Universidad Nacional de Rosario

Facultad de Ciencias Exactas, Ingeniería y Agrimensura

SAT-Solver

Un evaluador para modelos de lógica temporal

Autor:
Natalia Mellino

Diciembre 2021

Contenidos

1 Motivación

1.1 ¿Por qué un DSL para Lógica Temporal?

La principal motivación de la realización de este proyecto, no es más ni menos que poder proveer una herramienta que pueda ser útil al momento de aprender y poner en práctica los conocimientos adquiridos de la teoría de Lógica Temporal.

Al momento de realizar ejercicios o querer hacer alguna verificación sobre estos modelos lógicos, nos encontramos con que no siempre podemos realizar estas tareas de manera rápida y simple como deseamos. Es por ello, que mi intención es poder facilitar en la medida que sea posible a cualquier persona que se encuentre en alguna de las situaciones mencionadas anteriormente.

2 Introducción

2.1 Un repaso de CTL

El objetivo de esta sección no es dar una explicación exhaustiva de la teoría de lógica temporal, sino más bien, refrescar los conceptos que sean necesarios para un mayor entendimiento de cómo funciona este proyecto. Se asume que el lector posee un cierto entendimiento de los conceptos básicos de Lógica Temporal.

Si bien en la Lógica Temporal hay muchas ramas y tópicos, en este proyecto nos centramos en lo que se conoce como CTL, o más bien, Computation Tree Logic. Este es un tipo de lógica temporal en el que el se modela el tiempo como una estructura de árbol en donde el futuro no está determinado, sino que hay distintos caminos en el futuro donde cualquiera de ellos puede ser el camino que se vaya a realizar [1].

2.1.1 Sintaxis y Semántica

La **sintaxis** de CTL se puede definir de forma inductiva de la siguiente manera [2]:

$$\phi ::= \top \mid \bot \mid p \mid \neg\phi \mid \phi \wedge \psi \mid \phi \vee \psi \mid \phi \rightarrow \psi \mid \forall \bigcirc \phi \mid \exists \bigcirc \phi \mid$$
$$\forall[\phi \cup \psi] \mid \exists[\phi \cup \psi] \mid \forall\Box\phi \mid \exists\Box\phi \mid \forall\Diamond\phi \mid \exists\Diamond\phi$$

Un **Modelo** o **Sistema de transiciones** está formado por una tupla: $(S, \rightarrow, I, L)$ donde:

- S es un conjunto finito de estados.
- I es un conjunto de estados iniciales ($I \subseteq S$).
- $\rightarrow \subseteq S \times S$ es una relación de transición entre estados
- $L : S \rightarrow \mathcal{P}(AT)$ una función de etiquetado, donde a cada estado se le asigna un conjunto de proposiciones atómicas.

Ahora recordemos cómo era la **semántica** de estos operadores. Definimos la relación $\vDash$ por inducción en ϕ:

- $M, s \nvDash \bot$
- $M, s \vDash p_i \iff p_i \in L(s)$
- $M, s \vDash \neg\phi \iff M, s \nvDash \phi$

- $M, s \vDash \phi \wedge \psi \iff M, s \vDash \phi$ y $M, s \vDash \psi$
- $M, s \vDash \forall \bigcirc \phi \iff$ para todo s' tal que $s \to s'$ se cumple que $M, s \vDash \phi$.
- $M, s \vDash \exists \bigcirc \phi \iff$ para algún s' tal que $s \to s'$ se cumple que $M, s \vDash \phi$.
- $M, s \vDash \forall[\phi \cup \psi] \iff$ para cada traza $s_0 \to s_1 \to ...$ con $s_0 = s$ existe $j \in \mathbb{N}$ tal que:
  - $M, s_j \vDash \psi$
  - $M, s_i \vDash \phi$ para todo $i < j$
- $M, s \vDash \exists[\phi \cup \psi] \iff$ para alguna traza $s_0 \to s_1 \to ...$ con $s_0 = s$ existe $j \in \mathbb{N}$ tal que:
  - $M, s_j \vDash \psi$
  - $M, s_i \vDash \phi$ para todo $i < j$

Luego tenemos los **operadores derivados**:

- $\forall\Diamond\phi = \forall[\top \cup \phi]$
- $\exists\Diamond\phi = \exists[\top \cup \phi]$
- $\forall\Box\phi = \neg\exists\Diamond\neg\phi$
- $\exists\Box\phi = \neg\forall\Diamond\neg\phi$

Para los operadores $\to, \top$ se omitió su semántica ya que las mismas se derivan a partir de los otros operadores ya existentes. Esto en realidad vale también para varios de los operadores que definimos, en las próximas secciones veremos como esto nos sirve para facilitar la implementación del evaluador.

2.1.2 El Algoritmo SAT para *model-checking*

Recordemos rápidamente de qué se trataba el algoritmo SAT: dado un modelo y una fórmula CTL, devuelve el conjunto de estados del modelo que satisface dicha fórmula [2]. Esto es, en esencia, el objetivo de este proyecto. Para una fórmula y un modelo, nuestro algoritmo se comporta de la siguiente manera:

- $sat(\bot) = \emptyset$
- $sat(p_i) = \{s \in S \mid P_i \in L(s)\}$
- $sat(\neg\phi) = S - sat(\phi)$
- $sat(\phi \wedge \psi) = sat(\phi) \cap sat(\psi)$
- $sat(\phi \vee \psi) = sat(\phi) \cup sat(\psi)$
- $sat(\exists \bigcirc \phi) = pre_{\exists}(\phi)$
- $sat(\forall \bigcirc \phi) = pre_{\forall}(\phi)$
- $sat(\exists[\phi \cup \psi]) = exUntil(sat(\phi), sat(\psi))$
- $sat(\forall[\phi \cup \psi]) = forallUntil(sat(\phi), sat(\psi))$
- $sat(\forall\Diamond\phi) = inev(sat(\phi))$
- $sat(\exists\Diamond\phi) = exUntil(S, sat(\phi))$

Donde:

$preExists(Y) = \{s \in S \mid \exists s' : s \to s' \wedge s' \in Y\}$
$preAll(Y) = \{s \in S \mid \forall s' : s \to s', s' \in Y\}$

```
exUntil(X, Y):
  while (Y != Y u (X n preExists(Y))) do:
    Y <- Y u (X n preExists(Y))
  return Y

forallUntil(X, Y):
  while (Y != Y u (X n preAll(Y))) do:
    Y <- Y u (X n preAll(Y))
  return Y

inev(Y):
  while (Y != Y u preAll(Y)) do:
    Y <- Y u preAll(Y)
  return Y
```

3 Gramática

3.1 Sintaxis Concreta

Definimos la sintaxis concreta del DSL a partir de la siguiente gramática:

```
fields ::= States '=' sts ';'
           Relations '=' rels ';'
           Valuations '=' vals ';'
           CTLExp '=' ctl ';'

vals ::= '{' '}' | '{' valuations '}'

valuations ::= valuation | valuation ',' valuations

valuation ::= AT ':' sts

sts ::= '[' states ']' | '[' ']'

states ::= State | State ',' states

rels ::= '[' ']' | '[' relations ']'

relations ::= relation | relation ',' relations

relation ::= '(' State ',' State ')'

ctl ::= AT
      | BT
      | TOP
      | '!' ctl
      | ctl '&' ctl
```

```
| ctl '|' ctl
| ctl '->' ctl
| AX ctl
| EX ctl
| AU ctl ctl
| EU ctl ctl
| AF ctl
| EF ctl
| AG ctl
| EG ctl
| '(' ctl ')'
```

Para las fórmulas CTL, el **orden de precedencia** utilizado es el siguiente, comenzando desde los que tienen menos precedencia hasta los que más tienen. Dos o más operadores en el mismo item indican que la precedencia es la misma:

- Asocian a izquierda: $\wedge(\&), \vee(|)$
- Asocian a izquierda: $\rightarrow$ (->), $\forall \cup (AU), \exists \cup (EU)$
- !, $\forall \bigcirc (AX), \exists \bigcirc (EX), \forall \Diamond (AF), \exists \Diamond (EF), \forall \Box (AG), \exists \Box (EG)$

Entonces por ejemplo: $p \wedge q \rightarrow r$ se asocia: $p \wedge (q \rightarrow r)$ ya que $\rightarrow$ tiene más precedencia que $\wedge$.

En los casos de los operadores que tienen la misma precedencia, se utiliza la asociación a izquierda, esto significa que si escribimos: $p \wedge q \vee r$, este se asociará a la izquierda: $(p \wedge q) \vee r$.

Observación: siempre es posible hacer uso de los parentesis para explicitar el orden deseado de los operadores en una determinada fórmula.

3.2 Notación

Para más simplicidad al momento de parsear el modelo, cada símblolo se representa de manera distinta en el código. A modo de referencia, se puede consultar de forma rápida la siguiente tabla para entender qué significa la notación utilizada a lo largo de todo el programa.

Operador	Traducción
$\top$	TOP
$\bot$	BT
$\neg\phi$	$!\phi$
$\phi \wedge \psi$	ϕ & ψ
$\phi \vee \psi$	$\phi \mid \psi$
$\phi \rightarrow \psi$	ϕ -> ψ
$\forall \bigcirc \phi$	AX ϕ
$\exists \bigcirc \phi$	EX ϕ
$\forall[\phi \cup \psi]$	AU ϕ
$\exists[\phi \cup \psi]$	EU ϕ
$\forall\Box\phi$	AG ϕ
$\exists\Box\phi$	EG ϕ
$\forall\Diamond\phi$	AF ϕ
$\exists\Diamond\phi$	EF ϕ

4 Resolución

4.1 Parseo

El modelo, junto con la fórmula a evaluar, es pasado en un archivo con extensión `.sat` en el formato que explicita la sintaxis concreta presentada en la sección anterior. Además, en la carpeta `test` se pueden encontrar algunos tests a modo de ejemplo.

Se hizo uso de la herramienta *Happy*, un generador de parsers para Haskell, que dada una gramática específica, nos permite generar un archivo Haskell con nuestro parser. Esta herramienta fue muy útil ya que nos permitió de manera simple, especificar la asociatividad y precedencia de los operadores. Además de que usando Happy, no tenemos que preocuparnos por la recursión a izquierda, lo cual facilitó mucho el trabajo.

Podemos encontrar la implementación del parser en el archivo `src/Parse.y`

4.2 Uso de las Mónadas

Si vamos al archivo `src/Monads.hs` podemos ver que se hizo uso de la **mónada State** para llevar el modelo (estados, transiciones, valuaciones y fórmula) como estado global, ya que esto nos permitió implementar el evaluador de forma más simple.

4.3 Evaluación

El evaluador es la parte principal del proyecto, es el que se encarga de implementar y calcular nuestro algoritmo SAT. En `src/Eval.hs` podemos ver su implementación utilizando mónadas. La misma no difiere mucho de la explicación del algoritmo presentada al principio del informe, sigue exactamente la misma idea para implementar el algoritmo y las funciones auxiliares.

4.4 Otros Archivos

- En `src/CTL.hs` se definen las estucturas de datos utilizadas para representar nuestro modelo. Si recordamos nuestras clases de lógica, podemos advertir que la gramática para fórmulas CTL se puede simplificar bastante en el código utilizando el concepto de **conjuntos completos de conectivos**. Entonces, toda la gramática que representa a las fórmulas CTL, contiene **azucar sintáctico**. Al momento de parsear la fórmula en el programa, utilizamos los *patterns* de Haskell, que nos permiten representar a términos compuestos por otros. Por ejemplo, si parseamos la fórmula: $p \to q$ en nuestro programa, la misma será tratada 'internamente' como $\neg p \vee q$ y la evaluación se realizará sobre esta última fórmula, que es equivalente a la primera, permitiendonos así reducir la cantidad de términos distintos a tener en cuenta al momento de implementar funciones en Haskell.

- En `src/PPrint.hs` se implementó un simple pretty printer para mostrar los resultados en pantalla de manera más elegante.

- En `app/Main.hs` simplemente se encuentran las funciones que se encargan de:

  - Leer y parsear el archivo de entrada que contiene el modelo.
  - Llamar al evaluador con el modelo dado.
  - Tomar el resultado devuelto y mostrarlo en pantalla utilizando las funciones de `src/PPrint.hs`.

  En el main, también se realiza cierto chequeo sobre el modelo, para advertir al usuario sobre los errores que se puedan encontrar.

5 Instalación y Uso

Para comenzar a usar el evaluador basta con lo siguiente:

- Clonar el repositorio:
  `$ git clone https://github.com/natimellino/SAT-Solver.git`
- Una vez dentro del repositorio, correr:
  `$ stack setup`
- Para compilar corremos:
  `$ stack build`
- Finalmente, para ejecutar nuestro evaluador con un modelo:
  `$ stack exec TP-Final-exe "/path/to/file.sat"`

6 Trabajo a Futuro

- Implementar un mejor manejo de errores en el parser, que nos permita identificar con más precisión cuáles son los errores de sintaxis en el archivo de entrada.
- Hacer una versión interactiva del programa, esta nos podría permitir modificar el modelo pasado como argumento sin tener que volver a ejecutar el programa. Esto se realizaría agregando funciones que permitan modificar el estado de nuestra mónada que es el que lleva el modelo.
- Una vez que se tenga una versión interactiva del programa se pueden agregar más comandos que el de simplemente evaluar un modelo, podría ser por ejemplo imprimir en pantalla la fórmula o el modelo (mejorando nuestro pretty printer para mostrar resultados más elegantes); y también proveer los comandos que nos permiten modificar nuestro modelo de entrada, o bien, pasar un modelo nuevo, haciendo posible lo mencionado en el apartado anterior.

Referencias

[1] Michael Huth, Mark Ryan - *Logic in computer science modelling and reasoning about systems*, Cambridge University Press (2004).

[2] Diapositivas de teoría de *Lógica Temporal* de Dante Zanarini, FCEIA (año 2019).