modelo SCCP definido en el Capítulo 3 y cuya semántica operacional está dada por la teoría de reescritura en el Capítulo 4. La principal razón para crear este lenguaje es permitir a los programadores tener una medio más sencillo para modelar sistemas distribuidos de información.

La Sección 5.1 presenta la sintaxis del lenguaje de programación en la notación EBNF. La Sección 5.2 muestra cómo el lenguaje esta relacionado con el modelo SCCP descrito en el Capítulo 3. La Sección 5.3 presenta una explicación de los nuevos elementos del lenguaje. Por último, la Sección 5.4 contiene algunos ejemplos del funcionamiento del lenguaje de programación.

5.1. EBNF del lenguaje

EBNF (del ingés, *Extended Backus-Naur Form*) es un metalenguaje utilizado para definir formalmente la sintaxis de gramáticas libres de contexto, que utiliza expresiones regulares para permitir escribir especificaciones compactas [**7**].

Una descripción EBNF es una lista no ordenada de reglas EBNF. Cada una de las reglas EBNF esta compuesta de tres partes: el lado izquierdo, el lado derecho y el símbolo '::=' separando los dos lados, este símbolo debe ser leido "se define como" [**7**]. El lado izquierdo contiene una palabra escrita en minúscula y cursiva delimitada por los caracteres '⟨' y '⟩', la cual indica el nombre de la regla EBNF. En el lado derecho se encuentra la definición asociada a este nombre.

Los nombres asignados en el lado izquierdo de las reglas EBNF pueden ser utilizados como parte de la definición asociada en el lado derecho de las reglas, inclusive dentro de la misma regla, es decir, que puede haber recursión en la descripción del lenguaje.

Las reglas EBNF pueden incluir diez caracteres con significado especial: '::=', '|', '+', '∗', '[', ']', '(', ')', '?' y ';'. Todos los demás caracteres diferentes de los listados anteriormente y de los nombres de las reglas

$$
\begin{aligned}
\langle system \rangle &::= \langle variables \rangle * \langle body \rangle \ ; \\
\langle variables \rangle &::= \texttt{`var'}\ \langle id \rangle +\ (\texttt{`Int'}|\texttt{`Bool'})\ ; \\
\langle body \rangle &::= \texttt{`begin'}\ \langle processline \rangle +\ \texttt{`end'}\ ; \\
\langle processline \rangle &::= \langle process \rangle\ \texttt{`.'}\ ; \\
\langle process \rangle &::= \texttt{`tell('}\ \langle constraint \rangle\ \texttt{`)'}\ ; \\
&\quad | \ \texttt{`ask'}\ (\texttt{`<'}\ \langle location \rangle\ \texttt{`>'})?\ \langle constraint \rangle\ \texttt{`->'}\ \langle process \rangle\ ; \\
&\quad | \ \langle process \rangle\ \texttt{`||'}\ \langle process \rangle\ ; \\
&\quad | \ \texttt{`['}\ \langle process \rangle\ \texttt{`]\_'}\ \langle integer \rangle\ ; \\
\langle constraint \rangle &::= \langle boolean \rangle\ ; \\
&\quad | \ \langle id \rangle\ ; \\
&\quad | \ \langle expression \rangle\ ; \\
&\quad | \ \langle constraint \rangle\ \texttt{`and'}\ \langle constraint \rangle\ ; \\
\langle location \rangle &::= \langle integer \rangle\ (\texttt{`.'}\ \langle integer \rangle)*\ ; \\
\langle expression \rangle &::= \langle id \rangle\ \langle operator \rangle\ (\langle id \rangle\ |\ \langle integer \rangle)\ ; \\
\langle operator \rangle &::= \texttt{`>'}\ |\ \texttt{`<'}\ |\ \texttt{`=`}\ |\ \texttt{`=/=`}\ |\ \texttt{`>=`}\ |\ \texttt{`<=`}\ ; \\
\langle boolean \rangle &::= \texttt{`true'}\ |\ \texttt{`false'}\ ; \\
\langle integer \rangle &::= [0\text{-}9]+\ ; \\
\langle id \rangle &::= [A\text{-}Z]\ [A\text{-}Z0\text{-}9]*\ ;
\end{aligned}
$$

FIGURA 5.1. EBNF del lenguaje de programación.

EBNF se definen por si mismos (v.gr., letras, dígitos, signos de puntuación). A continuación se presenta una explicación de algunos de los caracteres especiales:

- Cada regla EBNF debe finalizar con el caracter ';'.
- '$*$' es un operador unario posfijo, el cual indica que la expresión aparece cero o más veces.
- '$+$' es un operador unario posfijo, el cual indica que la expresión aparece una o más veces.
- '?' es un operador unario posfijo, el cual indica que la expresión aparece cero o una vez.
- Múltiples opciones en la definición de un regla EBNF deben estar separadas por el caracter '|'.
- Todas las expresiones delimitadas por comillas simples aparecen de la misma forma en el lenguaje de programación.

En la Figura 5.1 se presenta la descripción EBNF del lenguaje de programación propuesto. La explicación de cada una de las reglas EBNF se presenta en las secciones 5.2 y 5.3.

5.2. Relación con SCCP

El lenguaje de programación que se propone está basado en el modelo SCCP presentado en el Capítulo 3, y es ejecutado en el ambiente de Maude por medio de la especificación formal presentada en el Capítulo 4. Por esta razón el lenguaje debe proveer la sintaxis parar los procesos que se han definido hasta el momento.

Esta sección contiene una explicación de la relación entre la definición de SCCP y algunas de las reglas EBNF de la Figura 5.1. En primer lugar se define el identificador de un agente denominado `location`.

- ⟨*integer*⟩ está definido como cualquier valor entre 0 y 9 repetido una o más veces.
- ⟨*location*⟩ está definido como un ⟨*integer*⟩ unido a cero o más repeticiones de un '`.`' y un ⟨*integer*⟩.

Los procesos que se describen en la Definición 3.4 y la Sección 4.2 son nombrados ⟨*process*⟩, para definirlo se requiere de algunos elementos adicionales, entre los que se encuentran, la definición de un banco de información de un agente o la restricción como argumento de un proceso, nombrado ⟨*constraint*⟩. Estos elementos se explican a continuación:

- ⟨*boolean*⟩ está definido como cualquier valor entre `true` y `false`.
- ⟨*operator*⟩ está definido como cualquier valor entre '`>`', '`<`', '`=`', '`=/=`', '`>=`' y '`<=`'.
- ⟨*expression*⟩ está definido como un ⟨*id*⟩, seguido de un ⟨*operator*⟩ y un ⟨*id*⟩ o un ⟨*integer*⟩.
- ⟨*constraint*⟩ está definido como cualquier opción entre ⟨*boolean*⟩, ⟨*id*⟩, ⟨*expression*⟩ y dos ⟨*constraint*⟩ unidos por una conjunción lógica (i.e., '`and`').

Un proceso, denominado ⟨*process*⟩ en el lenguaje de programación, hace referencia a los 5 procesos del modeloSCCP, incluyendo la extensión de **ask** descrita de la Sección 4.2. La sintaxis es similar a la presentada en la especificación formal. El proceso **tell** debe incluir un ⟨*constraint*⟩ contenido entre paréntesis. El proceso **ask** contiene un ⟨*constraint*⟩ y un ⟨*process*⟩, y puede incluir un elemento de tipo ⟨*location*⟩ relacionando al identificador de un agente de menor jerarquía. El proceso de ejecución en paralelo contiene los procesos a ser

ejecutados en paralelo. Finalmente, el proceso de especificación del espacio de ejecución requiere del proceso a ejecutar ⟨*process*⟩ y un numero ⟨*integer*⟩ referente al identificador del agente de menor jerarquía donde se va a ejecutar el proceso.

5.3. Nuevas características

Debido a que el objetivo del lenguaje propuesto es brindar a los programadores un medio más sencillo para modelar sistemas distribuidos de información por medio del modelo SCCP, se incluyen algunas características que faciliten el uso del lenguaje y la especificación de un sistema, como la declaración de variables.

Un programa ⟨*system*⟩ tiene dos partes: el encabezado ⟨*variables*⟩ y el cuerpo ⟨*body*⟩. El encabezado contiene la declaración de las variables a las cuales se les asigna un nombre ⟨*id*⟩ y un tipo (v.gr., `'Int'` o `'Bool'`), por ejemplo la sentencia `var X T Int` indica que `X` y `T` son variables de tipo entero. Un programa puede incluir cero o más declaraciones, y un ⟨*id*⟩ no puede corresponder a dos variables de diferentes tipos. Toda variable utilizada dentro del cuerpo del programa debe estar declarada en el encabezado, de lo contrario se producirá un error en la compilación del programa respectivo.

El cuerpo del programa debe comenzar con la palabra `'begin'` y finalizar con `'end'`; si alguna de estas palabras no se incluye se producirá un error en la compilación. Cada línea del cuerpo del programa debe corresponder a uno de los procesos explicados en la Sección 5.2 finalizada con el caracter `'.'`. El interprete omite todas las lineas que se encuentren después de la palabra `'end'`.

5.4. Ejemplo

Para mayor claridad en el uso del lenguaje de programación en la Sección 5.4 se presentan algunos ejemplos. Considere el ejemplo que se trabajó en las secciones 3.6 y 4.5. El programa requerido para obtener el estado inicial del sistema d se presenta a continuación:

```
var V W X Y Z Int
begin
tell(V > 10) .
[tell(W = 3)]_1 || [[tell(X = 17)]_3]_1 .
[tell(Y > 4)]_2 || [[tell(Z = 20)]_1]_2 .
end
```

Todas los procesos pueden ser escritos en una sola línea utilizando procesos en paralelo; para facilidad en la lectura en el ejemplo anterior

se presentan múltiples líneas de forma que se genera el árbol de la Figura 3.3 de izquierda a derecha. Ahora se definene los procesos R, P y Q respectivamente como:

```
[tell(V = 42)]_3 || tell(T = 8)
ask Y > 0 -> tell(Z > 10)
ask T > 7 -> [tell(S =/= 2)]_2
```

Con los procesos R, P y Q se construye S y se agrega al estado inicial para alcanzar el estado final del sistema presentado en la Figura 3.4. El programa correspondiente se muestra a continuación:

```
var V W X Y Z Int
var S T Int
begin
tell(V > 10) .
[tell(W = 3)]_1 || [[tell(X = 17)]_3]_1 .
[tell(Y > 4)]_2 || [[tell(Z = 20)]_1]_2 .
[[tell(V = 42)]_3 || tell(T = 8)]_1 ||
[ask Y > 0 -> tell(Z > 10) ]_2 ||
[ask T > 7 -> [tell(S =/= 2)]_2 ]_1 .
end
```

Tal como se indicó en la Sección 5.3 un proceso debe terminar con el caracter '.', por lo tanto cuando el proceso es extenso como S puede ser separado en múltiples líneas para facilitar su manejo. En este capítulo se presenta el programa que lleva al estado final del sistema; su ejecución y verificación se realiza en el Capítulo 6.

Considere otro sistema descrito por el siguiente programa:

```
var B0 B1 Bool
var X C Int
var Y B Int
begin
tell(true) .
ask true -> tell(X >= 5) .
[[tell(B0)]_1 || ask < 1 > B0 -> tell(Y < X)]_1 .
tell(true) || ask X > 1 -> tell(B1) .
[tell(X >= 5)]_2 .
ask B1 -> tell(C >= 5) .
end
```

A partir del programa descrito anteriormente, el estado final esperado del sistema es representado en la Figura 5.2. Debido a que la

información que tiene o transmite un agente simboliza relaciones o restricciones sobre las variables del sistema, el banco de infomación de un agente puede ser representado por elementos de tipo `Bool`, tal como las variables `B0` y `B1`.

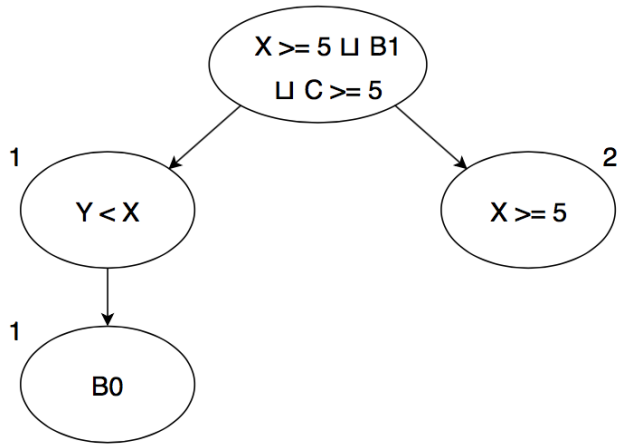


FIGURA 5.2. Estado inicial ejemploSCCP

La ejecución y verificación del programa se realiza en el Capítulo 6.

Capítulo 6

Ambiente para el lenguaje de programación

El programador requiere una herramienta que le permita ejecutar sistemas distribuidos basados en SCCP modelados en el lenguaje de programación presentado en el Capiťulo 5. Este capítulo presenta una herramienta desarrollada en Python 3 con una interfaz gráfica sencilla y fácil de usar, con la que se puede simular la ejecución de un sistema luego de realizar múltiples operaciones de transferencia de información entre diferentes agentes. La interpretación del lenguaje de programación se realiza por medio de un parser y un analizador léxico con los que se genera el código equivalente en Maude. Luego de obtener el estado final del sistema en Maude, este *se traduce* para mostrar el resultado al usuario por medio de la interfaz gráfica.

La Sección 6.1 presenta la descripción del parser utilizado en la herramienta. La Sección 6.2 presenta algunos detalles de su implementación y funcionamiento. La Sección 6.3 presenta una introducción a la interfaz gráfica y las funcionalidades de la herramienta. Finalmente, la Sección 6.4 presenta algunos ejemplos prácticos del uso de la herramienta propuesta.

6.1. Parser

Cuando se interpreta un lenguaje, es de mucha utilidad usar un árbol de sintaxis concreta (del inglés, *Concrete Syntax Tree*, abreviado CST), tradicionalmente denominados *parse tree*, para representar la estructura sintáctica del código fuente de acuerdo con alguna gramática libre de contexto. Este CST puede ser construido por un parser en el proceso de traducción del código fuente y compilación.

Un *parser* es un programa que a partir de una entrada (i.e., código fuente) produce una estructura de datos (i.e., CST o parse tree) basado en una representación estructural del lenguaje de programación utilizado para generar la entrada (i.e., EBNF), verificando durante el proceso que la sintaxis sea correcta. El parser es precedido por un analizador léxico o *lexer*, el cual crea expresiones o *tokens* a partir de una secuencia de caracteres.

Sin embrago, construir un parser desde cero es complicado e innecesario en muchas ocasiones debido a que existen múltiples herramientas que proveen la construcción del CST de forma automatizada en diferentes lenguajes de programación, entre ellos Python (v.gr., ANTLR, PyPEG, Parsiomonious).

La herramienta propuesta brinda al programador una interacción sencilla a través del lenguaje de programación presentado en el Capítulo 5, el cual es ejecutado transparentemente por el ambiente de Maude el cual interpreta especificaciones en la sintaxis de la especificación formal del Capítulo 4. Para lograrlo el objetivo es implementar una traducción por medio de un parser. Para este trabajo se escogio ANTLR [**14**], un generador para leer, procesar y traducir texto estructurado o archivos binarios.

ANTLR genera un parser en Python a partir de la descripción EBNF del lenguaje de programación. El parser puede construir el *parse tree* de cualquier código fuente escrito correctamente en el lenguaje de programación. Mediante un lexer que se desarrollado específicamente para dicho lenguaje se genera el código equivalente en Maude.

6.2. Detalles de la implementación

La herramienta propuesta está desarrollada en Python 3, un lenguaje de programación de alto nivel. Debido a que el análisis del sistema SCCP se realiza en Maude, la herramienta provee un enlace entre el parser generado por ANTLR y del ambiente de Maude, a través del modulo `subprocess` el cual permite crear nuevos procesos y conectarse con su canal de entrada, salida y errores. Este módulo permite ejecutar comandos en la terminal y obtener la salida, tal como se muestra a continuación:

```
path = subprocess.check_output("pwd",shell=True).
```

En este caso se usa el método `subprocess.check_output` para ejecutar el comando `pwd` y obtener su resultado, que en este caso es el directorio donde se está ejecutando el proceso.

Adicionalmente, el estado final del sistema que se obtiene de la ejecución en Maude es interpretada por una función que hace las veces de traductor para brindar al usuario un resultado fácil de entender.

La herramienta está divida en dos directorios: Python y Maude. En el primer directorio se encuentran la interfaz gráfica, el parser y el lexer; en el segundo se encuentra la especificación formal. La ejecución de la herramienta se realiza a través de la línea de comandos.

6.3. Interfaz gráfica

La interfaz gáfica de la herramienta está desarrollada por medio de Tkinter, una librería comúnmente utilizada para el desarrollo de aplicaciones GUI (del inglés, *Graphical User Interface*) en Python. Esta herramienta cuenta con dos ventanas, las cuales se describen en las Figuras 6.1 y 6.2.

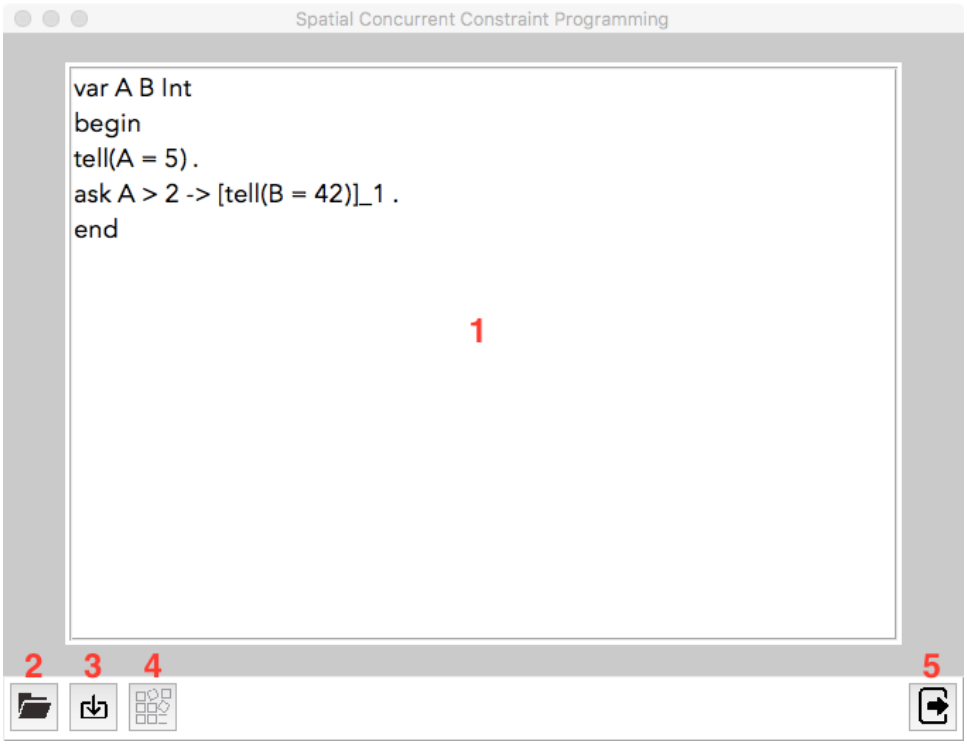


FIGURA 6.1. Ventana principal

La Figura 6.1 muestra la ventana principal de la herramienta, en la cual el programador ingresa el programa o sistema a ejecutar en el lenguaje de programación propuesto en el Capítulo 5. Esta ventana cuenta con un editor de texto y una barra inferior con cuatro botónes. El editor de texto, identificado con el número **1**, permite copiar, cortar, pegar y desplazamiento vertical, actualmente no tiene implementado funciones de deshacer y rehacer. La funcionalidad de los botónes se explica a continuación:

- El botón **2** permite abrir documentos de texto cuya extensión sea *txt*, *maude* o *in*, por medio de una ventana de manejo de archivos. Cuando se selecciona el archivo el contenido se muestra automáticamente en el editor de texto.

- El botón **3** permite guardar en un archivo de texto el contenido actual del editor de texto, por medio de una ventana de manejo de archivos.
- El botón **4** ejecuta el programa escrito en el editor de texto. En caso de que la sintaxis sea incorrecta se muestra un mensaje de error en la ventana de resultados.
- El botón **5** permite salir de la herramienta.

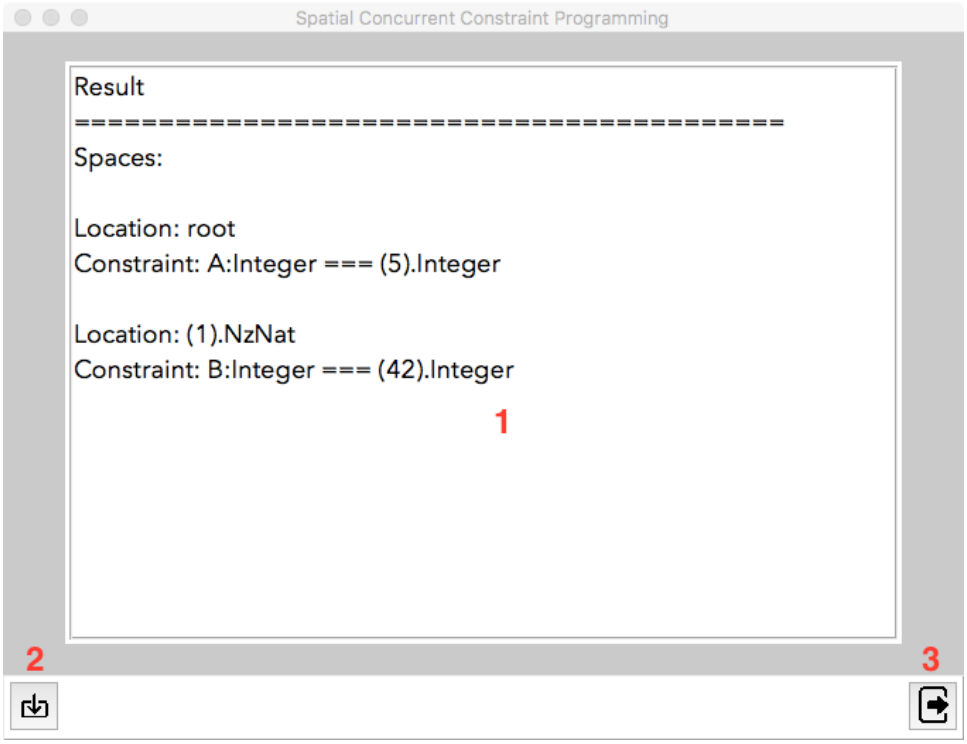


FIGURA 6.2. Ventana de resultados

La Figura 6.2 muestra la ventana de respuestas, en la cual se informa al programador acerca de la evolución del sistema ingresado. Esta ventana cuenta con un cuadro de texto y una barra inferior con dos botónes. En el cuadro de texto identificado con el número **1** se muestra el resultado de la ejecución del sistema; en caso de que haya un error en el sistema escrito por el programador se muestra un mensaje correspondiente. El cuadro de texto no permite modificar o seleccionar el texto. Los botónes **2** y **3** permiten guardar el resultado en un archivo de texto, además de cerrar la ventana de resultados.

En la ventana de respuestas se presenta el estado final del sistema en dos secciones: los agentes existentes y sus respectivos bancos de información, y los procesos que no se pudieron ejecutar.

Las dos ventanas cuentan con un menú desplegable con las mismas funcionalidades explicadas en esta sección.

6.4. Ejemplo

Este capítulo concluye con algunos ejemplos del funcionamiento de la herramienta. Utilizando la herramienta propuesta en este capítulo se utiliza el ejemplo trabajado en las secciones 4.5 y 5.4 para verificar que se llega al estado final deseado a partir del código escrito en la Sección 5.4. En la Figura 6.3 se muestra el estado incial d junto con el proceso S, tal y como se definió en la Sección 3.6.

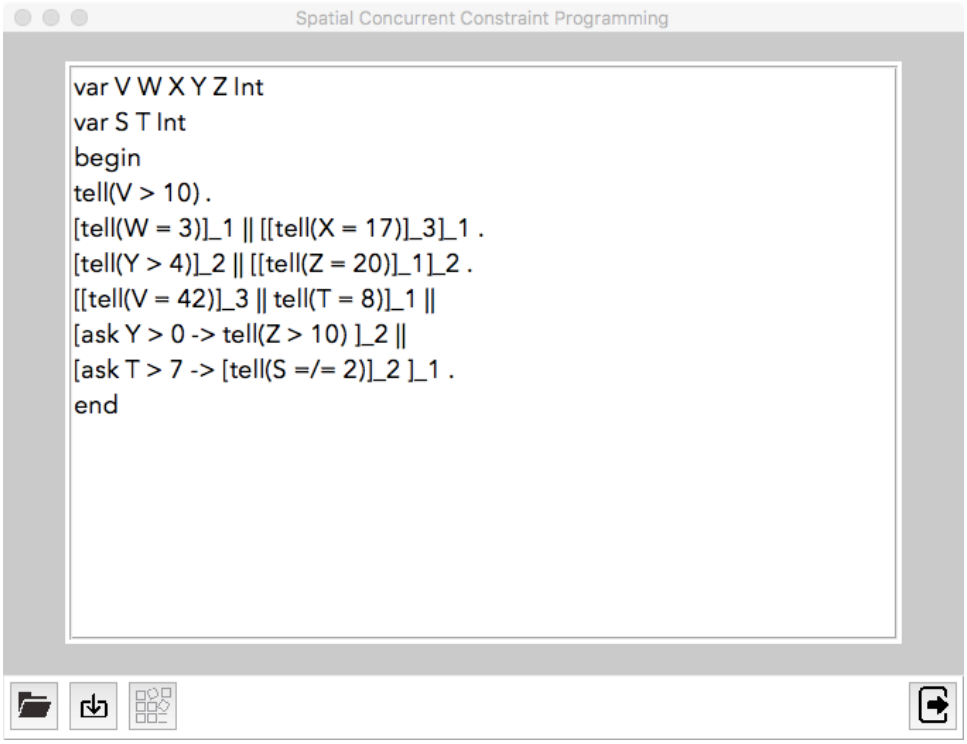
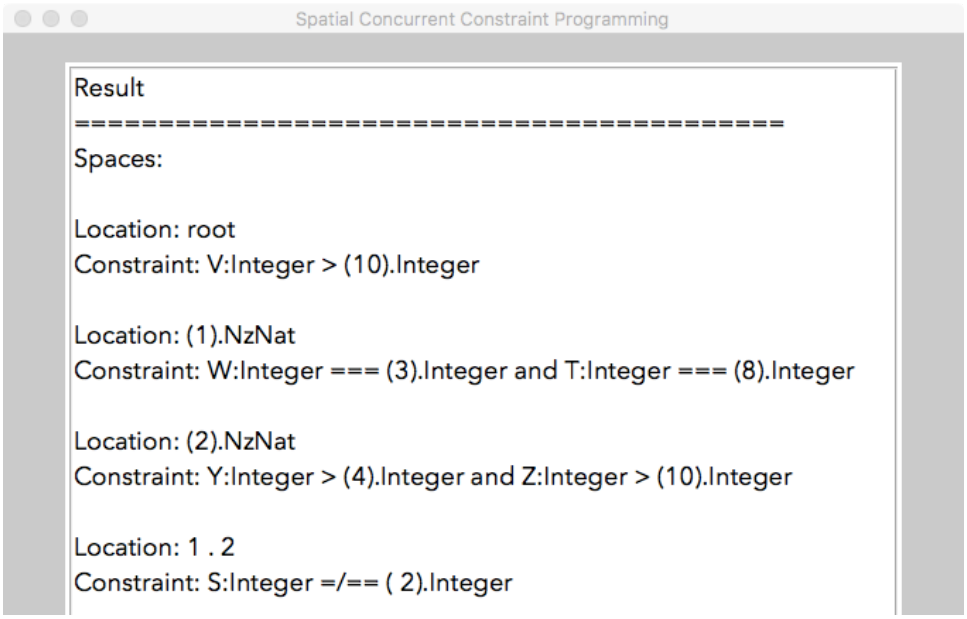


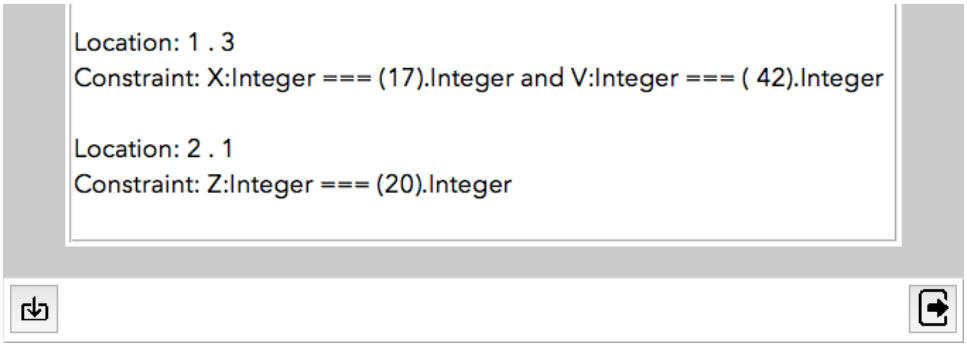
FIGURA 6.3. Estado inicial d del sistema

La Figura 6.4 muestra el estado final del sistema obtenido por la simulación en Maude. En esta caso el resultado concuerda con el estado final esperado descrito en la Figura 3.4. En el resultado, los agentes (*spaces*) son listados con su identificador (*location*) y su banco de información (*constraint*).

Considere un sistema en el cual no se puede ejecutar un proceso **ask** debido a que la restricción no es satisfacible por el banco de información del agente. En el siguiente ejemplo existen dos procesos que no



(A)



(B)

FIGURA 6.4. Estado final del sistema

se pueden ejecutar, por lo tanto el sistema no evoluciona y el resultado debe coincidir con su estado inicial. El estado inicial del sistema se describe a continuación:

```
var A B C D Int
begin
tell(A =/= 10) .
ask A = 10 -> tell(B >= 5) .
[tell(C > 5) || ask C > 6 -> tell(D = 8)]_1 .
end
```

En la Figura 6.5 se observa que existen dos procesos que no fueron ejecutados: el primero de ellos debido a que `A = 10` no es satisfacible a partir de `A =/= 10`, y el segundo de ellos porque `C >6` no es satisfacible a partir de `C >5`.

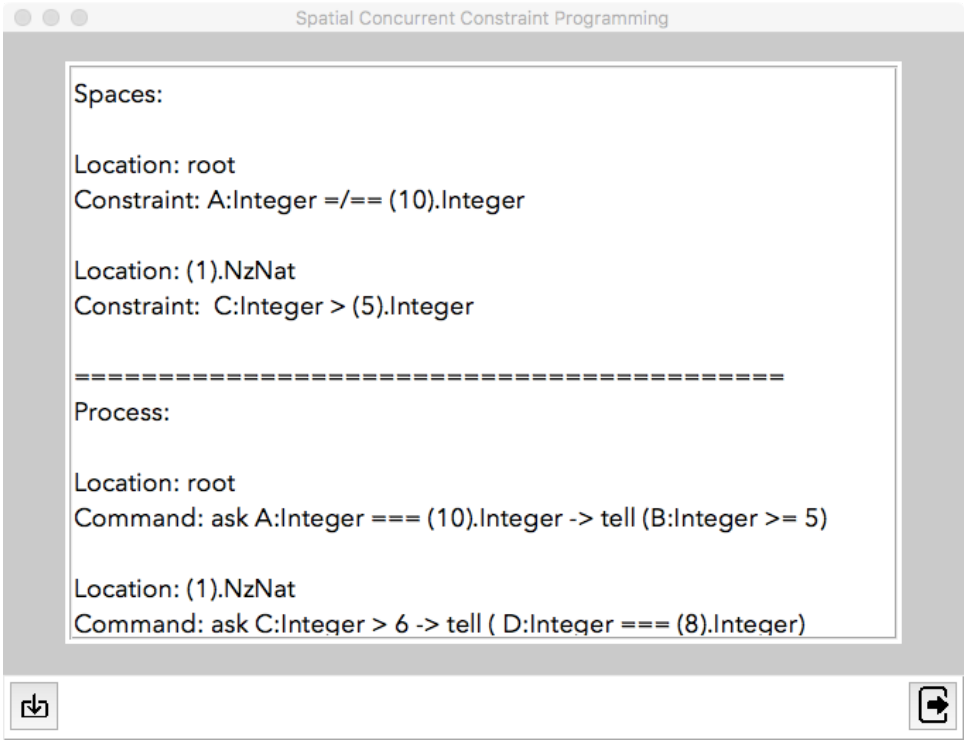


FIGURA 6.5. Estado final del sistema con procesos no ejecutados

Finalmente, si el programador no define una variable usada dentro del sistema se genera un mensaje de error, debido a que el lexer no sabe como manejar el token y por lo tanto no se genera la expresión correspondiente. El mensaje

```
Sorry, there is an error!.
Check your specification and try again."
```

aparece en la ventana de resultados.

Capítulo 7

Conclusión

La lógica de reescritura es una lógica para especificar sistemas concurrentes la cual, al ser combinada con Maude y sus herramientas, facilita no solo la especificación formal de sistemas sino también la verificación mecánica de las propiedades que debe satisfacer el sistema.

En este documento se presenta una semántica en lógica de reescritura para SCCP la cual es ejecutable en Maude. La semántica ejecutable en lógica de reescritura para SCCP descrita en este documento busca brindarle al programador un ambiente para análizar sistemas distribuidos.

El módulo de reescritura SMT soluciona los problemas relacionados con la ejecución simbólica y la combinación entre reescritura de términos con la solución de problemas de reestricciones.

Como trabajo futuro se puede adaptar la semántica en lógica de reescritura para variaciones del modelo SCCP, en donde se considere la existencia de mentiras transmitidas intencionalmente por agentes en el sistema. Extender la herramienta para brindarle al programador una traza de la evolución del sistema, con la que pueda observar paso a paso la ejecución de procesos y modificación de los bancos de información. Es importante extender la herramienta incluyendo operaciones como extrusión. Finalmente, se sugiere agregar más elementos al lenguaje de programación, en los que se incluya la declaración de procesos y especificación de agentes a priori.

Apéndice A

Especificación formal en Maude

Este anexo incluye los módulos funcionales `SMT-UTIL`, `AGENT-ID`, `SCCP-SYNTAX` y `SCCP-STATE` y el módulos de sistema: `SCCP`.

A.1. SMT-UTIL

```
load smt.maude

fmod SMT-UTIL is
  pr META-LEVEL .
  inc INTEGER .
  pr CONVERSION .
  op check-sat : Boolean -> Bool .
  op check-unsat : Boolean -> Bool .
  eq check-sat(B:Boolean)
   = metaCheck(['INTEGER], upTerm(B:Boolean)) .
  eq check-unsat(B:Boolean)
   = not(check-sat(B:Boolean)) .

  --- some Boolean identities
  eq B:Boolean and true
   = B:Boolean .
  eq B:Boolean and false
   = false .
  eq B:Boolean or true
   = true .
  eq B:Boolean or false
   = false .
  eq true and B:Boolean
   = B:Boolean .
  eq false and B:Boolean
   = false .
  eq true or B:Boolean
   = true .
```

```
  eq false or B:Boolean
   = false .
  eq not((true).Boolean)
   = (false).Boolean .
  eq not((false).Boolean)
   = (true).Boolean .
endfm
```

A.2. AGENT-ID

```
fmod AGENT-ID is
  sort Aid .
  pr NAT .
  subsort Nat < Aid .

  op root : -> Aid .
  op _._ : Aid Aid -> Aid [assoc left id: root] .
endfm
```

A.3. SCCP-SYNTAX

```
fmod SCCP-SYNTAX is
  pr INTEGER .
  pr AGENT-ID .
  sort SCCPCmd .

  op 0 : -> SCCPCmd .
  op tell_ : Boolean -> SCCPCmd .
  op ask_->_ : Boolean SCCPCmd -> SCCPCmd .
  op ask<_>_->_ : Aid Boolean SCCPCmd -> SCCPCmd .
  op _||_ : SCCPCmd SCCPCmd -> SCCPCmd [assoc gather (e E) ] .
  op <_>[_] : Aid SCCPCmd -> SCCPCmd .
endfm
```

A.4. SCCP-STATE

```
fmod SCCP-STATE is
  pr SCCP-SYNTAX .
  pr SMT-UTIL .

  sorts Cid Obj Cnf Sys .
  subsorts Obj < Cnf .
  ops store process : -> Cid .
```

```
  op [_,_,_] : Cid Aid Boolean -> Obj [ctor] .
  op [_,_,_] : Cid Aid SCCPCmd -> Obj [ctor] .
  op mt : -> Cnf [ctor] .
  op __ : Cnf Cnf -> Cnf [ctor assoc comm id: mt] .
  op {_} : Cnf -> Sys [ctor] .

  vars L L0    : Aid .
  vars C C0    : Boolean .
  vars X       : Cnf .

  --- auxiliary operations
  op exists-store? : Cnf Aid -> Bool .
  eq exists-store?(mt, L)
   = false .
  eq exists-store?( [ process, L0, C0 ] X, L)
   = exists-store?(X,L) .
  eq exists-store?( [ store, L0, C0 ] X, L)
   = (L0 == L) or-else exists-store?(X,L) .
endfm
```

A.5. SCCP

```
mod SCCP is
  pr SCCP-STATE .

  vars L L0 L1 : Aid .
  vars B B0 B1 : Boolean .
  vars C C0 C1 : SCCPCmd .
  vars X : Cnf .

  --- non-observable concurrent transitions
  eq [ process, L, 0 ]
   = mt .
  eq [ store, L, B0 ] [ store, L, B1 ]
   = [ store, L, B0 and B1 ] .

  --- observable concurrent transitions
  rl [tell] :
     { [ store, L, B] [process, L, tell B0 ] X }
  => { [ process, L, 0 ] [ store, L, B and B0] X } .

 crl [ask] :
```

```
    { [ store, L, B ] [ process, L, ask B0  -> C0 ] X }
 => { [ store, L, B ] [ process, L, C0 ] X }
 if check-unsat(B and not(B0)) .

crl [ask-ext0] :
    { [ store, L0 . L1, B ] [ process, L0, ask< L1 > B0  -> C0 ] X }
 => { [ store, L0 . L1, B ] [ process, L0, C0 ] X }
 if check-unsat(B and not(B0)) .

crl [ext-ask1] :
    { [ process, L0, ask< L1 > B0  -> C0 ] X }
 => { [ process, L0, C0 ] X }
 if exists-store?(X, L0 . L1) == false
 /\ check-unsat(not(B0)) .

 rl [parallel] :
    { [ process, L, C0 || C1 ] X }
 => { [ process, L, 0 ] [ process, L, C0 ] [ process, L, C1 ] X } .

 rl [space] :
    { [ process, L0, < L1 >[ C0 ] ] X }
 => { [ process, L0, 0 ] [ process, L0 . L1, C0] [ store, L0 . L1, true ] X } .

endm
```

Bibliografía

[1] R. Bruni and J. Meseguer. Semantic foundations for generalized rewrite theories. *Theoretical Computer Science*, 360(1-3):386–414, 2006.

[2] S. Burris and H. Sankappanavar. *A course in universal algebra*. Graduate texts in mathematics. Springer-Verlag, 1981.

[3] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. L. Talcott, editors. *All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic*, volume 4350 of *Lecture Notes in Computer Science*. Springer, 2007.

[4] F. S. de Boer, A. D. Pierro, and C. Palamidessi. Nondeterminism and infinite computations in constraint programming. *Theoretical Computer Science*, 151(1):37 – 78, 1995.

[5] L. de Moura, B. Dutertre, and N. Shankar. *A Tutorial on Satisfiability Modulo Theories*, pages 20–36. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007.

[6] F. Durán and J. Meseguer. On the church-rosser and coherence properties of conditional order-sorted rewrite theories. *The Journal of Logic and Algebraic Programming*, 81(7-8):816–850, 2012. Rewriting Logic and its Applications.

[7] Ebnf: A notation to describe syntax.

[8] S. Knight, C. Palamidessi, P. Panangaden, and F. D. Valencia. Spatial and Epistemic Modalities in Constraint-Based Process Calculi. In M. Koutny and I. Ulidowski, editors, *CONCUR 2012 - 23rd International Conference on Concurrency Theory*, volume 7454 of *Lecture Notes in Computer Science*, pages 317–332, Newcastle upon Tyne, United Kingdom, Sept. 2012. Springer.

[9] D. Kroening and O. Strichman. *Decision Procedures: An Algorithmic Point of View*. Springer Publishing Company, Incorporated, 1 edition, 2008.

[10] S. Lucas and J. Meseguer. Operational termination of membership equational programs: the order-sorted way. *Electronic Notes in Theoretical Computer Science*, 238(3):207 – 225, 2009. Proceedings of the Seventh International Workshop on Rewriting Logic and its Applications (WRLA 2008).

[11] J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73 – 155, 1992. Selected Papers of the 2nd Workshop on Concurrency and Compositionality.

[12] J. Meseguer. Membership algebra as a logical framework for equational specification. In F. Presicce, editor, *Recent Trends in Algebraic Development Techniques*, volume 1376 of *Lecture Notes in Computer Science*, pages 18–61. Springer Berlin Heidelberg, 1998.

[13] M. Nielsen, C. Palamidessi, and F. D. Valencia. Temporal concurrent constraint programming: Denotation, logic and applications. *Nordic Journal of Computing*, 9(2):145–188, June 2002.

[14] T. Parr. *The Definitive ANTLR 4 Reference*. Pragmatic Bookshelf, 2nd edition, 2013.

[15] C. Rocha, J. Meseguer, and C. Muñoz. Rewriting modulo SMT and open system analysis. *Journal of Logical and Algebraic Methods in Programming*, 86(1):269 – 297, 2017.

[16] V. A. Saraswat. *Concurrent Constraint Programming*. MIT Press, 1993.

[17] V. A. Saraswat and M. Rinard. Concurrent constraint programming. In *17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 232–245. ACM, 1990.

[18] V. A. Saraswat, M. Rinard, and P. Panangaden. The semantic foundations of concurrent constraint programming. In *18th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 333–352. ACM, 1991.