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

Paralloy: Learning y Restart en un Sat Solver distribuido

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

Matías Hernando Pérez

Director: Nicolas Rosner

Codirector: Ignacio Vissani

Buenos Aires, 24 de julio de 2016

Paralloy: Learning y Restart en un Sat Solver distribuido

El problema de satisfacibilidad, también conocido como problema *SAT*, es uno de los problemas con mayor estudio dentro de las Ciencias de la Computación. En el año 1971, Stephen Cook definió lo que hoy llamamos NP-completitud en términos de este problema, que se convirtió en el primer problema NP-completo. En la actualidad tiene una amplia variedad de aplicaciones prácticas, entre ellas la verificación automática de software y de hardware. Dada la importancia de este problema, existe un gran interés en el desarrollo de piezas de software que sean capaces de resolverlo de forma eficiente, los *SAT-solver*.

Paralloy es un *SAT-solver* que, mediante un esquema de *Divide & Conquer*, realiza la resolución de forma paralela y distribuida. En este trabajo se analizan los efectos de agregar un esquema de *learning* y *restart distribuido* a la herramienta, ya que al realizar la división en sub-problemas independientes la información que estos generan queda aislada, haciéndola visible sólo localmente. Asimismo, los *restarts* que se generan en cada nodo también tienen un efecto exclusivamente local. Este trabajo intenta romper con este aislamiento de cada una de las resoluciones de los sub-problemas.

Luego de evaluar distintas estrategias de distribución de la información entre las unidades de cómputo del sistema (*learning*), se logró una reducción del 30 % en el tiempo de ejecución total promedio para los problemas considerados. En algunos casos la mejora alcanzada fue de 50× con respecto a la velocidad original de la herramienta. De esta forma se logra expandir el alcance de la herramienta, posibilitando la resolución de un nuevo conjunto de problemas en tiempos tolerables.

Si bien la estrategia de *restart* global implementada no logró reducir el tiempo de ejecución promedio, sí se observó una disminución en el tiempo de ejecución a lo largo de las distintas iteraciones de *restart*. Este fenómeno promisorio amerita ser investigado, y abre un posible camino para trabajo futuro.

Palabras claves: Sat-Solving, Paralelo, Distribuido, Learning, Restart, Paralloy.

Paralloy: Learning and Restart for a distributed Sat Solver

The satisfiability problem, also known as *SAT* problem, is among the most studied in Computer Science. In 1971, Stephen Cook defined what we currently call NP-completeness in terms of it, making it the first NP-complete problem. There are many practical applications of this problem, including automatic verification of software and hardware. Given the importance of this problem, there is broad interest in developing software tools that are capable of tackling it in a highly efficient way, known as *SAT-solvers*.

Paralloy is a *SAT-solver* that uses a *Divide & Conquer* approach in order to parallelise and distribute the solving effort. In this thesis we study the effects of adding *learning* and *global restarts* to the tool. The motivation to do this arises from the fact that, when a problem is divided into independent *sub-problems*, the information learned by the solver is isolated, limiting its usefulness to the local context. In the same way, the *restarts* performed at each node do not have a global impact. This work aims at breaking this isolation and strengthening the interaction between nodes of the system.

After analysing various strategies for the distribution of information between system nodes, we observed reductions of up to 30 % in the average total execution time for the problems in our benchmark. In some cases the improvement reached a factor of *50x* with respect to the original execution time. Thus, the techniques incorporated in this work were able to expand the proving power of the tool, enabling it to solve more complex and time-consuming problems.

Although the *global restart* strategy implemented in this work did not achieve a reduction in mean execution time, we did observe decreasing execution times across succesive iterations of *restart*. This promising result deserves further exploration and opens a path for future research.

Keywords: Sat-Solving, Parallel, Distributed, Learning, Restart, Paralloy.

AGRADECIMIENTOS

La verdad, hay tanta gente para agredecer que este trabajo haya visto la luz que me resulta difícil elegir a alguien para empezar. Creo que fue realmente fundamental contar con todos, en mayor o menor medida, cada uno a los que le dedico la siguientes lineas hicieron posible que estas se escriban.

Voy a empezar por mis papás, ellos me apoyaron siempre, no dudaron ni medio segundo cuando decidí cambiarme de carrera para empezar computación, simplemente me dijeron "bueno, si es lo que querés, nos parece bien". Tener ese tipo de respaldo hace que todo sea mucho más fácil. Siempre me dieron ánimo luego de cada tropiezo. Desde la primaria, secundaria, catequesis, CBC, cada vez que algo no me salía siempre estaban apoyándome creyendo que la próxima vez me iba a salir. Así fueran 2, 3, 4 o 10 los intentos que me llevara, siempre conté con su confianza y su ayuda si la necesitaba. Saber que después de cursar hasta las 10 de la noche y llegar a casa a las 11, me iba a estar esperando una cena, hace la diferencia. Ellos fueron los que soportaron mis cambios de humor los días previos a los exámenes, que yo sinceramente no sé si hubiera sido capaz de soportar. Me hubiera gustado que mi papá estuviera acá para compartir este momento con él, pero así como sé que él nunca dudó que lo iba a lograr, también sé que de alguna forma él está.

También quiero agradecer a mis hermanos, voy a empezar con Fede por que es el menor y el que capaz no se dio cuenta de la gran contribución que hizo. Fede: de verdad gracias, te juro que no hay forma más efectiva para desestrezarme que hablar con vos, no fuiste el que me enseñó nada académico (aunque ahora ya sí lo hagas), pero fuiste mi compañero designado cada vez que necesité pensar en otra cosa. Si hay una regla de oro que seguí durante toda mi carrera, es que *el día del parcial no se estudia* y para eso siempre supe que te podía buscar y podíamos hacer algo juntos. Por más que al día siguiente tuviera que presentar mi tesis, no importa, yo estaba tranquilo porque estábamos boludeando al ritmo de "Qué e' lo que pasa?!", y eso fue fundamental.

Danielita *muy bien*, durante la primaria y secundaria siempre fuiste el oído a buscar cuando tenía algún problema con un compañero o cuando algo de literatura no salía. La que siempre tenía algún truco para memorizar la capital del país que no se queda. Y cuando fui creciendo no perdió eso, ahora ya no eran problemas de niños de secundario, pero situaciones un toque más complejas. Ya no era memorizar una capital sino aprobar sociedad y estado. Verla pasar a ella primero por el mismo proceso de hacer y presentar una tesis, si bien me dio pánico, también en algún punto me demostró que era posible. Siempre admiré tu fuerza de voluntad y tenacidad para hacer lo que tenías que hacer, creo que nunca pude igualar eso, pero tratando de hacerlo, fui avanzando.

A Santi y Male, que siempre estuvieron pendientes de cómo iba y me dieron su apoyo y ánimo sin presionarme. A veces son esas simples palabras de "está todo bien" las que hacen más fácil el camino. Cuando un fin de semana no escribía, sino que me juntaba con mis amigos, me ayudaban a sacarme la culpa y hacer más obvio lo real, que es que está bien, que se puede igual, no importa qué. Gracias chicos.

El barilocho, Diego, mi hermano mayor. Creo que fue él el que se ocupó de despertar mi curiosidad por las ciencias. Fue el que me hizo programar por primera vez en turbo pascal. Que siempre que algo no se entendía, sabía que le podía pedir ayuda a Diego para entenderlo, a veces era que me lo explique y otras ver juntos qué era de lo que estaban

hablando. Como buen hermano mayor fue siempre un punto de apoyo y el arma de último recurso, la confiable, la que no falla.

Y hablando de entender de qué hablan los textos o porqué vale un paso de una demostración, nadie mejor que Rodirgo. Con él aprendí a estudiar, aprendí a seguir indagando hasta que todo cierre por todos lados, no dejar un problema tranquilo hasta entender realmente de qué se trata. A cuestionar todo y nunca dejar nada por la mitad. Si no se entiende bien, hay que entenderlo bien. Desde el primer trabajo de Algoritmos I hasta los slides de la presentación de mi tesis, estuvimos siempre acompañándonos tratando de entender. Creo que tengo suficientes pruebas para decir que Probabilidad y Estadística nunca la hubiera entendido si no fuera porque la cursamos juntos. O definitivamente nunca al nivel que terminé entendiéndola. Agradezco habernos cruzado tan rápido, porque si bien no fue suficiente, fue necesario. Y este trabajo, es en gran porcentaje posible gracias a vos.

También quiero aprovechar para agradecer a los profesores y ayudantes de la Facultad. Algunos con clases más dinámicas y divertidas, otros con más rigurosidad. Pero todos definitivamente con muchas ganas. Si hay algo que se ve en el departamento es las ganas de que cada día sea mejor. Y por suerte tenemos profesores de lujo para lograrlo, y ayudantes que le ponen muchas ganas a enseñar y que juntos llevan las clases a otro nivel. Siempre agregando clases de consulta cuando se necesitan o inclusive acercándose cuando nos ven estudiar en la biblioteca para ver si hay algo en lo que estemos trabados. La verdad que se nota que no es sólo un trabajo. Gracias a todos.

En particular quiero agradecer especialmente a mis directores de tesis, que me bancaron durante todo este tiempo; porque bancar es la palabra. Que cuando entendieron lo colgado que soy, me ayudaron de mil maneras para que no pare y siga avanzando. Siempre super disponibles para cualquier consulta o propuesta, con las puertas de la oficina siempre abiertas, ayudándome a seguir y definir este trabajo. Creo que al final nos quedó lindo. :)

También quiero agradecer a las personas que se sumaron a esta carrera en los últimos metros. Mis compañeros de trabajo que me bloquearon los fines de semana para que escriba la tesis y que me hacían un seguimiento paso a paso para asegurarse de que no me estanque. Que con sus consejos y ánimo me empujaron hasta la linea de meta, gracias!.

A mis amigos, que simplemente hacen de la vida una buena vida. Porque de nada sirve recibirse si no se pueden enfrentar a los dioses antigüos con fernet y picada. Si bien un poco se quejaban cuando faltaba a una juntada porque tenía que hacer la tesis, sólo se quejaron lo suficiente para demostrarme su cariño. Gracias por los largos fines de semana y las charlas esperando el bondi. Vamos ganando mayoría en el grupo de los recibidos!

Y bueeeno, a vos Caro, que sos el amor de mi vida y que me apoyaste como si esta fuera tu tesis. Que la leíste conmigo, la mejoramos juntos y me diste la mezcla perfecta de ayuda y espacio. Y por sobre todo mucho, pero mucho amor. Me ayudaste a preparar la presentación, para que no sea sea sólo una descripción técnica y fuiste también la primer persona con la que la practiqué. Sin vos no me hubiera sentido tan cómodo presentándola. Gracias por aparecer en mi vida y gracias por quererme tanto y darme tu apoyo tan incondicional.

Para Fernando, te extraño viejo.

Índice general

1.. Introducción 1

2.. Preliminares 5
2.1. Problema SAT 5
2.1.1. Fórmula Proposicional 5
2.1.2. Valuación 6
2.1.3. Satisfacibilidad 7
2.2. SAT-solver 7
2.2.1. *SAT-solver* DPLL - CDCL 9
2.2.2. Learning en Sat-Solvers Secuenciales 12
2.2.3. Restart en Sat-Solvers Secuenciales 14
2.3. Herramienta Paralloy 15
2.3.1. Proceso Client de la herramienta Paralloy 17
2.3.2. Proceso Worker de la herramienta Paralloy 19

3.. Desarrollo 21
3.1. Cláusula de conflicto 22
3.1.1. Transmisión de la información 23
3.1.2. Rol del proceso Client en el esquema de Learning 24
3.1.3. Rol del proceso Worker en el esquema de Learning 24
3.2. Proceso de Restart distribuido 25
3.2.1. Estrategia de Restart por Unsat 26
3.2.2. Estrategia de Restart por Abort 27
3.2.3. Estrategia de Restart por longitud de colas de trabajo 27
3.2.4. Experimentación preliminar sobre Restart 27
3.3. Learning de cláusulas aprendidas 28
3.3.1. Learning de cláusulas aprendidas en Paralloy 29
3.3.2. Rol del proceso Client en el Learning distribuido 30
3.3.3. Rol del proceso Worker en el Learning distribuido 33

4.. Evaluación experimental 35
4.1. Marco experimental 35
4.1.1. Dispersión del tiempo de ejecución 35
4.1.2. Hardware utilizado durante la experimentación 36
4.1.3. Configuración del sistema 36
4.1.4. Configuración base de Paralloy 37
4.2. Aspectos del sistema analizados 38
4.2.1. Estrategias de Restart analizadas. 38
4.2.2. Estrategias de Learning analizadas 39
4.2.3. Problemas considerados 39
4.3. Experimentación 40
4.3.1. Efecto del Learning sobre sub-problemas aislados 41
4.3.2. Análisis sobre la distribución del esfuerzo en los Workers. 45

4.3.3. Rendimiento general de la herramienta introduciendo las técnicas propuestas. 49

5.. Conclusiones, Limitaciones y Trabajo Futuro 55
5.1. Análisis de los resultados 55
5.1.1. Impacto de la técnica de Learning aplicada 55
5.1.2. Impacto del Restart sobre la herramienta 57
5.2. Limitaciones y Trabajo futuro 58
5.3. Conclusiones finales 59

Apéndice A..Configuración y uso de la herramienta. 61

Apéndice B..AVL hbnd 63

Apéndice C..Closure 65

Apéndice D..Mark & Sweep 67

Apéndice E..Network Routing 69

Apéndice F..Stable Mutex Ring 77

1. INTRODUCCIÓN

La lógica proposicional y el álgebra booleana son dos de los pilares fundamentales de las ciencias exactas. Su estudio ha generado diversos problemas a lo largo de los años. Uno de estos problemas, el problema de la satisfacibilidad (también conocido como SAT), es un problema que atraviesa distintas áreas; desde la lógica, la teoría de grafos, ingeniería de software, investigación operativa, etc. El problema consiste en, dada una fórmula de la lógica proposicional, determinar si existe alguna valuación de las variables que haga verdadera a dicha fórmula, o si, por el contrario, no existe ninguna valuación que satisfaga a dicha fórmula.

En 1971, Stephen Cook [Coo71] lo definió como el primer problema NP-Completo, convirtiéndolo en un problema fundamental de las ciencias de la computación, por lo que su estudio y análisis es de una gran relevancia.

A través de los años diversos problemas prácticos se han expresado como instancias de un problema SAT, lo que llevó a que su resolución generara interés no sólo dentro de la comunidad científica. Entre ellos podemos nombrar problemas de planeamiento, asignación de recursos y verificación, tanto de hardware como de software, entre otros. En este contexto, se ha puesto gran énfasis en el desarrollo de software para su resolución. Una prueba de ello son las competiciones SAT[1] [SAT02] [18t15], donde se presentan diversos *SAT-solvers*[2] compitiendo para determinar cuál es capaz de resolver la mayor cantidad de problemas de la manera más eficiente para un rango de tiempo acotado.

Paralloy [Vis13] nació como un *SAT-solver* orientado a resolver problemas de verificación de software. Su nombre se debe al lenguaje Alloy [Jac06] y su herramienta homónima, una formalización que permite definir modelos, relaciones y propiedades, para luego chequear su consistencia usando un enfoque de *Bounded Model Checking* [BCCZ99] (verificación acotada de modelos).

La técnica de *Bounded Model Checking* pretende una exploración exhaustiva de todos los posibles estados de los modelos definidos, relaciones y sus propiedades para así lograr un chequeo de consistencia en la especificación planteada. Pero para lograr un proceso completo y decidible, acota el dominio (posiblemente infinito) del modelo planteado. Por ejemplo, para chequear una propiedad sobre los números enteros, sólo se chequea dicha propiedad para los números enteros menores a 1000. Esto genera una consecuente pérdida de completitud ya que prueba la validez de una fórmula en un dominio acotado, pero no se prueba que la misma se mantenga en el dominio original[3]. Considerando esta limitación, este procedimiento permite realizar un chequeo automático sobre una pieza de software otorgando cierto grado de confianza sobre su correctitud.

Para realizar este chequeo, la herramienta Alloy traduce la especificación a chequear a una instancia de un problema SAT, de manera tal que mediante su resolución es capaz de decidir la validez o no de la propiedad analizada. Más específicamente, para probar una propiedad se trata de encontrar un modelo que cumpla la negación de la propiedad, si dicho modelo existe (SAT), éste define un contraejemplo válido, pero si dicho modelo

[1] The international SAT Competitions, SAT Competition 2016

[2] Pieza de software dedicado a resolver instancias de problemas SAT

[3] Asumiendo que el dominio original es distinto del dominio en el que se realiza el chequeo.

resulta inexistente (UNSAT) la propiedad es entonces válida para el dominio definido.

En general, el costo de probar una propiedad mediante este método crece de manera exponencial en relación al tamaño del dominio considerado. Esto se debe a que para un nuevo elemento del dominio, se deben probar todas las configuraciones posibles, esto se refleja en el modelo en un aumento al menos lineal de las variables del modelo. Como ya dijimos es un problema SAT de la clase NP-Completo, por lo que los algoritmos de resolución conocidos son de complejidad exponencial en cuanto al tamaño de variables.

En este contexto, y dado el auge de los sistemas distribuidos y los clusters de cómputo, *Paralloy* nació como un intento de expandir el poder de Alloy para ser capaz de verificar propiedades sobre dominios de mayor tamaño, aprovechando el poder combinado de varias unidades de cómputo. A su vez siendo un *SAT-solver* totalmente funcional es posible su utilización para resolver cualquier problema SAT, sea proveniente de una especificación Alloy o no.

Vale aclarar que *Paralloy* no es el primer intento de aprovechar el poder de varias unidades de cómputo para resolver un problema SAT. Si bien ha habido intentos en el campo de Solving paralelo, la mayoría de ellos han sido para procesadores multi-core, utilizando en los casos más exitosos técnicas de *portfolio* como MANYSat [HS09], SATZilla [XHHLB08] o Plingeling [Bie10]. Este enfoque, consiste utilizar un conjunto de *SAT-solvers* secuenciales con alguna diferencia en su configuración que hace variar el tiempo de resolución para un mismo problema. Al correr las diferentes versiones en paralelo, es posible tomar el resultado del solver que finalice primero, disminuyendo el tiempo total para el solver. A su vez, algunos de ellos poseen un mecanismo de cooperación mediante el intercambio de cláusulas aprendidas (ya sea inyectando o manteniendo una base de datos de cláusulas aprendidas en común). Esto hace que el tiempo total sea inferior al menor de los tiempos entre las diferentes versiones, pero en todos estos casos cada unidad de cómputo intenta resolver la totalidad del problema, sin realizar ningún tipo de división del problema.

Existen también esfuerzos aislados para desarrollar *SAT-solvers* distribuidos, pero no hemos encontrado ninguna herramienta completamente funcional. Otro *SAT-solver* que vale destacar es PSATO [ZBP$^+$96], que utiliza un esquema *master-slave* también inspirado en el algoritmo de CDCL. Si bien hay gran interés en el campo, creemos que aún es un área que puede aportar avances significativos.

Paralloy está basado en el algoritmo DPLL (Davis–Putnam–Logemann–Loveland [DP60] y [DLL62]), un algoritmo de *backtracking* refinado a mediados de los años 90 (Conflict-Driven Clause Learning CDCL [BS97] y [MSS99]) que significó un salto cualitativo en el campo del sat-solving. En las citas referidas se presenta la técnica de *learning*; una técnica de poda del espacio de búsqueda mediante el uso de información de las ramas procesadas anteriormente.

Otro gran avance se dio en el año 1998 [GSK98] cuando se propuso el mecanismo de *restart*, este mecanismo consiste en reiniciar la búsqueda mediante algún criterio una vez comenzada la exploración, . Su objetivo es reducir la incidencia de malas decisiones tomadas en etapas tempranas sobre el total de la ejecución.

Ambas técnicas son esenciales para obtener el nivel de eficiencia que se obtiene del algoritmo CDCL en la actualidad. Es por eso que en este trabajo nos abocamos a estudiar el impacto de estas técnicas en un *SAT-solver* paralelo como lo es *Paralloy*.

Tanto *learning* como *restart* son técnicas que requieren un estado global del sistema,

y mientras que otras optimizaciones que tienen injerencia local[4] no se ven afectadas al particionar el cómputo en sub-problemas independientes, si no existe un mecanismo de interacción entre los nodos del sistema la implementación de dichas estrategias globales no es posible.

Este trabajo busca subsanar este problema aportando un sistema de distribución de la información y colaboración entre las distintas unidades de cómputo, implementando un sistema de *learning* y *restart* distribuido, estudiando su impacto en el rendimiento global del sistema.

Preguntas de Investigación

El objetivo principal de este trabajo se puede resumir en las siguientes dos preguntas de investigación:

1. ¿Es posible definir un sistema de distribución de la información aprendida para el *SAT-solver Paralloy* que reduzca el tiempo de resolución promedio para el sistema?
   a) ¿La información que se genera en cada una de las unidades distribuidas de computo, resulta relevante[5] para el resto del sistema?
   b) Dado que la cantidad de información generada durante la resolución se presume demasiada para ser transmitida a la totalidad del sistema, ¿Es posible priorizarla de manera que la ganancia supere el costo de su administración?
2. La técnica de *restart* demuestra tener un efecto positivo en el tiempo de ejecución promedio para los *SAT-solvers* secuenciales. Una vez definido un procedimiento de aprendizaje distribuido, ¿Cuál es el efecto que tiene un proceso de *restart* global sobre un esquema de solving distribuido?
   a) ¿Es posible lograr una mejora en el tiempo promedio para el sistema Paralloy aplicando esta técnica?

El trabajo se encuentra organizado de la siguiente manera: en el Cap. 2 se presentan ciertas nociones y definiciones preliminares que serán necesarias a lo largo del trabajo. Luego, en el Cap. 3, se explican las motivaciones y el trabajo llevado a cabo sobre la herramienta *Paralloy* para probar las distintas hipótesis y tratar de encontrar respuesta a las preguntas planteadas; a continuación tiene lugar al Cap. 4 que explica los experimentos realizados para analizar el comportamiento de la herramienta; y finaliza en el Cap. 5, donde se estudian los resultados obtenidos realizando un análisis comparativo con la versión previa de la herramienta, dando respuesta a las preguntas planteadas.

[4] Un ejemplo de una optimización local puede ser *Two Watched Literals*[MMZ+01].

[5] Información que no será rápidamente descartada y qué tendrá impacto en el tiempo de ejecución necesario para la resolución de un problema.

2. PRELIMINARES

En el siguiente capítulo se presentan nociones y definiciones preliminares que serán necesarias a lo largo del trabajo. En la Sec. 2.1 se presenta el problema de satisfacibilidad proposicional o SAT. En esta sección se realiza una definición formal del problema. Luego, en la Sec. 2.2, se introducen los *SAT-solvers* secuenciales, se explica su funcionamiento básico y algunas de las técnicas utilizadas que tendrán gran relevancia en este trabajo, como ser *learning* y *restart*. Por último, en la Sec. 2.3 se explica la herramienta *Paralloy* y nociones de su funcionamiento y arquitectura que resultan relevantes para la comprensión de este trabajo.

2.1. Problema SAT

El problema de satisfacibilidad proposicional (o SAT) es un problema de decisión sobre la existencia o no de una valuación que satisfaga una fórmula proposicional dada. Si dicho modelo existe, se dirá que la fórmula es Satisfacible (o SAT) y en caso de no existir, la fórmula resulta Insatisfacible (o $UNSAT$). Vale aclarar que en caso de ser SAT puede existir más de un modelo que la satisfaga.

A continuación detallaremos en profundidad qué es una **fórmula proposicional**, qué es una **valuación** y el concepto de **satisfacibilidad**.

2.1.1. Fórmula Proposicional

Toda **fórmula proposicional** está formada por un conjunto de variables ($\mathcal{V}ar$) y conectores lógicos: negación ($\neg$), conjunción ($\vee$), disyunción ($\wedge$), implicación ($\Rightarrow$) y equivalencia ($\Leftrightarrow$)[Men97].

Definición 1 (Fórmula bien formada)**.** Se define $Form$ como el conjunto de **fórmulas bien formadas**. Diremos que $\phi \in Form$ si y solo si se cumple con alguna de las siguientes condiciones:

1. $\phi \in \mathcal{V}ar$ (es una variable)
2. $\phi = \neg\varphi$, y $\varphi \in Form$
3. $\phi = \varphi \vee \psi$, y $\varphi, \psi \in Form$

Los conectores lógicos $\wedge$, $\Rightarrow$ y $\Leftrightarrow$ se pueden reescribir como:

$$\varphi \wedge \psi \equiv \neg\left(\neg\varphi \vee \neg\psi\right)$$

$$\varphi \Rightarrow \psi \equiv \neg\varphi \vee \psi$$

$$\varphi \Leftrightarrow \psi \equiv (\varphi \Rightarrow \psi) \wedge (\psi \Rightarrow \varphi)$$

Definición 2 (Soporte)**.** El **soporte** de una fórmula, $sop(\varphi)$, representa el conjunto de variables implicadas en la fórmula. Se define $sop : Form \to \mathcal{P}(\mathcal{V}ar)$[1]:

$$\forall v \in \mathcal{V}ar, sop(v) = \{v\}$$

[1] Conjunto de partes de $\mathcal{V}ar$

$$\forall \varphi \in Form, sop(\neg\varphi) = sop(\varphi)$$

$$\forall \varphi \in Form, sop(\varphi \vee \psi) = sop(\varphi) \cup sop(\psi)$$

Observación 3. *En este trabajo sólo tendremos en cuenta las fórmulas proposicionales finitas. Es decir, aquellas cuyo soporte sea un conjunto finito.*

2.1.2. Valuación

Definición 4 (Valuación para variables)**.** Una **valuación** V para una fórmula proposicional φ es una función total $V_\varphi : sop(\varphi) \to \{Verdadero, Falso\}$, que asigna **a cada variable** en φ un valor de verdad.

Definición 5 (Valuación parcial)**.** Diremos que una valuación es **parcial** si sólo está definida para un conjunto $C \subseteq sop(\varphi)$, la denotaremos $V|_C$.

$$V|_C(v) = \begin{cases} V(v) & \text{si } v \in C \\ Indef & \text{si } v \notin C \end{cases}$$

Observación: $V|_{sop(\varphi)} \equiv V$.

Definición 6 (Extensión de una valuación)**.** A su vez, dada una valuación parcial $V|_C$, diremos que:

- $V|_D$, con $D \equiv C \cup \{v\}$ es una **extensión** de $V|_C$ **para la variable** v.
- $V|_E$, con $E \supset C$ es una **extensión** de $V|_C$.

Para simplificar la notación también representaremos una valuación $V|_C$ como un conjunto de pares ordenados, variable $\to$ valor de verdad:

$$\{(v, c) | \forall v \in C, \text{ tal que } V|_C(v) = c\}$$

Definición 7 (Valuación para fórmulas proposicionales)**.** Se define $V^* : Form \to \{Verdadero, Falso\}$ como:

$$\forall v \in \mathcal{V}ar, V^*(v) \equiv V(v)$$

$$V^*(\neg\varphi) = \begin{cases} Verdadero & \text{si } V^*(\varphi) = Falso \\ Falso & \text{si } V^*(\varphi) = Verdadero \end{cases}$$

$$V^*(\varphi \vee \psi) = \begin{cases} Falso & \text{si } V^*(\varphi) = Falso \text{ y } V^*(\psi) = Falso \\ Verdadero & sino \end{cases}$$

2.1.3. Satisfacibilidad

- Se dice que una valuación V **satisface** una fórmula φ sii $V^*(\varphi) = Verdadero$.
- Si tal valuación no puede ser construida, φ resulta insatisfacible.

$$\forall \varphi \in Form, \varphi \in SAT \leftrightarrow \exists V \text{valuación, tal que } V^*(\varphi) \equiv Verdadero$$

Por ejemplo, la fórmula $(\neg p \vee q) \wedge (\neg p \vee \neg q)$ es satisfacible, ya que por ejemplo la siguiente valuación la satisface:

$$V(x) = \begin{cases} Falso & \text{si } x = p \\ Verdadero & \text{si } x = q \end{cases}$$

Es más, cualquier extensión de $V|_{\{p\}} = \{(p, Falso)\}$, satisface φ.

Por otro lado, la fórmula $\varphi = p \wedge (\neg p \vee q) \wedge (\neg p \vee \neg q)$ es insatisfacible. Ya que cualquier extensión de $V|_{\{p\}} = \{(p, Falso)\}$ no satisface φ. Y lo mismo sucede para las extensiones de $V|_{\{p\}} = \{(p, Verdadero)\}$, porque si $V(p) = Verdadero$, tanto $V(q) = Verdadero$ como $V(q) = Falso$ también resulta en $V^*(\varphi) = Falso$. Es decir:

- $V = \{(p, Verdadero), (q, Verdadero)\} \Rightarrow V^*(\varphi) = Falso$
- $V = \{(p, Verdadero), (q, Falso)\} \Rightarrow V^*(\varphi) = Falso$

Observación 8. *Notar que para $\psi \equiv \phi \wedge \varphi$, si se demuestra que $\forall V^*_\phi(\phi) = Falso$, entonces no sólo ϕ resulta* UNSAT*, sino que también ψ lo es. Ya que toda V_ψ es una extensión de alguna V_ϕ.*

De esta manera, para toda fórmula finita, probando exhaustivamente todas las valuaciones posibles se puede encontrar una que satisfaga φ o se puede demostrar que no existe valuación que la satisfaga. Esto es gracias a que el soporte de una fórmula es finito y los valores de verdad que puede tomar cada variable del soporte también son una cantidad finita (dos: Verdadero y Falso). Esta es básicamente la búsqueda que llevan a cabo los *SAT-solver*, el procedimiento se explicará en mayor detalle en la siguiente sección.

2.2. SAT-solver

Una fórmula en Forma Normal Conjuntiva (*CNF* por sus siglas en inglés) es una fórmula definida como una conjunción de disyunciones de literales. Siendo los literales variables o negaciones de variables.

Es decir, una fórmula de la forma:

$$\varphi = \bigwedge_{i=1}^{n} \left(\bigvee_{lit \in C_i} lit \right)$$

Siendo cada C_i un conjunto de literales.

Por ejemplo, la fórmula

$$\varphi = \neg p \Rightarrow (q \wedge \neg r)$$

No se encuentra en *CNF* porque contiene una implicación, pero se puede rescribir a *CNF* de la siguiente manera:

$$\varphi = (p \vee q) \wedge (p \vee \neg r)$$

Se puede demostrar que toda fórmula bien formada se puede llevar a *CNF* sin perdida de generalidad[2].

Definición 9 (Árbol de decisión)**.** Dada una fórmula $\varphi \in Form$. Un árbol de decisión de φ es un árbol binario completo donde los nodos internos son las variables de $sop(\varphi)$. Los ejes salientes representan el valor de verdad de la variable (Verdadero y Falso). Las hojas son vacías. Por último, todo camino desde la raíz a cada hoja contiene todas las variables de $sop(\varphi)$ sin repetidos.

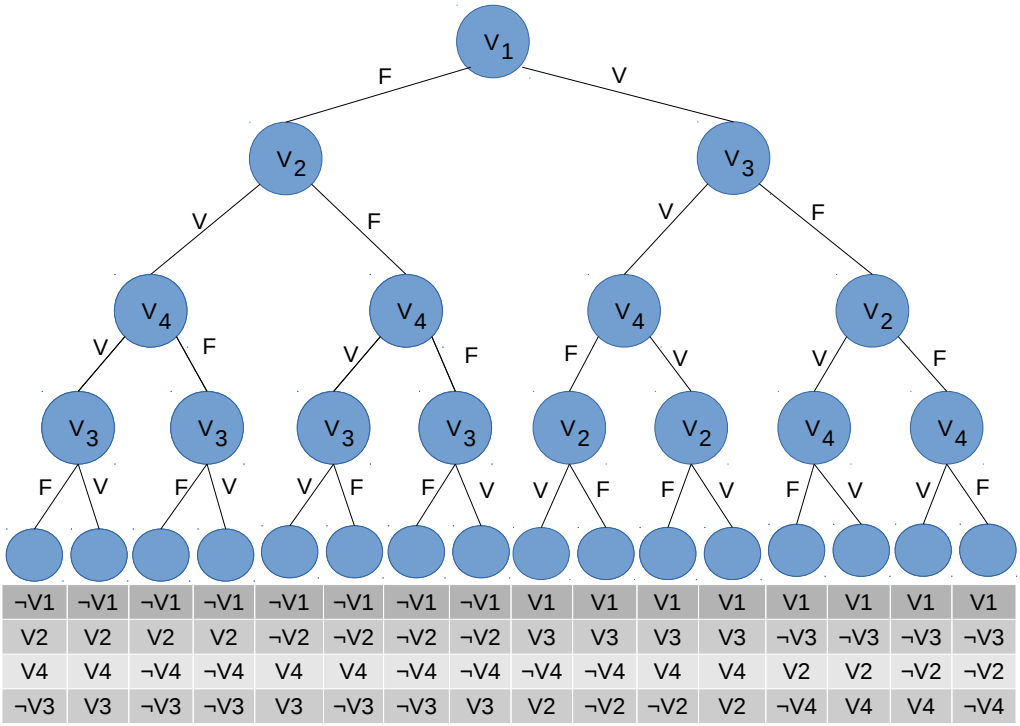

Fig. 2.1: Ejemplo de un árbol de decisión para un conjunto de variables $\{v_1, v_2, v_3, v_4\}$.

Definición 10 (Valuación asociada)**.** Dado un camino de un árbol de decisión, $v_{i_0}, v_{i_1}, ..., v_{i_n}$. llamaremos **valuación asociada al camino** a la valuación V, tal que:

$$V(v_{i_j}) = \begin{cases} Verdadero & \text{si el eje } v_{i_j} \to v_{i_{j+1}} \text{ en el camino es } Verdadero \\ Falso & \text{si el eje } v_{i_j} \to v_{i_{j+1}} \text{ en el camino es } Falso \end{cases}$$

Observación 11. *Diremos que un camino de un árbol de decisión satisface una fórmula sii al valuación asociada al camino lo hace.*

[2] La demostración de esta propiedad se puede ver en [Men97]

Observación 12. *En la figura 2.1 se puede ver un posible árbol de decisión para una fórmula con soporte* $\{v_1, v_2, v_3, v_4\}$*. Debajo de cada hoja se muestra la valuación asociada a cada camino.*

Teorema 1 (Completitud)**.** *Todo árbol de decisión para una fórmula* φ *contiene todas las posibles valuaciones.*

Corolario 1. Dada una fórmula φ, si se toma un árbol de decisión de φ:

- Si existe un camino que satisfaga φ, entonces es satisfacible por la valuación asociada al camino.
- Si ninguno de los caminos del árbol satisface φ, como el árbol contiene todas las posibles valuaciones, la fórmula es insatisfacible.

2.2.1. *SAT-solver* DPLL - CDCL

Los *SAT-solver* DPLL se basan en los conceptos descriptos para encontrar una valuación que satisfaga una fórmula dada o determinar la insatisfacibilidad de la misma. Para ello realizan una búsqueda sobre un árbol de decisión de la fórmula. El algoritmo CDCL es un refinamiento de DPLL que tomándolo como base incorpora mecanismos para disminuir el tiempo de resolución, uno de ellos es el *learning*. A su vez los *SAT-solvers* basados en estos algoritmos también utilizan (en su mayoría) un proceso de *restart* para mejorar su eficiencia. En el transcurso de este capítulo nos adentraremos en estos conceptos.

Antes de adentrarnos en el algoritmo DPLL, vale hacer una aclaración en cuanto al valor de verdad de una valuación para una fórmula en *CNF*; cuando se tiene una fórmula en *CNF*, dado que se trata de una conjunción de cláusulas, basta que una de las cláusulas sea falsa para que la fórmula resulte falsa, sin importar el valor del resto de las variables[3]. A su vez cada cláusula es una disyunción de literales, por lo que esta será falsa sólo cuando todos y cada uno de los literales que la componen sean falsos (recordar que un literal puede ser una variable o su negación).

Teniendo esto en cuenta, a partir de una valuación parcial, se pueden determinar tres tipos de cláusulas:

(Tipo I) Clausulas con al menos un literal con valor Verdadero (que la valuación hace verdadero)

(Tipo II) Clausulas con *todos* sus literales con valor Falso (que la valuación hace falsos)

(Tipo III) Clausulas con al menos un literal libre[4] y sin literales con valor Verdadero (Que no sean de Tipo I).

[3] Ver observación 8.

[4] En este contexto se llamará *literal libre* a un literal cuya variable asociada no está definida en la valuación parcial

Dada una valuación V tal que:

$$\begin{cases} V(v_1) = Verdadero \\ V(v_2) = Falso \\ V(v_3) = Indef \\ V(v_4) = Indef \\ V(v_5) = Falso \end{cases}$$

Se pueden ejemplificar cláusulas de los tipos antes descriptos como:

(Tipo I) $(v_3 \vee v_1 \vee v_2)$ por contener a v_1.

(Tipo II) $(v_2 \vee v_5)$ porque v_2 y v_5 tienen valor $Falso$ en la valuación.

(Tipo III) $(v_2 \vee v_3 \vee v_4)$ por contener a v_3 o v_4.

Definición 13 (Cláusula unitaria)**.** Una cláusula se dice unitaria para una valuación parcial si es una cláusula de Tipo III con sólo un literal libre. En el ejemplo anterior, $(\neg v_1 \vee v_2 \vee v_3)$ es una cláusula unitaria, siendo v_3 su único literal libre.

Entonces, como ya dijimos, el algoritmo DPLL realiza una búsqueda sobre un árbol de decisión, pero dicho árbol es generado dinámicamente durante la búsqueda de un camino que haga a la fórmula satisfacible. La metodología base para este procedimiento se puede describir de la siguiente manera:

1. Elegir una variable raíz.
2. ↓ Para la variable elegida elegir un valor de verdad, a esto le llamaremos decisión.
3. Chequear la validez de la fórmula con este nuevo valor de verdad
4. - Si no hay cláusulas de Tipo II ir al paso 6.
   - ↑ Si existe alguna cláusula de Tipo II deshacer la ultima decisión y marcar la decisión como explorada, esta rama se debe descartar.
5. - ↓ Si la decisión contraria no está marcada como explorada, tomarla e ir al paso 3.
   - Sino significa que ambos valores de verdad dan valor Falso a la cláusula, entonces:
     - Si me encuentro en la raíz, la fórmula es insatisfacible → Terminar.
     - ↑ Sino deshacer la última decisión, marcar la decisión como explorada y volver al paso 5.
6. Elegir una nueva variable libre e ir al paso 2. Si no hay variables libres se tiene una valuación satisfactoria → Terminar.

En la descripción se marcaron con ↓ los pasos donde se toma una transición en el árbol de decisión extendiendo la valuación y con ↑ aquellos donde se sube un nivel, descartando la última extensión de la valuación considerada.

Propagación de cláusulas unitarias

Proposición 1. Para que una cláusula unitaria sea verdadera, sólo existe una posibilidad para el valor de verdad del literal libre de la misma:

- Si el literal es v_k, se debe cumplir $V(v_k) = Verdadero$
- Si el literal es $\neg v_k$, se debe cumplir $V(v_k) = Falso$

Observación 14. *Dado que éste es el único literal libre de la cláusula, sólo las extensiones de la valuación actual que cumplan con esto podrán satisfacer a la fórmula.*

Una mejora que se puede realizar al algoritmo planteado es la llamada *propagación de cláusulas unitarias* (o *UP*, las siglas de Unit Propagation). La misma trabaja sobre cláusulas de Tipo III con sólo un literal libre, o *cláusulas unitarias*.

Cuando se encuentra una cláusula unitaria, dado que para que la fórmula sea Satisfacible toda cláusula debe ser Verdadera, toda extensión de la valuación actual que no cumpla con la proposición 1 debe ser descartada. Al chequear esta condición cada vez que se define una nueva extensión de la valuación actual, se puede reducir drásticamente el tiempo de ejecución descartando tempranamente ramas del árbol de decisión.

Definición 15 (Contradicción)**.** Decimos que una contradicción ocurre cuando una cláusula unitaria requiere que una variable (v_i) tenga cierto valor de verdad (L) y en la valuación parcial (V) está definida con el valor contrario. Es decir:

$$V(v_i) = \neg L$$

Realizando este chequeo también se puede llegar a una contradicción de forma temprana. Cuando existen dos cláusulas unitarias C_1 y C_2, donde el único literal libre es v_k y $\neg v_k$ respectivamente, se puede afirmar que no existe una extensión de la valuación actual tal que $C_1 \wedge C_2 = Verdadero$. Ya que sería necesario que $V(v_k) = Verdadero$ y $V(v_k) = Falso$ al mismo tiempo. Esto define que la valuación parcial o lo que es equivalente, el camino actual, debe ser desestimado. Nótese que este chequeo se realiza sólo entre las fórmulas de tipo III y es iterativo ya que, si se encuentra una cláusula unitaria y se define el valor de la variable involucrada, esto puede generar una nueva cláusula unitaria.

Esta es una explicación del concepto general detrás de la propagación de cláusulas unitarias (o BCP). Para más información ver [BBH+09]. Este proceso es parte esencial de los *SAT-solver* DPLL dado que significa cerca de un %80 del tiempo de ejecución total.

Two watched literals

Para realizar esta propagación es necesario detectar qué cláusulas son unitarias. Para realizar esto de una manera eficiente existe una técnica ampliamente adoptada llamada *two watched literal*[MMZ+01]. La misma consiste en definir en todo momento dos literales libres *a mirar* para cada cláusula de Tipo III. De esta manera sólo será necesario chequear si esa cláusula se transforma en cláusula unitaria cuando se defina el valor de verdad de alguno de estos dos literales, es decir, sea incorporado a la valuación parcial.

Dados dos literales *a mirar* (l_i, l_j) en una cláusula, cuando se le asigna valor de verdad[5] a alguno de los dos, por ejemplo l_i, si la cláusula tiene al menos un tercer literal libre l_k, se

[5] Se le asigna un valor de verdad a un literal cuando la valuación es extendida definiendo un valor para la variable que forma al literal.

lo definirá a este en remplazo del literal l_i, obteniendo nuevamente una dupla (l_k, l_j) para la cláusula dada[6]. En caso de no tener otro literal libre, la cláusula se convierte en cláusula unitaria y se debe propagar el valor de verdad de acuerdo a lo definido por Prop. 1 para l_j.

Durante la propagación, como ya se explicó, se puede llegar a una contradicción. El *aprendizaje* (o *learning*) es una técnica mediante la cual se logran descartar ramas del árbol de forma prematura a partir de la información obtenida de contradicciones previas. A continuación se explicará más detalladamente este concepto.

2.2.2. Learning en Sat-Solvers Secuenciales

Las técnicas de poda sirven para descartar prematuramente ramas del árbol de decisión que resulten en valuaciones insatisfactorias sin necesidad de recorrerla en su totalidad. Con esto se reduce el espacio de búsqueda y por lo tanto se disminuye el tiempo de ejecución sin pérdida de correctitud.

Como ya se vio en la sección anterior, durante el proceso de *UP* se puede llegar a dos cláusulas unitarias que resultan contradictorias. Es decir, para la valuación parcial actual $V|_D$, dos cláusulas unitarias de la forma:

$$C_1 = l_{i_1} \vee ... \vee l_{i_n} \vee v_k$$

$$C_2 = l_{j_1} \vee ... \vee l_{j_m} \vee \neg v_k$$

Tal que ${V|_D}^*(l_{i_k}) = Falso$ con $1 \leq k \leq n$, ${V|_D}^*(l_{j_p}) = Falso$ con $1 \leq p \leq m$ y $v_k \notin D$. Entonces, cualquier extensión de $V|_D$ resultará insatisfactoria.

Pero esta contradicción proviene de decisiones que se tomaron previamente, ya que para que dicha contradicción exista es necesario que se cumpla lo siguiente:

$${V|_D}^*(l_{i_1} \vee ... \vee l_{i_n} \vee l_{j_1} \vee ... \vee l_{j_m}) = Falso$$

Entonces, para que φ sea satisfacible:

$${V|_D}^*(l_{i_1} \vee ... \vee l_{i_n} \vee l_{j_1} \vee ... \vee l_{j_m}) = Verdadero$$

Definición 16 (Conjunto de decisión)**.** Dentro del contexto del algoritmo de CDCL, diremos que el conjunto de decisión de un literal son aquellas decisiones que implicaron el valor de la variable que lo forma. Más formalmente, dada una formula φ, una valuación parcial $V|_C$ y un literal l, $D \subseteq C$ es el conjunto de decisión de l sii:

$$V|_D(\varphi) = Indef \wedge V|_{D \cup \{sop(l)\}}(l) = Falso \wedge V|_{D \cup \{sop(l)\}}(\varphi) = Falso$$

[6] Notar que si en vez de asignar un valor de verdad a l_i, se le hubiera asignado a l_k, el algoritmo no necesita realizar ningún chequeo sobre la cláusula, sólo la tomará en cuenta cuando se le asigne valor a l_i o al_j.

En el algoritmo CDCL, el valor de verdad de una variable $v \in D$ (para un camino dado D) puede haber sido definido por dos causas distintas: a) mediante la toma de decisiones o, b) mediante el algoritmo de propagación de cláusulas unitarias . En el primer caso diremos que el *conjunto de decisión* de los literales cuyo soporte es v, es el conjunto unitario con dicha decisión. En el segundo caso, el valor de verdad de la variable v fue determinado a partir de la propagación una cláusula unitaria. Por lo tanto, el resto de los literales de dicha cláusula $l_0, ..., l_n$ cumplen con que $V_D^*(l_i) = Falso$. En este caso definimos el *conjunto de decisión* como la unión de los conjuntos de decisión de cada uno de estos literales.

Hay otro caso particular que es aquel en el cual el problema original contiene cláusulas unitarias, en ese caso se dirá que el conjunto de decisión para esos literales será el conjunto vacío.

De esta forma, a partir de una contradicción se puede rastrear el conjunto de decisiones que la generó.[7].

Definición 17 (Literal asociado a una decisión)**.** Dada una decisión sobre una variable v_k, si el valor de verdad es $Verdadero$ diremos que v_k será el literal asociado a dicha decisión. En cambio si el valor de verdad es $Falso$, $\neg v_k$ será el literal asociado.

Dada una contradicción para una resolución de una fórmula φ, si tomamos entonces su conjunto de decisiones (las decisiones que fueron necesarias para llegar a la contradicción), y formamos el conjunto de literales asociados a cada uno de los elementos: $L_{asoc} : \{l_{c_1} ... l_{c_o}\}$, tenemos que:

$$\varphi \wedge \bigwedge_{l \in L_{asoc}} l \Rightarrow \neg l_{i_1} \wedge ... \neg l_{i_n} \wedge \neg l_{j_1} \wedge ... \neg l_{j_m}$$

Para que φ sea satisfacible, es necesario que se cumpla:

$$\neg(\bigwedge_{l \in L_{asoc}} l)$$

O lo que es equivalente,

$$\bigvee_{l \in L_{asoc}} \neg l$$

Esta fórmula define una nueva cláusula que es necesaria para que φ sea satisfacible, a esta cláusula se le llama *cláusula aprendida*. Nótese que el conjunto L_{asoc} podría ser de longitud 1, en este caso particular, al literal a satisfacer se le llamará *hecho* o *fact*.

Tanto las *cláusulas aprendidas* como los *hechos* son requisitos necesarios para satisfacer φ y no dependen del árbol de decisión ni de ningún otra particularidad. Si se recomenzara la resolución del mismo problema y se generará un nuevo árbol de decisiones, estas cláusulas aprendidas se mantendrían válidas.

El proceso de recomenzar a resolver un problema generando otro árbol de decisión se define como *restart*. En la siguiente sección se explicará en qué consiste y sus beneficios potenciales.

[7] Existen distintos algoritmos para encontrar un buen conjunto de decisiones de forma eficiente dada una contradicción. Ver [BBH+09]

2.2.3. Restart en Sat-Solvers Secuenciales

En principio, decidir descartar el trabajo realizado y recomenzar nuevamente podría parecer una mala idea. Pero empíricamente, en los *SAT-solver* actuales basados en el algoritmo CDCL, esta estrategia ha demostrado dar buenos resultados. Este fenómeno se puede dar por la gran variación en cuanto al tiempo de ejecución entre dos corridas que exploren el espacio de búsqueda en distinto orden. Consideremos el siguiente ejemplo:

$$\varphi = (v_1 \vee v_2 \vee v_3 \vee v_4 \vee v_5 \vee v_6 \vee v_7 \vee v_8 \vee v_9) \wedge (\neg v_{10} \vee v_{11}) \wedge (\neg v_{11} \vee v_{10}) \wedge (v_{11} \vee v_{10}) \wedge (\neg v_{11} \vee \neg v_{10})$$

Se puede observar que la fórmula planteada es insatisfacible, porque no existe combinación de valores para v_{10} y v_{11} que la hagan verdadera. Sin embargo, mientras se defina verdadera a alguna de las variables en $\{v_1 ... v_9\}$, entre estas variables, no exisitrá contradicción.

Entonces, un árbol de decisión que comience por las variables $v_1 ... v_9$ no encontrará contradicciones hasta que decida darle valor de verdad a la variable v_{10} o a la variable v_{11}, es sólo en ese punto que se encontrará una contradicción. Luego al probar el valor contrario de la variable llegará también a una contradicción, logrando así desechar la rama del árbol y determinando que la fórmula es insatisfacible. En cambio, si la raíz del árbol de decisiones fuera v_{10} o v_{11}, llegaría a una contradicción en el segundo nivel sin necesidad de tomar ninguna decisión sobre el resto de las variables. Esta noción puede explotarse para generar un problema cuyo tiempo de resolución varíe drásticamente dependiendo del orden de las variables sobre las que se genere el árbol de decisión.

Si bien este caso se presenta sólo a modo de ejemplo, existen casos más complejos donde ciertas variables claves hagan a una fórmula insatisfacible, mientras que otro conjunto de variables resulte superfluo, y de igual manera, el tiempo total de ejecución dependerá del orden de las decisiones que se tomen. Si bien los *SAT-solver* actuales cuentan con varias heurísticas para la elección de la variable de decisión ([Sil99], [MMZ+01]) la variabilidad entre las distintas ejecuciones se mantiene.

La técnica de *restart* reconoce esta variabilidad y actúa para disminuir el tiempo típico de ejecución. Si bien la técnica descarta el árbol de decisión generado hasta el momento para recomenzar con el armado de un nuevo árbol, conserva las *cláusulas aprendidas*, esto hace posible podar caminos ya conocidos como insatisfacibles en el árbol anterior. A su vez, también mantiene información para los algoritmos de decisiones, lo que hace que el trabajo invertido no sea en vano y que la corrida subsiguiente sea también menor a lo que hubiera sido sin dicha información.

Para el mecanismo de *restart*, si bien su implementación no resulta tener mayores desafíos (ya que esta es relativamente sencilla), en qué momento realizar el *restart* resulta crítico. Una mala estrategia de *restart* podría empeorar el rendimiento del *SAT-solver* de manera notable.

Si por ejemplo se utiliza una estrategia muy conservadora y se elige recomenzar muy pocas veces, no se logrará obtener el potencial beneficio de la técnica, pero sí quizás su desventaja. Si por el otro lado se decidiera recomenzar muy seguido el algoritmo no podría avanzar y no se generaría nueva información, haciendo que el tiempo de ejecución aumentase. A su vez, es necesario también mantener la completitud del algoritmo. Es decir, asegurar que el algoritmo siempre termine.

Los *SAT-solver* tradicionales emplean distintas estrategias de *restart*, las más difundidas se basan en una progresión creciente para alguna restricción, como ser por ejemplo la

cantidad de decisiones o contradicciones. Al limitar la restricción usando una progresión creciente (y dado que se elige la variable a restringir de manera tal que exista un valor para el cual la resolución está asegurada) se conserva la completitud del algoritmo.

Por ejemplo, una estrategia de *restart* usada en los últimos años está dada por una progresión Luby [LSZ93] sobre la cantidad de conflictos [HH14]. Es decir, dada una secuencia de Luby (1, 1, 2, 1, 1, 2, 4, 1, 1, 2, 1, 1, 2, 4, 8 ...) llamaremos $l_i, i \in \mathbb{N}_{>0}$ al i-ésimo elemento de la secuencia. Dada una base $B > 1 \in \mathbb{N}$, el i-ésimo *restart* se dará luego de $C_i = B^{l_i}$ conflictos.

Como para toda fórmula se necesitan a lo sumo obtener tantos conflictos como hojas de un árbol de decisión y la altura del árbol está acotada por la cantidad de variables que tiene la fórmula. Entonces la cantidad de conflictos necesarios queda también acotada. Como pedimos que $B > 1$, y la secuencia $\{l_i, i \in \mathbb{N}\}$ no se encuentra acotada superiormente, entonces

$$\forall L \in \mathbb{N}, (\exists K \in \mathbb{N}, (B^{l_K} = C_K > L))$$

Entonces para toda fórmula existe una cantidad K, tal que a lo sumo luego de K *restart* el algoritmo tiene asegurado llegar a un resultado.

2.3. Herramienta Paralloy

Divide et Impera
– Julio César

Paralloy [Vis13] es un Sat-Solver distribuido desarrollado en el departamento de computación de la Facultad de Ciencias Exactas y Naturales. Inicialmente fue desarrollado para resolver problemas generados por el software Alloy [Jac06] utilizado para chequeo automático de modelos de especificación de software. *Paralloy* fue diseñado para ejecutar en un cluster utilizando un algoritmo de *Divide & Conquer*.

En la figura 2.2 se describen los componentes básicos del sistema. Este consta de un proceso *client* que es el encargado de coordinar el trabajo y de la lógica de la distribución del trabajo. En el resto del cluster se ejecutan procesos *worker* destinados a la generación y resolución de sub-problemas del problema original y un proceso *master* que es la puerta de entrada de comunicación entre el *client* y los *workers*. La comunicación *client* $\leftrightarrow$ *master* se realiza mediante TCP/IP, mientras que la comunicación *master* $\leftrightarrow$ *worker* se realiza mediante MPI [MF08].

Como ya se ha dicho *Paralloy* utiliza un mecanismo de *Divide & Conquer*, dividiendo cada problema en una cantidad de sub-problemas de la siguiente manera:

Sea P un problema de Satisfacibilidad en forma normal conjuntiva dado por:

$$P = \bigwedge_i \left(\bigvee_{l \in C_i} l \right)$$

Dado un conjunto C de variables, tal que $C \subset sop(P)$, se define el conjunto de conjuntos de literales:

$$\mathcal{L} = \{\{l = \neg v / v \in A\} \cup \{l = v / v \in B\}, \text{ tal que } A \in \mathcal{P}(C) \wedge B = C - A\}$$

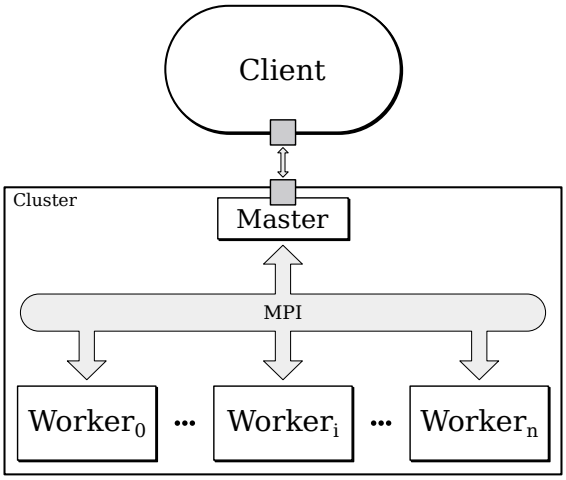

Fig. 2.2: Esquema de Componentes del sistema *Paralloy*.

Este conjunto contiene todos los conjuntos de combinaciones de valores verdaderos y falsos para las variables del conjunto C.

Para cada $\mathcal{L}_i \in \mathcal{L}$ se define el sub-problema P_i:

$$P_i = P \wedge \left(\bigwedge_{l_i \in \mathcal{L}_i} l_i \right)$$

Diremos que P_i es un sub-problema de P dado el conjunto de literales $\mathcal{L}_i$

Observación 18. *Ver que agregar un literal a un problema de esta forma, obliga a un sólo valor posible para la variable del literal. De otra forma, la valuación no sería capaz de satisfacer al nuevo problema definido.*

Entonces, dado un subconjunto de variables, se generan todas las combinaciones de valores de verdad para el conjunto de literales y para cada una de esas combinaciones se definen dichos valores de verdad para la fórmula de entrada. Por ejemplo, a partir de la fórmula $(v_1 \vee v_2 \vee \neg v_3 \vee v_4)$ tomando el conjunto unitario $\{v_1\}$ se generan los problemas: $(Verdadero \vee v_2 \vee \neg v_3 \vee v_4)$ y $(Falso \vee v_2 \vee \neg v_3 \vee v_4)$.

Definición 19 (Trail)**.** Se llama *trail*, al conjunto de literales $\mathcal{L}_i$ que diferencian a un sub-problema del problema original P.

En manera general, a partir de un problema P, se define:

$$Trail(P) = \emptyset$$

Y para todo P_i^{k+1} sub-problema de P_j^k mediante los literales $\mathcal{L}_i^{k+1}$:

$$Trail(P_i^{k+1}) = Trail(P_j^k) \cup \mathcal{L}_i^{k+1}$$

Dividiendo un problema P de esta manera en sub-problemas P_i y resolviendo cada uno:

- si se obtiene un modelo para algún P_i, dicho modelo será modelo del problema original.
- si no se encuentra ningún modelo para ningún sub-problema (todos los sub-problemas son insatisfacibles) el problema original es insatisfacible.

De esta forma es posible la resolución del problema original mediante la resolución de cada uno de los sub-problemas generados. A su vez, cada problema contiene $|C|$ menos variables, reduciendo potencialmente su dificultad. Y posibilitando la ejecución en paralelo de cada sub-problema. También, si se desea, es posible volver a dividir un sub-problema eligiendo un nuevo conjunto de variables no incluidas en el *trail* sobre el que realizar la nueva subdivisión, agregando al *trail* los literales elegidos para cada sub-problema generado. De esta manera, la partición de problemas resulta en un proceso iterativo que puede realizarse en todo momento sobre cualquier sub-problema con variables libres[8].

Para explicar el funcionamiento de *Paralloy* explicaremos por separado el funcionamiento de *client* y los *workers*. El proceso *master* es sólo la puerta de entrada que tiene el *client* para comunicarse con los *workers*, no hace más que enviar mensajes en ambas direcciones sin realizar ningún procesamiento relevante.

2.3.1. Proceso Client de la herramienta Paralloy

El *client* es el proceso con el que el usuario se comunica, es el encargado de la distribución de trabajo y de la recopilación de los resultados. El mismo puede ser ejecutado fuera del cluster de cómputo donde se ejecuten el *master* y los *workers*, es por esto que la comunicación *client* $\leftrightarrow$ *master* se realiza mediante TCP/IP.

Para comenzar, se debe definir el problema a resolver en formato DIMACS-cnf [DIM93]. El *client* entonces, distribuye el archivo cnf entre todos los *workers* y a continuación define un *worker* para que cree un problema raiz o *root* (no es más que un problema para comenzar la ejecución del sistema). A partir de este punto se seguirá una estrategia definida para la partición y distribución de problemas.

Si bien *Paralloy* puede ser configurado con distintas estrategias de resolución y distribución e inclusive puede ser configurado para trabajar en modo interactivo esperando directivas por linea de comando, para este trabajo se utilizó una estrategia totalmente automatizada que mantiene los siguientes invariantes:

- Si hay un *worker* ocioso con trabajos pendientes, elegir un trabajo pendiente y resolverlo.
- Si hay algún *worker* sin trabajo y trabajos pendientes en otro *worker*, transferir parte de estos trabajos al *worker* ocioso.
- Si algún *worker* está trabajando hace mucho tiempo[9], dividir el problema en sub-problemas.

Tanto luego de resolver un problema, como luego de realizar una división en sub-problemas el *worker* queda ocioso.

[8] Si no existieran variables libres el valor de verdad de la fórmula ya es conocido.

[9] Esta medida es configurable, pero se tiene en cuenta la cantidad de sub-problemas resueltos por *worker* por segundo que se logran y el tiempo de ejecución de cada problema.

Lo crucial en el proceso de división explicado en la sección 2.3 es definir el conjunto C de variables a elegir en cada caso. Ya que mientras una buena elección puede generar sub-problemas de menor complejidad, una mala elección puede no producir ningún cambio en cuanto a la complejidad de los sub-problemas (por ejemplo eligiendo variables superfluas que no lleven a ninguna contradicción). Pero es éste exactamente el mismo problema que se enfrentan los *SAT-solvers* secuenciales al elegir las variables de decisión. Es por esto que se decidió también tomar la medida de *actividad*[10] de las variables para priorizar una variable por sobre otra, tomando las variables de mayor actividad para generar los sub-problema.

Ahora bien, éste método es heurístico, por lo que también puede generar sub-problemas que sean fácilmente resolubles, *triviales*, y para estos problemas el costo de generar un sub-problema y su resolución (y potencialmente su distribución) no se vería amortizado durante su posterior ejecución. Es decir, para un problema cuya resolución consume un tiempo mínimo, debido al overhead de administración de los sub-problemas, su costo final pasaría a ser considerablemente mayor. Es por esto que antes de generar un sub-problema, se realiza un *microsolving*[11] para determinar si el problema es trivial o no. Si durante esa fracción de segundo el solver resuelve el sub-problema como UNSAT, este no se generará. En cualquier otro caso el sub-problema se genera para su posterior ejecución.

El sistema cuenta ya con una gran variabilidad en el tiempo de ejecución entre dos corridas para un mismo problema. Es por esto que para tener mayor control sobre la herramienta, se realizaron una serie de modificaciones:

- El tiempo de microsolving pasó a ser configurable.
- Además se agregó una cota superior para la cantidad de *workers* realizando división de problemas cuando hay poca frecuencia de problemas resueltos.
- También una cota inferior antes de abortar un problema para generar una división del mismo.
- Y una cota superior al tiempo que pueda llevar una división.

De esta manera se consiguió tener mayor control sobre el proceso y la estrategia de división del sistema.

En resumen, proceso *client* es el cerebro de la herramienta, es el responsable de coordinar y definir la estrategia de resolución y partición del sistema. Para poder llevar a cabo esta tarea de la mejor manera se vale de información sobre el estado de cada uno de los *workers* y sub-problemas (tanto activos como pendientes). Además de la división del trabajo, es también el *client* el que se encarga de supervisar la transferencia de sub-problemas entre *workers*, del proceso de resolución y recopilar los datos de SAT y UNSAT generados. Monitoreando el estado de cada *worker* y dando las directivas necesarias en el momento que lo crea necesario.

[10] La actividad de una variable se corresponde con la cantidad de conflictos en los que estuvo involucrada durante la resolución.
[11] Resolución de fracción de segundo.

2.3.2. Proceso Worker de la herramienta Paralloy

Los *worker* son las unidades que se encargan de la resolución y del proceso de división de sub-problemas. En la sección 2.3.1 se explica que el cliente es el encargado de coordinar el trabajo y recibir la información que los workers le proveen, pero el trabajo de mayor requerimiento de cómputo se delega en los *workers* del sistema.

Cada *worker* cuenta con una copia local del problema *CNF* que se desea resolver y con un archivo por cada sub-problema que posea localmente asignado. Estos archivos definen el *trail* del sub-problema y pueden contener también información adicional obtenida en el proceso de división[12].

Los *workers* son los encargados de generar (ante una directiva del *client*) la división de un problema, y una vez finalizada la división y creados los archivos para cada sub-problema, el *worker* notifica al *client* los problemas que han sido generados y queda en estado ocioso hasta recibir una nueva directiva.

Cuando el *client* lo decida puede hacer que se transfiera un sub-problema de un *worker* a otro. Para esto se comunica con el *worker* destino y le indica qué sub-problema debe conseguir y en qué *worker* se encuentra. Una vez que el *worker* recibe el mensaje la comunicación se realiza *worker* $\leftrightarrow$ *worker* por medio de *MPI*. Todo *worker* cuenta con un proceso autónomo que se encarga de este tipo de mensajes y transferencia, siendo posible así la transferencia aún cuando uno o ambos *workers* no se encuentren ociosos. Luego de transferir el archivo, se elimina la copia local del *worker fuente* y se genera otra en el *worker destino*.

Además de estas tareas que podrían considerarse administrativas, la principal función de estos componentes es la **resolución** o ***solving*** de los sub-problemas. Para esto es necesario la copia local del problema original y el archivo del sub-problema que contiene el *trail*. Una vez que se tienen estos dos archivos, se crea una instancia de un solver secuencial y se carga el archivo del problema original, realizando una resolución parcial al *trail* del sub-problema. Como *SAT-solver* secuencial, *Paralloy* utiliza MINISAT [NE03].

Uso de Minisat como Sat-Solver Secuencial

MINISAT es un SAT-Solver secuencial que se ha destacado en varias competencias de SAT-Solving. Esta herramienta cuenta con un método llamado *solve_limited* que toma una lista de literales y realiza una resolución *asumiendo que dichos literales son verdaderos*.

Para lograr esto MINISAT define estos literales como decisiones, cuya negación ya ha sido exhaustivamente explorada y descartada por insatisfacible dejando sólo un valor posible para dichas variables (los valores *asumidos*). De manera que la única posibilidad de que exista un modelo para el problema es un modelo que contenga estos literales. Así, si se el problema resulta SAT, lo será bajo la asunción de que los literales definidos son verdaderos. Y si por el contrario resultara $UNSAT$, querría decir que no es posible encontrar un modelo que satisfaga al problema bajo la asunción dada.

En el último caso, MINISAT cuenta además con la posibilidad de expresar esta imposibilidad en términos de la asunción. Es decir, obtener exactamente qué literales entre los que se definieron, hacen que el problema no tenga un modelo. Estos son aquellos literales que durante la resolución formaron parte de algún conflicto, este podría ser cualquier

[12] Para la configuración de *Paralloy* definida, dichos archivos sólo contienen el *trail* del sub-problema.

sub-conjunto, hasta el conjunto vacío, lo que implicaría que el problema es *UNSAT* independientemente de la asunción. Además, al realizar la resolución bajo asunciones de esta manera, toda cláusula aprendida que sea generada lo será en términos del problema original, manteniéndose válida en cualquier contexto.

Toda resolución que se lleve a cabo, puede tener dos resultados:

- Puede terminar (en *SAT* o *UNSAT*), en cuyo caso se le notificará esta situación al *client*, en el caso de *SAT* además se notificará el modelo encontrado
- o puede ser abortada por el proceso *client*, en este caso se interrumpirá al solver secuencial y se informará que se ha abortado con éxito la resolución.

En ambos casos el *worker* luego se encontrará en estado ocioso esperando futuras instrucciones.[13]

Es mediante esta cooperación *client* $\leftrightarrow$ *worker* que el sistema realiza la resolución total de un problema dado. Se puede observar que las cláusulas que un *worker* aprende durante la resolución de un sub-problema sólo son utilizadas de forma local en esa resolución. Uno de los objetivos de este trabajo es que dicha información (que puede ser relevante para la resolución de otro sub-problema) no se pierda y se encuentre accesible para el resto del sistema. Otro de los objetivos es también generalizar el proceso de *restart* que se lleva a cabo en los *SAT-solver* secuenciales (como por ejemplo MINISAT), haciendo que tenga injerencia global para probar su eficacia en este esquema distribuido.

[13] Típicamente, luego de abortar un problema se continua con la división del mismo, pero estos son dos mensajes independientes.

3. DESARROLLO

Los hermanos sean unidos
porque esa es la ley primera;
tengan unión verdadera
en cualquier tiempo que sea,
porque si entre ellos pelean
los devoran los de ajuera.
– José Hernandez, Martín Fierro

En el siguiente capítulo se explican las motivaciones, el trabajo llevado a cabo y las modificaciones que se realizaron sobre la herramienta *Paralloy* para probar las distintas hipótesis y buscar respuesta a las preguntas planteadas. Está dividido en tres secciones principales:

En la sección 3.1 se explica el primer enfoque de *learning* incorporado al sistema, se explica qué información se pretendía transmitir y se detalla el mecanismo implementado para la transmisión de información entre los distintos componentes del sistema, este mecanismo es el mismo que se utiliza en la versión final del la implementación de *learning*.

En la sección siguiente (3.2), se explica el procedimiento de *restart* global planteado para la herramienta y las distintas estrategias propuestas, explicando las modificaciones realizadas sobre la herramienta.

Y por último en la sección 3.3 se detalla el modelo de *learning* final implementado para este trabajo, que agrega las cláusulas aprendidas al conjunto de información transmitida, se discuten las decisiones tomadas sobre qué cláusulas transmitir, explicando también la administración del conjunto de cláusulas que se realiza en los distintos componentes del sistema.

Como se vio en el capitulo 2.3, *Paralloy* es un SAT solver que mediante una estrategia de divide & conquer logra resolver en forma paralela una instancia de un problema de Satisfacibilidad. Pero al paralelizar la resolución se pierde parte de la cooperación que existe en los *SAT-solver* secuenciales, es por esto que este trabajo busca atacar esta debilidad tratando de lograr mayor interacción entre las distintas unidades de resolución.

Durante el desarrollo de las nuevas funcionalidades se consideraron diferentes estrategias hasta llegar a la configuración final. El primer problema que se atacó fue la falta de cooperación entre los *workers* y la pérdida de la información generada en la resolución de los sub-problemas, con el objetivo de logar que el trabajo del sistema no sea sólo la suma de las partes sino también crear una sinergia que otorgue mayor valor al sistema total.

Información disponible en el sistema.

Dado que *Paralloy* resuelve distintos sub-problemas simultáneamente, existe información de una resolución que no habrá sido generada previa a la resolución de otro subproblema, mientras que hubiera estado disponible con una resolución secuencial.

Es decir, tomando como ejemplo el árbol de decisión de la figura 3.1; si se resolviera de manera secuencial en orden alfabético, la información sobre la resolución de A, estará disponible para resolver B y C. Mientras que si se cuenta con 3 nodos que resuelvan

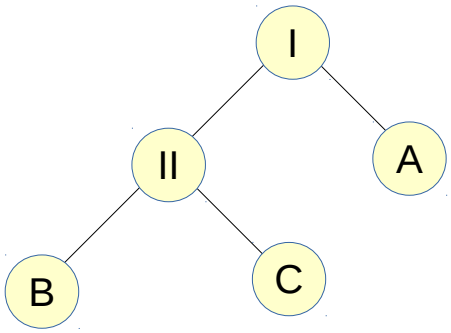

Fig. 3.1: Árbol de decisión.

simultáneamente A, B y C, dicha información no estará disponible para B ni para C. [1] Por otra parte, si primero se resuelve A y luego se resuelven simultáneamente B y C, la información sobre A se encontrará potencialmente disponible para la resolución de B y de C, aunque en este caso, la información obtenida a partir de la resolución de B no estará disponible para la resolución de C.

Este trabajo ataca el segundo caso, donde la información proveniente de la resolución de un sub-problema anterior se encuentra disponible para resoluciones futuras, siendo potencialmente útil para los sub-problemas subsiguientes.

3.1. Cláusula de conflicto

Al resolver un sub-problema condicionado a un *trail* cuyo resultado es *UNSAT*, *MINISAT* da la posibilidad de obtener el subconjunto de los literales del *trail* que generó que dicho sub-problema sea insatisfacible. Estos son los literales, que forman parte del *trail* y que formaron parte en algún conflicto durante la resolución.

Es decir, cuando se llega a una contradicción, se buscan las razones del conflicto (el conjunto de decisiones que lo generaron), como los literales del *trail* se definen como asunciones[2] pueden pertenecer a dicho conjunto. Los literales que pertenecieron a alguno de estos conjuntos son identificados como la razón de *UNSAT*[3], dado que son motivo suficiente para determinar la insatisfacibilidad del problema.

Si se toman estos literales y se genera una cláusula que sea la disyunción de su negación, se obtiene así una cláusula que debe mantenerse verdadera para que el problema pueda ser satisfacible. A esta cláusula la llamaremos **cláusula de conflicto**. (ver 2.3.2)

Si bien *Paralloy* está diseñado de manera tal que nunca generará dos sub-problemas con el mismo *trail*, es posible que tengan una porción del *trail* en común. A su vez, lo que hace insatisfacible a un problema condicionado al *trail* es un subconjunto de este, entonces todos los sub-problemas que contengan dicho subconjunto serán a también insatisfacibles. Esta información es la que busca capturar la cláusula de conflicto. Al agregar dicha cláusula al conjunto de cláusulas del problema, se logran descartar estos casos de manera inmediata.

[1] Parte de ella ni siquiera estará generada para el momento de la resolución en que se hubiera usado en el caso secuencial.

[2] variables que ya han sido exploradas con el valor de verdad opuesto. Ver 2.3.2

[3] Para más información sobre los métodos *solve_limited* y *conflict* ver [NE03]

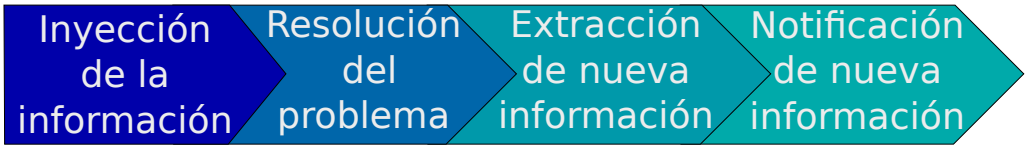

Fig. 3.2: Esquema de learning.

Las cláusulas de conflicto fue la primer información considerada para propagar por el sistema con el objetivo de eliminar sub-problemas de manera temprana. Al agregala como una cláusula extra al problema, aquellos sub-problemas que contengan en el *trail* una combinación de variables que la hagan insatisfacible resultarán en *UNSAT* instantáneamente. A su vez, aquellos problemas que contengan una porción de esta combinación, si luego durante la resolución se eligieran el resto de las variables como decisiones, la cláusula añadida generará un *UNSAT* de manera temprana, sin necesitad de realizar todo el procesamiento que fue necesario para la resolución de la cuál se dedujo el conflicto. Así la cláusula de conflicto actúa como una condición extra a chequear en el problema, reduciendo potencialmente su tiempo de resolución.

3.1.1. Transmisión de la información

Una vez obtenida información relevante a partir de la resolución de un sub-problema, el objetivo es lograr que esta información se encuentre disponible en la totalidad de los *workers* del sistema. Además es deseable minimizar la cantidad de mensajes que se transmiten por el sistema a fin de evitar la congestión de la red, manteniendo también el control sobre qué información y cuándo se transmite.

Consideraciones previas

Una de las decisiones tomada es que la información obtenida de una resolución se mantendrá hasta el fin de la ejecución de la herramienta, dado que no se encontraron evidencias concretas de que dicha información pierda relevancia. A su vez, queda fuera del alcance del trabajo la implementación de un mecanismo de eliminación y/o clasificación de la información. Es por esto que se optó por restringir la información transmitida manteniéndola hasta el fin de la ejecución, descartando el resto de la información generada.

Si bien la inyección de información durante la ejecución de un *SAT-solver* es posible, no es trivial, y dado que *Paralloy* realiza resoluciones relativamente breves[4], se estima que el porcentaje de información generado durante este tiempo será pequeño en comparación con la información ya generada en resoluciones previas por la totalidad del sistema(recordar que la información generada se mantendrá hasta el final de la ejecución).

Otro punto a tener en cuenta es que una de las ventajas de *Paralloy* consta en utilizar un *SAT-solver* secuencial para la resolución de los sub-problemas, pretendiendo del mismo una cantidad mínima de requerimientos. Esta fue una decisión del diseño original que brinda la posibilidad de avanzar paralelamente al desarrollo de los solvers secuenciales, pudiendo aprovechar las nuevas técnicas que se desarrollen en un futuro. Es por esto que es deseable mantener a *Paralloy* desacoplado del solver que se utilice.

[4] Recordar que en el caso de que una resolución se extienda en el tiempo, *Paralloy* detiene dicha ejecución y divide el problema en sub-problemas.

Esquema de transmisión de información propuesto

Considerando todos estos puntos, se propuso el esquema descripto en la figura 3.2, que explicaremos más en detalle a continuación.

Desde el punto de vista de los *workers*, existen dos pasos fundamentales:

- antes de comenzar la resolución de un sub-problema se inyecta la información disponible hasta el momento
- al finalizar la resolución de un sub-problema se extrae las información generada.

De esta forma se logra a su vez una mínima intrusión en el *SAT-solver* a utilizar. Notar que la administración del flujo de la información no es una responsabilidad del *worker*, sino que se delega en el *client*, para así lograr mayor control sobre la misma.

Si bien centralizar la información en un componente conlleva riesgos como la creación de un posible cuello de botella, dado que la información se transmitirá sólo antes y después de la resolución de un problema, esto no genera problemas para las redes de los clusters actuales. Además simplifica la administración de la información a transmitir, ya que el *client* puede mantener un registro de cual es la información que cada *worker* conoce y sólo transmitir la faltante. Y al estar centralizado, otorga un mayor control para la manipulación y el estudio del sistema.

Entre las opciones para la distribución del aprendizaje se consideró, por un lado, una estrategia descentralizada donde cada *worker* transmita la nueva información obtenida al resto de los *workers*, y la centralización de la información sumada a un mecanismo de broadcast desde el *client*. Ambos enfoques, al ser asíncronos desde el punto de vista del receptor, no aseguran que toda la información se encuentre disponible al iniciar un sub-problema. Como uno de los objetivos del trabajo es analizar el impacto del *learning*, se optó por enviar la información en el mismo mensaje para la resolución de un sub-problema. Si bien tiene un costo potencialmente más alto en tiempo, ya que podría realizarse de manera asíncrona, la ventaja de dicho protocolo es que asegura que el *worker* esté actualizado en cuanto a la información disponible en el sistema al comenzar una resolución.

A continuación se explicarán en mayor detalle las responsabilidades del *client* en este proceso.

3.1.2. Rol del proceso Client en el esquema de Learning

Luego de la resolución de cada sub-problema, el *client* recibe una nueva cláusula de conflicto proveniente del *worker*, y es el encargado de mantener actualizado el conjunto de cláusulas para cada uno de los *workers* del sistema. Para administrar esta información el *client* mantiene el conjunto de las cláusulas de conflicto ordenadas en el tiempo, conociendo para cada *worker* hasta qué punto temporal se encuentra actualizado. De esta manera se mantiene registro de qué porción de la información se debe transmitir a cada *worker*. Una vez transmitida dicha información a un *worker* dado y obtenida la confirmación de la recepción del mensaje, se actualiza este registro.

3.1.3. Rol del proceso Worker en el esquema de Learning

Sabemos que la principal función del *worker* es la resolución de sub-problemas, sin embargo para resolver el problema planteado, fue necesario sumarle la responsabilidad de

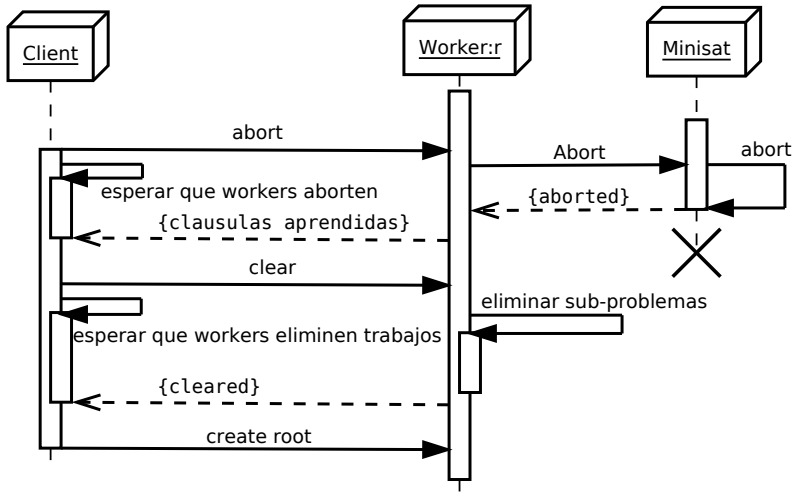

Fig. 3.3: Diagrama de secuencia de interacción *client* ↔ *worker* de *restart*.

inyectar y extraer la información sobre las cláusulas de conflicto. Resultando en que cada *worker* realice las siguientes acciones adicionales:

- Recibir las cláusulas de conflicto enviadas por el *client* y almacenarlas localmente.
- Inyectar al solver la totalidad de las cláusulas almacenadas antes de comenzar con la resolución de cada sub-problema.
- Extraer la cláusula de conflicto de cada sub-problema que resulte *UNSAT* y transmitirlo al *client* (esto se realiza junto con el mensaje de *UNSAT*).

Luego de la resolución de un sub-problema, la instancia de *MINISAT* utilizada queda inutilizable. Es por esto que para la siguiente resolución se debe crear una nueva instancia; siendo necesaria la carga del problema original y el *trail* a considerar, debiendo el *worker* inyectar la totalidad de las cláusulas de conflicto.

3.2. Proceso de Restart distribuido

Actualmente la mayoría de los *SAT-solver* secuenciales implementan *restart* (Ver sección 2.2.3). Esta es una técnica que ha dado muy buenos resultados prácticos y que se pierde en el enfoque original de la herramienta.

Sabemos que *Paralloy* utiliza *MINISAT* para resolver los sub-problemas, y *MINISAT* realiza internamente *restarts*, pero estos sólo tienen injerencia local. Si pensamos la partición de sub-problemas como un árbol, este nunca cambiará durante la resolución total del problema, y dicha resolución se verá entonces atada a las particiones que se hayan tomado al inicio. Es por esto que se decidió implementar un mecanismo de *restart* global, evitando así que decisiones erradas al comienzo de la resolución afecten de manera permanente el rendimiento de la herramienta.

En sí, el proceso de detener la ejecución y recomenzar desde el inicio (manteniendo la información recopilada hasta ese momento) no resulta un problema complejo a resolver.

Para lograr esto se implementó una directiva en el *client* llamada *restart*, dicho proceso se puede observar en el diagrama de secuencias 3.3.

En lineas generales, es el *client* el que transmite un mensaje de *ABORT* a todos los *workers* del sistema, tras el cual cada *worker* detiene la ejecución si estuviera ejecutando, y también detiene cualquier transferencia de sub-problemas que pudiera estar ocurriendo, quedando en estado ocioso esperando un próximo mensaje. Una vez detenidos los *workers*, el *client* envía un mensaje de *clear* para que borren los sub-problemas que pueden tener localmente para no generar conflictos con los nuevos sub-problemas que se generarán.

Una vez que todos los sub-problemas generados durante la resolución previa han sido eliminadas en todos y cada uno de los *workers*, se recomienza con una nueva ejecución manteniendo la información previamente recopilada. Dicha información continúa siendo válida, dado que es inherente al problema y no está atada a ninguna particularidad de la ejecución.

Como ya se detalló en la sección 2.2.3, el aspecto más importante del *restart* es decidir bajo qué condiciones se realiza el mismo. Con este objetivo se implementaron diferentes estrategias basadas en un indicador de progreso, limitado por una secuencia estrictamente creciente. Se implementaron las estrategias utilizando una progresión geométrica[5] y una progresión Luby[6], ambas totalmente configurables en cuanto a la base y el exponente. Definiendo a la vez tres indicadores diferentes: la cantidad de aborts, la cantidad de unsat y la longitud de las colas de trabajos. A continuación se explicaran más en detalle los motivos para utilizar estos indicadores.

3.2.1. Estrategia de Restart por Unsat

Una estrategia de *restart* ampliamente adoptada por los *SAT-solvers* secuenciales se trata de limitar la cantidad de *UNSAT* utilizando una progresión creciente. En el caso de *Paralloy* cuando se habla de *UNSAT* se consideran los *UNSAT* globales, es decir, la cantidad de sub-problemas que resultaron en *UNSAT*.

Al igual que en el caso secuenciales, este enfoque se basa en que existe una cantidad de *UNSAT* mínima necesaria para resolver un problema dado. Es decir, una forma de dividir el problema en sub-problemas de forma tal que cada sub-problema sea suficientemente sencillo de resolver y no sea necesario subdividirlo para su resolución. La motivación detrás de este enfoque es buscar esta forma óptima de subdivisión y descartar divisiones que no decrementen la complejidad de los sub-problemas que genere.

A su vez, existe una cantidad máxima de *UNSAT* para la cuál se puede asegurar que el problema se resolverá. En particular, no se pueden generar más de $2^{|sop(P)|}$ sub-problemas distintos, siendo una cota superior para la cantidad de hoja que puede contener un árbol de decisión. Por lo que la cantidad de *UNSAT* siempre será menor o igual a dicho número (a lo sumo existe un *UNSAT* por hoja del árbol de decisión), de esta forma, si se lo limita mediante una progresión estrictamente creciente, se asegura la completitud del algoritmo.

[5] Para un exponente e y una base b se define una progresión geométrica $S_b^e = (t_1, t_2, ...)$ con $t_i = b * e^i$

[6] Para un exponente e y una base b se define la serie de Luby $S_b^e = (t_1, t_2, ...)$ con

$$t_i = \begin{cases} b * e^{k-1}, & \text{si } i = 2^k - 1 \\ t_{i-2^{k-1}+1} & \text{si } 2^{k-1} \leq i < 2^k - 1 \end{cases}$$

Lo que genera: $S_b^e = (b, b, b * e, b, b, b * e, b * e^2, b, b, b * e, b, b, b * e, b * e^2, b * e^3, b...)$ [LSZ93]

3.2.2. Estrategia de Restart por Abort

Otro indicador de progreso que se decidió tomar es la cantidad de veces que el sistema realiza un *ABORT*. Este enfoque está relacionado con el antes mencionado, ya el sistema decide abortar un sub-problema en las siguientes dos situaciones: el sub-problema está siendo procesado hace demasiado tiempo[7], o bien porque existen *workers* en estado ocioso. El primer caso es un indicador de que el sub-problema generado resulta de complejidad similar a su ancestro, mientras que el segundo habla de una subutilización de los recursos disponibles.

Al igual que en el caso del *restart* por *UNSAT*, en este caso existe una cota superior para la cantidad de *ABORT* que se pueden realizar en el sistema. Esta es la cantidad de nodos internos del árbol de decisión, que está acotada por $2^{|sop(P)|} - 1$. Así que también, al limitar este indicador por una progresión estrictamente creciente se mantiene la completitud del algoritmo.

3.2.3. Estrategia de Restart por longitud de colas de trabajo

Otro limitante que se consideró fue la longitud de las colas de trabajo de los *worker*. Cuando *Paralloy* realiza una división de un sub-problema, los problemas generados son asignados a la cola de trabajo que cada *worker* tiene. La longitud de estas colas representan el trabajo pendiente del sistema, son todos los problemas que se deben resolver para poder asegurar llegar a un resultado[8].

Si las colas ganan mucha longitud es porque la velocidad de generación de sub-problemas es mucho mayor que la resolución de los mismos. Esto puede ser tanto un efecto causado por la dificultad intrínseca del problema a resolver o puede ser indicativo de una mala elección de variables para las divisiones que generan problemas de complejidad.

En este caso, este indicador está acotado por la suma de los anteriores. Al representar la cantidad total de sub-problemas totales generados; esta métrica está acotada por la cantidad de nodos del árbol de decisión ($2^{|sop(P)|+1}$). Al ser una medida acotada, la completitud del algoritmo se mantiene si se utiliza una progresión estrictamente creciente.

3.2.4. Experimentación preliminar sobre Restart

Se realizaron una serie de experimentos preliminares para ver el efecto que tienen estas estrategias sobre el tiempo total de ejecución, comparando el promedios del tiempo total de ejecución para dos problemas.

La figura 3.4 muestra los resultados para una estrategia de *restart* sobre la cantidad de *UNSAT* para una progresión geométrica de base 2 y constante 100. A partir de las mediciones del tiempo de ejecución, se obtuvieron la media y la varianza muestral, calculando a partir de estos valores los intervalos de confianza (de nivel 0.9) para las siguientes configuraciones:

- Caso base: Sin ningún tipo de *learning* ni *restart*.
- Restart: Utilizando la estrategia de *restart* descripta, sin *learning*.

[7] El limite de tiempo está dado en comparación relativa al tiempo consumido por el resto de los sub-problemas en ejecución.

[8] En el caso de los problemas *SAT* la resolución puede completarse antes de resolver todos los sub-problemas generados

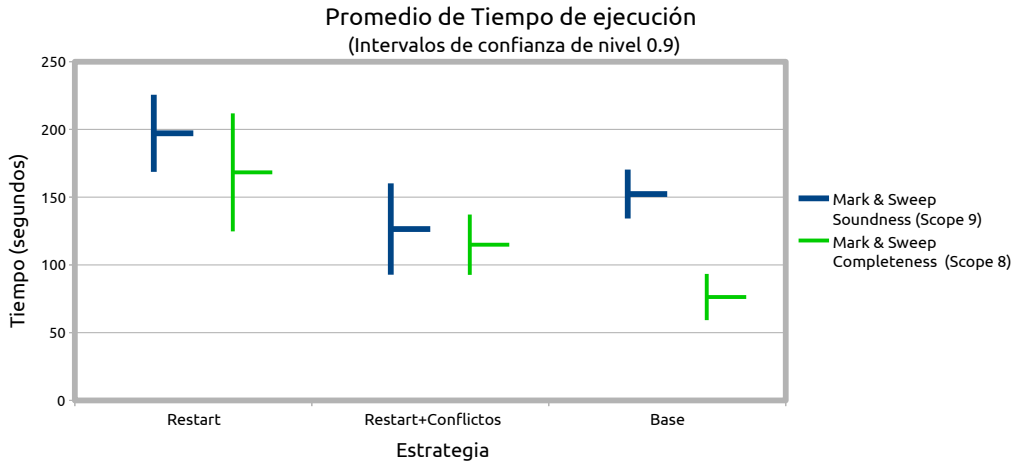

Fig. 3.4: Experimentación preliminar para una estrategia de Restart por *UNSAT*.

- Restart+Conflictos: Utilizando la misma estrategia de *restart* y también utilizando *learning* sobre las cláusulas de conflicto.

Se puede ver como el tiempo de ejecución se ve perjudicado cuando se utiliza sólo la técnica de *restart* para ambos problemas. Cuando se lo combina con *learning* se pueden observar efectos positivos en uno de los experimentos, mientras que en el otro, si bien el tiempo de ejecución comparado con el caso en el que se utilizan sólo *restart* es menor, el efecto no es suficiente para mejorar el tiempo base de la herramienta. En ambos casos cuando se observan los intervalos de confianza, se ve que los resultados no son concluyentes, pero se tratan sólo de experimentación preliminar.

Vale aclarar que en los experimentos utilizando *restart*, el sistema realizó en promedio 5 *restarts* por corrida.

3.3. Learning de cláusulas aprendidas

La división del espacio de búsqueda que realiza *Paralloy*, si bien ayuda a que el tiempo de ejecución sea sustancialmente menor que el tiempo secuencial para algunos casos, en la práctica se pudo observar que la diferencia no es tan grande como la esperada. Una de las posibilidades de este fenómeno puede deberse a que los solvers secuenciales cuentan con un algoritmo de aprendizaje [SS96] que hace que la resolución de un sub-problema se vuelva más rápida utilizando la información generada durante la resolución de un sub-problema anterior.

La idea detrás del algoritmo de aprendizaje es registrar para cada conflicto, los literales del problema que lo ocasionaron y a partir de ellos generar una *cláusula aprendida*[9]. Con dicha información, si luego llegara a presentarse una configuración con los mismos literales, se evita la propagación de las cláusulas hasta llegar al conflicto descartando esta rama de manera prematura.

Estas cláusulas en el enfoque original de *Paralloy* son descartadas al finalizar cada resolución.[10] El objetivo de la funcionalidad propuesta es determinar un subconjunto de

[9] El *learning* se explica en 2.2.2

[10] Una estrategia disponible de *Paralloy* transfiere las cláusulas aprendidas de un nodo padre a sus hijos

estas cláusulas y propagar esa información a lo largo del sistema para que sea utilizada en la totalidad de las resoluciones siguientes. Esperando con esta modificación disminuir el tiempo de resolución de cada sub-problema.

3.3.1. Learning de cláusulas aprendidas en Paralloy

Como ya se explicó en la sección 2.2.2 los solvers secuenciales se valen del aprendizaje para podar el árbol de decisiones que generan. A su vez, las *cláusulas aprendidas* generadas durante el *solve_limited* de *MINISAT* (2.3.2) son definidas en base a las variables originales del problema, lo que las hace globalmente válidas, esto permite aprovecharlas para la resolución de otro sub-problema.

Antes de inundar el sistema con las cláusulas hay que tener en cuenta el efecto que esto puede traer. Para los *SAT-solver* secuenciales, el algoritmo de *BCP* representa cerca del %90 del tiempo de ejecución. A su vez, la complejidad de dicho algoritmo está atada a la cantidad de cláusulas sobre las que se realice la propagación, entre las que se encuentran las cláusulas aprendidas. Estudios como [AS09] fueron dedicados al análisis de los efectos que tienen las cláusulas aprendidas sobre el rendimiento de los *SAT-solver*, logrando mejoras para problemas *SAT* cuando se realizan purgas agresivas sobre las *cláusulas aprendidas*, lo que evidencia que una cantidad excesiva de *cláusulas aprendidas* resulta perjudicial. Esto también conlleva un efecto negativo para el caso de los problemas *UNSAT*, dada la perdida de información que estas purgas generan.

Hay que tener en cuenta también que al ser un sistema distribuido, estos efectos serían potenciados, ya que las cláusulas aprendidas serán generadas no por una única fuente sino que por tantas como *workers* tenga el sistema. Haciendo que el conjunto de cláusulas aprendidas aumente sustancialmente.

Debido al efecto negativo sobre el rendimiento que puede generar una cantidad excesiva de *cláusulas aprendidas*, los *SAT-solver* secuenciales implementan un mecanismo de purga del conjunto de *cláusulas aprendidas*, descartando en tiempo de ejecución un subconjunto de las *cláusulas aprendidas* según algún criterio definido[11]. Cuando la cantidad de cláusulas aprendidas supera cierto punto, el costo de realizar *BCP* sobre las mismas es mayor que la ventaja que se obtiene de ellas. Debido a esto, determinar qué cláusulas eliminar y qué cláusulas mantener no es un problema menor, ya que el costo de cualquier algoritmo de priorización tiene que ser menor que la ganancia que genere.

Es por esto muchos *State-of-the-art SAT-solver*, como *MINISAT*, simplemente priorizan las cláusulas de longitud 1 y 2, manteniéndolas durante toda la ejecución, mientras que para el resto de las cláusulas, se deshace de la mitad con menor actividad[12] cuando el conjunto supera cierto límite establecido.

En este trabajo, se decidió usar también una priorización por longitud, haciendo configurable la longitud máxima de las *cláusulas aprendidas* a transmitir entre los *workers*. Se utiliza la longitud como heurística para determinar su importancia, ya que a menor longitud, el efecto que tiene es mayor: por ejemplo, las cláusulas unitarias eliminan variables libres, ya que dejan sólo un posible valor de verdad. Mientras que las binarias, al tomar una decisión sobre una de sus variables dejan a la otra definida.[13].

al momento del *split*, pero esto experimentalmente no tuvo resultados notablemente favorables [Vis13].

[11] Para descartar *cláusulas aprendidas* cada *SAT-solver* puede implementar su propio criterio

[12] El grado de actividad de una cláusula depende de la cantidad de veces que fue usada durante el algoritmo de *BCP*.

[13] Si bien esta medida es heurística, ha probado ser de utilidad en los *SAT-solver* secuenciales como

En cuanto a la purga de las *cláusulas aprendidas*, se decidió no implementar ningún mecanismo de purga global y se delegó en el solver que se utilice (en este caso *MINISAT*) la responsabilidad de realizar una purga local sobre las mismas. Para lograr esto, las *cláusulas aprendidas* no se agregan como cláusulas del problema, sino que se incorporan al conjunto de *cláusulas aprendidas* interna del solver.

El esquema que se utilizó para las *cláusulas aprendidas* es similar al utilizado para las cláusulas de conflicto. Cada *worker* al finalizar una ejecución de un solver (ya sea que finalice por UNSAT o por ABORT); recopila las *cláusulas aprendidas* que cumplan con la restricción de longitud configurada; remueve las que ya eran conocidas previas a la ejecución del solver para determinar cuales son las *nuevas* y las transmite al *client* junto con la cláusula de conflicto. Cuando el *client* recibe esta información, la registra para luego comunicarla al resto de los *workers* del sistema[14]. En los diagramas de secuencia se detalla esta interacción (Fig. 3.5 y Fig. 3.6).

3.3.2. Rol del proceso Client en el Learning distribuido

Desde el punto de vista del *client*, la administración de las *cláusulas aprendidas* resulta más complejo que la administración de las cláusulas de conflicto. Este proceso continúa siendo el encargado de recibir la nueva información obtenida por cada *worker*, y transmitir al resto de los *workers* el conjunto de *cláusulas aprendidas* que desconozca. Con este fin mantiene el conjunto de todas las *cláusulas aprendidas* recopiladas por los diferentes *workers* (I_{global}). Contando además con un registro por *worker*, donde se guarda la información que se le debe comunicar a dicho *worker* (I^{worker}_{enviar}). Este registro es un diccionario $D : W_i \rightarrow I^{W_i}_{enviar}$.

Cuando un *worker* (W_O) entrega nueva información ($I^{W_0}_{aprendida}$):

1. Se remueve dicha información de la información a comunicarle al *worker*.

   $$I^{W_0}_{enviar} = I^{W_0}_{enviar} - I^{W_0}_{aprendida}$$

2. Se remueve además la información ya conocida de las novedades.

   $$I_{nueva} = I^{W_0}_{aprendida} - I_{global}$$

3. Se agrega la nueva información a la información global.

   $$I_{global} = I_{global} \cup I^{W_0}_{aprendida}$$

4. Se agrega la información nueva al conjunto de información a comunicar para los *workers* restantes.

   $$\forall W_i \neq W_0, I^{W_i}_{enviar} = I^{W_i}_{enviar} \cup I_{nueva}$$

De esta manera se mantiene actualizado el registro de la información que se debe enviar a cada uno de los *workers*, a la vez que se minimiza la información redundante que se transmite. Esta transmisión de información se realiza junto con el mensaje *solve*, ya que es cuando se desea que los *workers* contengan toda la información disponible.

MINISAT

[14] Esto se realiza junto con la orden de *solve*.

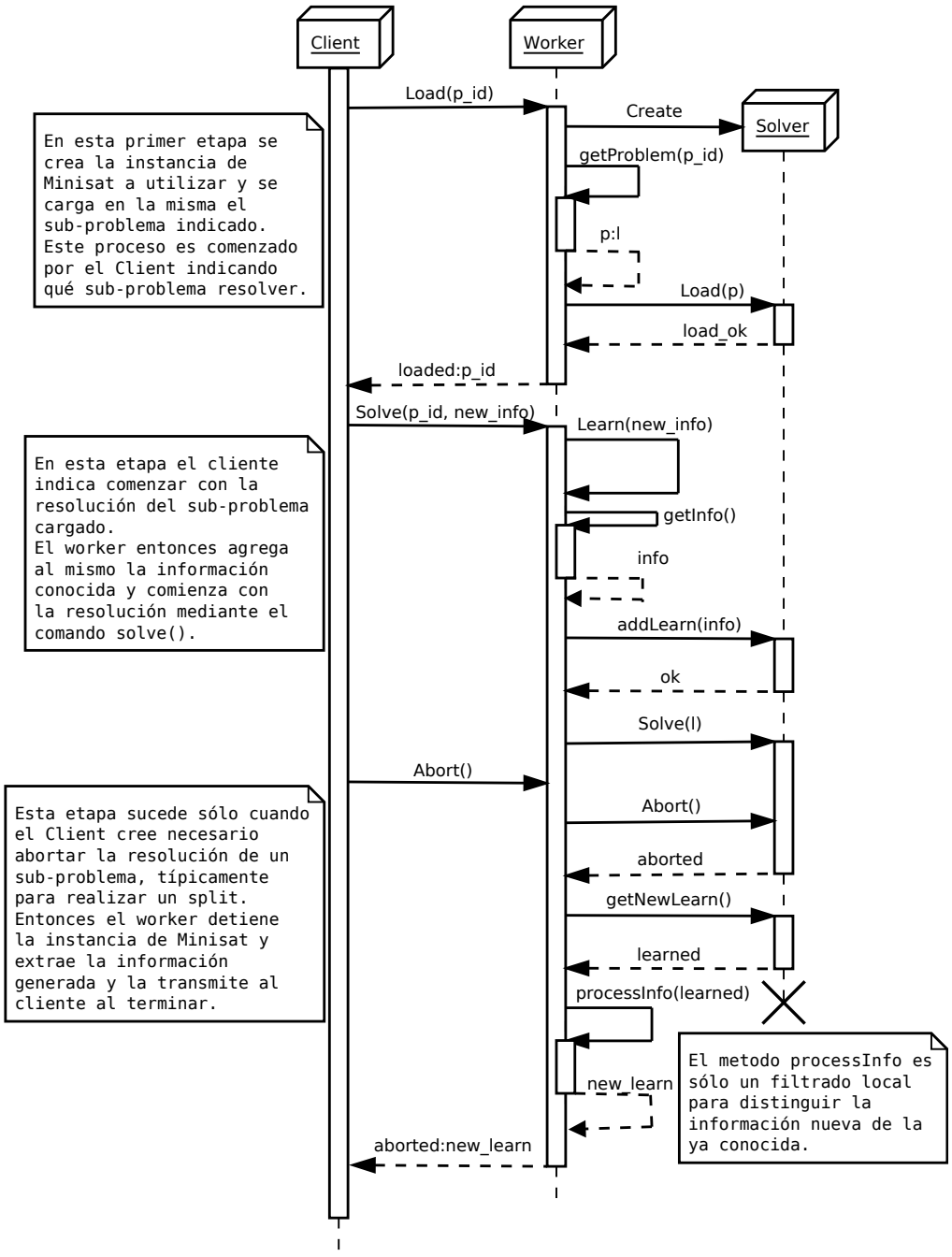

Fig. 3.5: Diagrama de secuencia de la carga y resolución de un sub-problema (Abort).

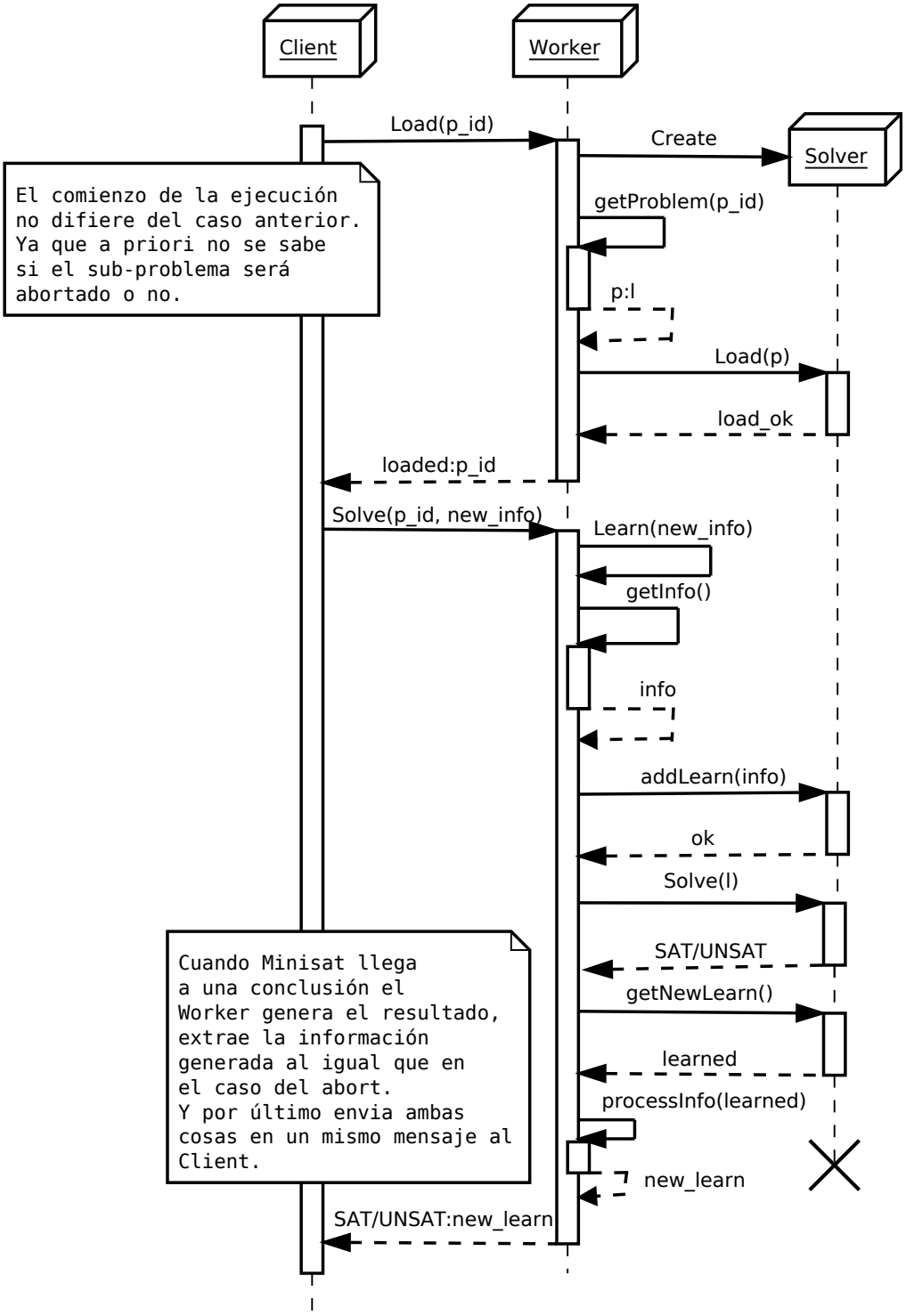

Fig. 3.6: Diagrama de secuencia de la carga y resolución de un sub-problema (Solving finalizado).

3.3.3. Rol del proceso Worker en el Learning distribuido

Cada *worker* realiza tres acciones:

- Recibe las cláusulas aprendidas por el resto de los *workers*.
- Extrae las cláusulas aprendidas durante la resolución de cada sub-problema.
- Inyecta al solver la totalidad de las cláusulas aprendidas antes de comenzar a resolver un sub-problema.

Para realizar esto cada *worker* requiere conocer el conjunto de *cláusulas aprendidas* (tanto las propias como las que el *client* le transmita) localmente. Cuando llega un nuevo mensaje con cláusulas, se agregan a dicho conjunto, y al extraer las *cláusulas aprendidas* del solver una vez finalizada una resolución (ya sea por *ABORT*, *SAT* o *UNSAT*), debe eliminar de dicho conjunto las *cláusulas aprendidas* previamente conocidas. De esta manera se evita transmitir cláusulas ya conocidas por el sistema.

Para poder lograr incorporar al *SAT-solver* el conjunto de *cláusulas aprendidas* se incorporaron dos métodos a *MINISAT*; uno para la incorporación de los *facts*[15] y el segundo para agregar las cláusulas aprendidas[16].

[15] Cláusulas unitarias, de longitud uno.
[16] Claúsulas de longitud mayor o igual a dos.

4. EVALUACIÓN EXPERIMENTAL

En el siguiente capítulo se explican los experimentos realizados para analizar el efecto de las modificaciones hechas sobre la herramienta y su efecto en el rendimiento. En la sección 4.1 se realiza una breve introducción sobre la variabilidad del tiempo de ejecución sistema y se presenta la configuración tanto del *Hardware* como de la herramienta a utilizar en la experimentación. Luego, en la sección 4.2 se explican los distintos aspectos que se busca evaluar en la experimentación y los problemas que se consideraron. Finalmente, en la sección 4.3 se detallan los experimentos y los resultados obtenidos.

4.1. Marco experimental

En esta sección se establecen los lineamientos generales para la configuración utilizada.

4.1.1. Dispersión del tiempo de ejecución

Una de las características de los *SAT-solvers* CDLL es la gran variabilidad en cuando al tiempo de ejecución para un mismo problema. Por el enfoque mismo del algoritmo, resultan sistemas altamente inestables, donde una pequeña diferencia en alguna decisión modifica notablemente el rendimiento total. Esta inestabilidad se debe en parte a que el árbol de decisiones es generado aleatoriamente, y como se explicó en secciones anteriores, el tiempo de resolución para dos árboles de decisión diferentes puede diferir sustancialmente.

Para subsanar este problema, los *SAT-solvers* definen una semilla para su algoritmo de generación de aleatoriedad, a partir de esta semilla los valores generados son siempre los mismos, logrando así un tiempo reproducible para la resolución para un problema dado. Incluso existe una variedad de *SAT-solvers* paralelos que utiliza la técnica de *portfolio*[1] donde la implementación de cada *SAT-solver* es la misma, pero se utiliza una semilla diferente, explotando de esta forma la aleatoriedad ya mencionada.

Paralloy, al utilizar un *SAT-solver* tradicional para la resolución de cada sub-problema, también resulta en un sistema altamente inestable en cuanto al tiempo de ejecución. Sumado a esto, la estrategia de *split* utilizada toma en consideración el tiempo de ejecución de los distintos sub-problemas; es decir, la diferencia entre decidir si se debe detener el sub-problema A o el sub-problema B en un momento dado, se define tomando en cuenta el tiempo de ejecución de ambos sub-problemas[2]. Estas decisiones generan árboles totalmente diferentes, lo que hace aún más marcada la gran dispersión en el tiempo de resolución para un problema dado.

En el caso del sistema distribuido, al contrario que en secuencial (donde la fuente de inestabilidad se debe a la generación de decisiones al azar, por lo que definiendo un proceso repetible para estas decisiones, se obtiene un sistema sólo dependiente del problema a resolver), existen factores externos como ser por ejemplo la transmisión de mensajes que generan diferencias en el comportamiento del sistema. Como el tiempo de transmisión de

[1] un conjunto de distintos *SAT-solvers* secuenciales que se ejecutan en paralelo tomando el resultado del primero que finalice.

[2] Se dividirá al problema con mayor tiempo de ejecución

mensajes es dependiente de, entre otros, factores físicos inherentes al Hardware, dichas variaciones resultan inevitables. Lo que resulta en gran variabilidad del tiempo de resolución de la herramienta para dos problemas con exactamente la misma configuración.

Es por esto que para obtener resultados confiables es necesario realiza un análisis estadístico para evaluar el rendimiento de la herramienta. Haciendo necesaria la repetición de un mismo experimento[3] para obtener la media esperada sobre el tiempo de ejecución y poder determinar el efecto (si lo hubiera) de las distintas estrategias propuestas.

4.1.2. Hardware utilizado durante la experimentación

Este trabajo utilizó recursos del CCAD de la Universidad Nacional de Córdoba, en particular el Cluster Mendieta, el cual forma parte del SNCAD del MinCyT de la República Argentina.

Se utilizaron los nodos del Cluster que cuentan con la siguiente configuración:

- Dos procesadores Intel Xeon E5-2680 v2 de diez cores de 2.8 GHz.
- Memorias RAM de 64 GiB DDR3 a 1600 MHz.
- Almacenamiento NAS de 26 TiB de capacidad.
- Conectividad Infiniband QDR y doble Gigabit Ethernet (NFS, administración).
- Switch Infiniband Mellanox MTS3600R de 36 puertos.

Utilizando una configuración de hasta 18 *workers* por nodo.

4.1.3. Configuración del sistema

Como ya se explicó, la herramienta *Paralloy* es altamente configurable, a su vez este trabajo incorpora nuevas opciones y configuraciones para las distintas estrategias, tanto de *restart* como de *learning*.

A su vez, el foco del trabajo es extender el poder de la herramienta para problemas de complejidad media-alta, lo que hace que los problemas elegidos para analizar el rendimiento tengan tiempo típico de resolución alto[4]. Además, debido a la varianza del tiempo de resolución para un mismo problema, es necesario repetir un mismo experimento para obtener una medida estadísticamente representativa. Es por esto que para obtener una medida del rendimiento de una configuración dada son necesarias decenas de horas de cómputo.

Dado que la cantidad de horas con las que se cuenta para la realización de este trabajo es limitada, ha sido necesario acotar la cantidad de configuraciones analizadas. Tomando en cuenta experimentación preliminar y trabajos relacionados se establecieron prioridades sobre las configuraciones y problemas considerados, a continuación se detallan las mismas.

[3] Tanto el problema a resolver como la configuración a utilizar
[4] Más de dos horas de resolución

4.1.4. Configuración base de Paralloy

Para poder analizar el efecto de las funcionalidades implementadas sobre *Paralloy*, fue necesario establecer una configuración base y así poder realizar un análisis comparativo. Dicha configuración no utilizará técnicas de *restart* global ni *learning* distribuido. Para determinarla se tomó como referencia el trabajo inicial de desarrollo del sistema [Vis13], concluyendo en la siguiente configuración:

- Estrategia de resolución: BFSStrategy. Esta es la estrategia que gobierna al sistema, define cuando realizar un *split* y cómo distribuir los trabajos entre los *workers*. Se encarga de la detección de nodos en el sistema ociosos y de la frecuencia de resolución. BFSStrategy es la estrategia descripta previamente en este trabajo.

- Learnstrat, Learnargs: vacío. Estos parámetros definen la estrategia a utilizar en cuanto a qué información se transmite a los sub-problemas generados a partir de un sub-problema particionado. Debido a que no se lograron resultados notablemente favorables realizando esta transmisión, y para evitar perturbaciones al medir la eficacia del nuevo mecanismo de *learning*, se decidió no transmitir ningún tipo de información a partir de la división de sub-problemas[5].

- nvars: 70. Este parámetro define una cota inferior para la cantidad de sub-problemas a generar en un *split*. Nótese que al elegir una variable extra en principio se duplican la cantidad de sub-problemas generados hasta el momento, pero a su vez, luego de realizar el *microsolving* cierto subconjunto de problemas pueden ser descartados. Esta variable define un piso para la cantidad de problemas a generar. El *split* no sólo está limitado por esta variable sino por *split_time_limit*.

- splitstrat: newvsids. Se decidió utilizar esta estrategia entre todas disponibles no sólo porque demostró obtener mejores resultados para *Paralloy*, sino que también ha demostrado tener buenos resultados en los *SAT-solver* secuenciales. La misma consta en elegir las variables sobre las que particionar el problema priorizando las de mayor actividad.

- max_splitters: 10. Este parámetro pone un límite a la cantidad de *workers* que simultáneamente pueden estar dedicados a ejecutar un *split*. La elección de este parámetro fue totalmente arbitraria. Dado que la cantidad de *workers* totales del sistema serían alrededor de 50, tener más de un %20 de los mismos realizando *split* se consideró excesivo. Y poner una limitación más fuerte se consideró que resultaría demasiada intrusiva con la estrategia de resolución elegida.

- targetunsatfreq: 0,15. Este parámetro define la frecuencia de unsats bajo la cual se realizará un *split* automático. Nuevamente este parámetro se definió basado en los resultados obtenidos en el trabajo previo.

- split_time_limit: 60. Este parámetro define una cota de tiempo para realizar un *split*. Dado que el *split* sólo busca generar sub-problemas y descartar los triviales mediante

[5] Notar que las *cláusulas aprendidas* durante una resolución abortada (que luego devendrá en un *split*) son transmitidas al resto del sistema mediante el procedimiento propuesto en este trabajo. Por lo que este caso resulta en parte cubierto por la nueva estrategia.

microsolving, y que *nvars* se define en 70, se considera que más de 60 segundos para obtener dicha cantidad de sub-problemas resulta excesiva.[6]

- split_solve_epsilon: 0,01. Este parámetro define el tiempo dedicado al *microsolving*.
- split_backoff_time: 5. Este parámetro es una "protección" a los sub-problemas jóvenes. Define cuanto tiene que ser el tiempo mínimo de resolución de un problema para permitir que el mismo sea dividido en sub-problemas[7].
- workers: 52. La elección de usar 52 workers se debe a limitaciones de infraestructura y tiempo. Ya que como cada experimento requiere ser ejecutado múltiples veces y el Hardware es limitado, se eligió una cantidad de nodos que sea suficiente para lograr una buena paralelización y analizar el rendimiento del sistema a la vez que se cumplen con los requerimientos de Hardware y tiempo.

Esta configuración es la que se utilizará como base de referencia, para así determinar el efecto que tienen sobre la herramienta las nuevas funcionalidades presentadas en este trabajo. El resto de las configuraciones son modificaciones de esta agregando las distintas estrategias de *learning* y *restart* a analizar.

4.2. Aspectos del sistema analizados

A continuación se detallarán los distintos ángulos sobre los que se estudiaron los cambios introducidos a la herramienta.

4.2.1. Estrategias de Restart analizadas.

Para realizar los experimentos de *restart* fue necesario analizar las distintas estrategias creadas. Estos experimentos se consideran preliminares, ya que se realizaron sobre un conjunto acotado de experimentos y no se realizaron la misma cantidad de repeticiones que en el caso de los experimentos finales.

Se midió el tiempo total de ejecución utilizando *restarts* por *UNSAT*, por *ABORT* y por longitud de colas de trabajos. Se probó sólo el uso de una progresión geométrica para el caso de los *restarts* por longitud de las colas y progresiones tanto geométricas como de Luby para los *restart* por cantidad de *ABORT* y *UNSAT*. A su vez se probaron distintas configuraciones de base y exponente para lograr distintas frecuencias de *restarts*.

En todos los casos las diferencias se encontraron dentro de la varianza estadística. Por lo que no se pudo priorizar una configuración por sobre el resto con esta información. Si bien las pruebas no fueron concluyentes, se decidió por una estrategia de *restart* basada en la cantidad de *ABORT* limitada por una progresión de Luby[8]. Este tipo de estrategia de *restart* es la misma que se utiliza en el *SAT-solver MINISAT*. Es por esto que al no

[6] En experimentación preliminar se observó que el sistema entraba en un ciclo de generación de problemas de forma tal que la gran mayoría resultan triviales, por lo que se los descarta durante el *microsolving*. Lo que conlleva a demorar la generación y en ciertos casos inclusive a resolver el problema durante el *split* descartando todos los posibles sub-problemas generados. Para mitigar los efectos perjudiciales de este comportamiento se decidió poner un limite al tiempo de generación de sub-problemas.

[7] En experimentos preliminares se advirtió sobre la posibilidad de que el sistema cree sub-problemas e inmediatamente realice un *split* sobre los mismos debido a algún *worker* en estado ocioso.

[8] La base elegida es de 60 *ABORT* y el exponente 2 (60, 60, 120, 60, 60, 120, 240, 60, 60, 120, 60, 60, 120, 240, 480, 60...)

lograr encontrar evidencia empírica para realizar otro tipo de priorización, se la eligió por sobre el resto.

4.2.2. Estrategias de Learning analizadas

Si bien la estrategia de *learning* utilizando sólo las cláusulas de conflicto creemos que puede implicar una mejora a la herramienta, nos parece aún más relevante hacer un análisis exhaustivo de su efecto junto al *learning* de las *cláusulas aprendidas*. Es por esto que se decidió priorizar los experimentos utilizando *learning* de *cláusulas aprendidas*[9] y analizar el efecto del *learning* acotado a las cláusulas de conflicto para un subconjunto de problemas y configuraciones.

En cuanto a la elección de la longitud máxima de las *cláusulas aprendidas* a propagar por el sistema, se definió que sea 3. De manera que se propaguen (además de las cláusulas de conflicto) las cláusulas unitarias, binarias y ternarias. Esto es debido a que como se describe en trabajos relacionados [MFM04] las cláusulas de menor tamaño conllevan un mayor poder de propagación, a la vez que una gran cantidad de *cláusulas aprendidas* genera una mayor carga sobre el algoritmo de *BCP* [AS09]. Y dado que queda fuera del alcance de este trabajo un mecanismo de purga sobre la base de cláusulas aprendidas, el conjunto de las cláusulas a aprender debe ser limitado.

4.2.3. Problemas considerados

Para definir qué problemas se considerarían se utilizaron problemas generados por Alloy [Jac06]. Una de las ventajas de estos problemas es que son configurables en cuanto a su dificultad o *scope*. Al ser chequeos sobre modelos acotados, cambiando las cotas del modelo se logra aumentar o disminuir su complejidad. Se buscaron problemas cuya complejidad aumente de manera previsible de forma tal que se puedan generar problemas de resolución compleja, pero que a la vez el tiempo se mantenga dentro de los margenes de medición. Estableciendo un Time-Out de aproximadamente dos días, se espera que el problema termine con el planteo original de la herramienta en más de dos horas, pero antes de que se produzca el Time-Out.

Para tener una noción sobre su dificultad, se resolvió un conjunto de problemas utilizando *MINISAT* para distintas complejidades[10] (o *scope*) de cada problema. Un subconjunto de problemas se destacaron cumpliendo con los requerimientos antes descriptos. Figura 4.1

Los problemas a utilizar (y el *scope* considerado para cada uno) fueron los siguientes:

- AVL Acíclico (8): Para un modelado de un árbol, se chequea que para todo árbol AVL, su altura se encuentra acotada.

- Closure (14): Se trata de un modelo sencillo que involucra un dominio de datos A y una relación $R \subseteq A \times A$, a la que (sólo) se le exige que sea reflexiva y transitiva. La propiedad a verificar afirma que $R^* \equiv R$, esto es, que la clausura reflexo-transitiva de una relación reflexiva y transitiva es exactamente esa relación. Lo interesante de este problema es la simplicidad del modelo que genera una traducción simple con

[9] Que incluye la distribución de las cláusulas de conflicto entre los *workers*.

[10] Cuando se habla de la complejidad de los problemas generados por modelos Alloy, hablamos de las cotas para los dominios de las distintas variables del problema. Ya que el costo de probar una propiedad mantiene una relación directa con el tamaño de dichos dominios.

una cantidad de variables relativamente baja, manteniendo un costo de resolución exponencialmente creciente.

- Mark & Sweep - Completeness (10): Para un modelado de un algoritmo de Garbage Collection basado en Mark & Sweep, el problema chequea que tras realizar el algoritmo, todo nodo inalcanzable desde el *heap* efectivamente aparece en la lista de nodos libres.

- Network Routing (10): Este caso proviene de un modelo formal de ruteo en redes heterogéneas, desarrollado en AT&T, que involucra agentes móviles, identificadores, rutas y múltiples dominios de red. El modelo en cuestión incluye diversas propiedades, la que se consideró es *Structure Sufficient For Pair Returnability*. Para más detalles referimos al lector al artículo "Compositional Binding in Network Domains" [Zav06].

- StableMutexRing (14): Modelo basado en el algoritmo de K-estados de exclusión mutua para un anillo plantado por Dijkstra [Dij74]. Define un anillo de nodos con estados, donde cada nodo define su estado basado únicamente en su propio estado y el estado de uno de sus vecinos. Este problema prueba que dado un estado inicial del sistema válido, todo estado subsiguiente también lo será. Para más información referirse al articulo [TNPK01]

Al finalizar este documento, en la sección de Apéndices, se pueden encontrar las definiciones de estos problemas.

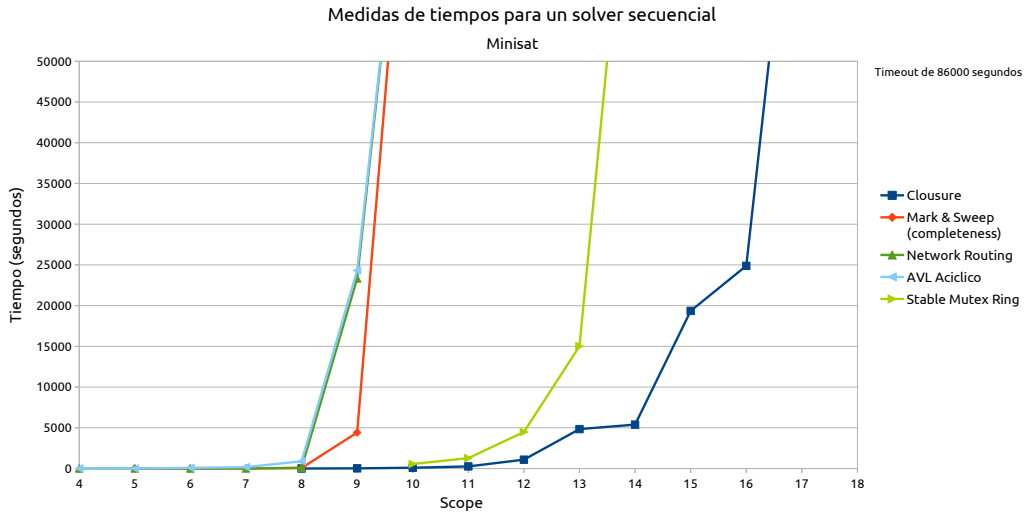

Fig. 4.1: Tiempo de ejecución en un solver secuencial (*MINISAT*) para los problemas considerados.

4.3. Experimentación

En la siguiente sección se da comienzo a la experimentación explicando su motivación y los resultados obtenidos. Los mismos serán discutidos en el capítulo 5.

4.3.1. Efecto del Learning sobre sub-problemas aislados

Dado que *Paralloy* es un sistema complejo con interacciones automáticas que dificultan la predicción y análisis, el primer experimento que se realizó fue enfocado a determinar el efecto del *learning* más allá del tiempo total de ejecución del sistema. Para lograr esto se instrumentó *Paralloy* de forma tal que para cada resolución de un sub-problema se registre no sólo el tiempo de resolución, sino también el tamaño del conjunto de *cláusulas aprendidas* con el que se contaba así como también el *trail* que lo define.

Al obtener dicha instrumentación sobre la herramienta se diseñó un experimento que consta de dos pasos:

- Resolver un problema utilizando *learning* con la herramienta instrumentada.
- Resolver cada sub-problema obtenido durante la primer ejecución utilizando *MINISAT* sin ningún tipo de modificación. Así se logra obtener para cada sub-problema de la resolución inicial, el tiempo de ejecución aislado (sin el efecto del *learning*).

Este experimento permite realizar un análisis comparativo sobre el efecto del *learning* para cada sub-problema que la herramienta genera, aumentando la granularidad del análisis. Si bien este experimento no toma en consideración los efectos secundarios que el *learning* puede ocasionar sobre la ejecución total de la herramienta (ya que puede modificar las decisiones que esta toma), permite realizar un análisis aislado, determinando entre otras cosas si el efecto es el esperado, o si el tiempo de ejecución ve incrementado al utilizar *learning*.

Resultados comparativos del efecto aislado de Learning en sub-problemas

En las figuras 4.2, 4.3 y 4.4 se muestra la *diferencia de tiempo de ejecución* de cada sub-problema generado por la herramienta, con respecto al tiempo del mismo sub-problema utilizando *MINISAT*. Se detalla a su vez por el color, dentro de qué iteración de *restart* se encontró el sub-problema al ejecutarse en la herramienta. Además el eje horizontal determina la cantidad de sub-problemas que ya habían sido resueltos por el sistema previo a la resolución del sub-problema, de esta forma se puede obtener una noción de la cantidad de la información recopilada disponible durante su resolución. Estos gráficos fueron realizados para el problema Mark & Sweep de Scope 10. En un sistema con 32 *workers*. Utilizando *restart* por *UNSAT* acotado por una sucesión geométrica de exponente dos y constante tres.

A su vez, el conjunto de los sub-problemas se dividió en tres grupos según el tiempo que tomó su resolución en el *SAT-solver* secuencial. Los sub-problemas que tardaron menos de un segundo se presentan en la figura 4.2. En la figura 4.3 se representan el conjunto de sub-problemas con tiempo mayor a uno y menos a cinco segundos. Y finalmente, el resto de los sub-problemas (con tiempo mayor a cinco segundos) se encuentran representados en la figura 4.4. Esta división es meramente para simplificar la visualización y análisis de los datos.

Efecto del Learning en la resolución aislada de sub-problemas

Previo a la realización de este experimento se hubiera esperado obtener una relación entre la ganancia en la resolución de los problemas y la cantidad de información con la que se cuenta. Es decir, esperar que la ganancia sea mayor cuando mayor sea la cantidad de

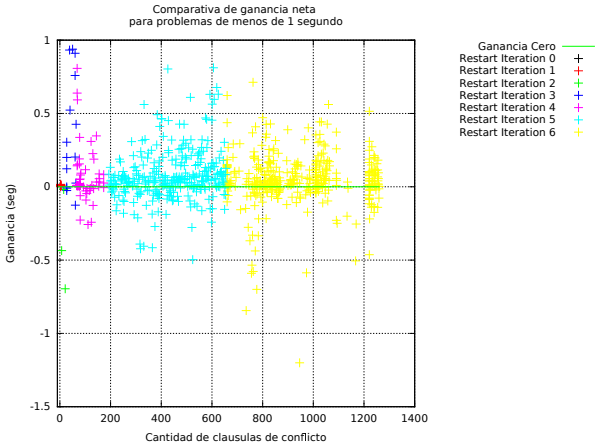

Fig. 4.2: Rango de problemas menores a 1 segundo.

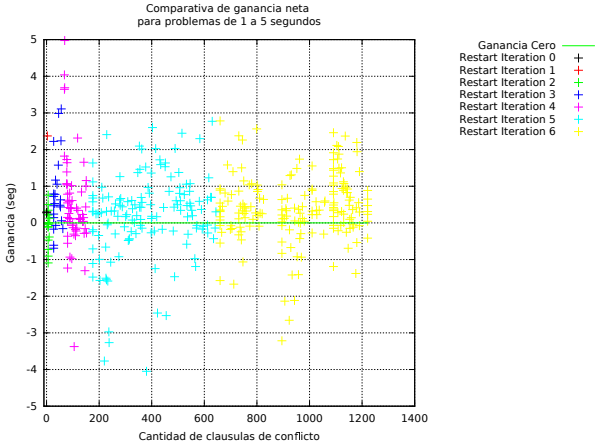

Fig. 4.3: Rango de problemas entre 1 y 5 segundos.

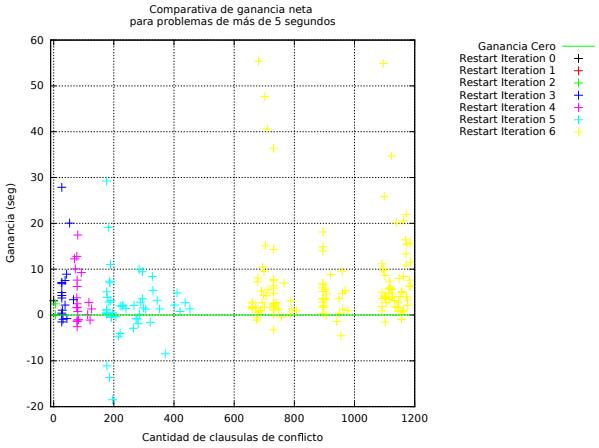

Fig. 4.4: Rango de problemas mayores 5 segundos.

Fig. 4.5: *
Comparativa *offline* con un *SAT-solver* secuencial dividido en problemas con tiempo de resolución menor a un segundo, entre uno y cinco segundos y mayor a 5 segundos.

sub-problemas ya resueltos previamente. Esta relación no fue detectada en los gráficos para los problemas de costo menor a cinco segundos. Y si bien se puede observar cierta tendencia para los problemas de tiempo de resolución mayor a cinco segundos, dicha tendencia no resulta concluyente.

Por otro lado, se observa una ganancia mayor para los problemas de costo superior a cinco segundos. Esto puede ser debido a que para estos problemas la información adicional resulta valiosa. Es decir, para los problemas de simple resolución, la información adicional que se puede agregar debería ser muy pertinente para poder reducir aún más el tiempo total. Mientras que para un problema de complejidad mayor, lo que conlleva una búsqueda más extensa, la probabilidad de que la información adicional sea pertinente y logre reducir el tiempo de ejecución es mayor.

Por otro lado se puede observar en las figuras 4.6 y 4.7 la suma de los tiempos de ejecución de los sub-problemas, tanto cuando fueron resueltos utilizando *Paralloy* como durante la resolución en el *SAT-solver* secuencial (*LearningTime* y *BaseTime* respectivamente). Se analizaron dos aspectos diferentes, por un lado se buscó detectar alguna diferencia entre la resolución de los problemas en las distintas iteraciones de *restart*, mientras que también se analizó el efecto que tiene el *learning* sobre las distintas complejidades de los sub-problemas.

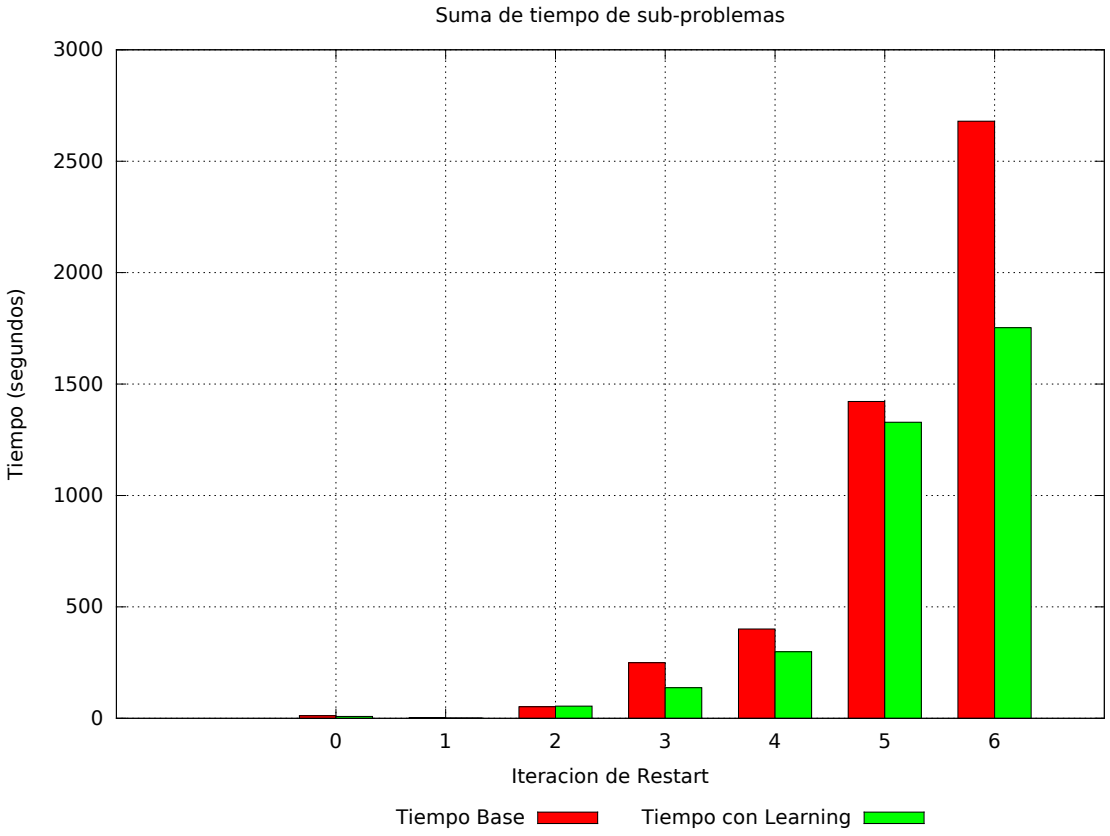

Fig. 4.6: Comparativa *offline* con un *SAT-solver* secuencial.

En la figura 4.6 se dividió el conjunto de sub-problemas por iteración de *restart*, es decir, la suma de los tiempos de los sub-problemas resueltos previos al primer *restart*, los

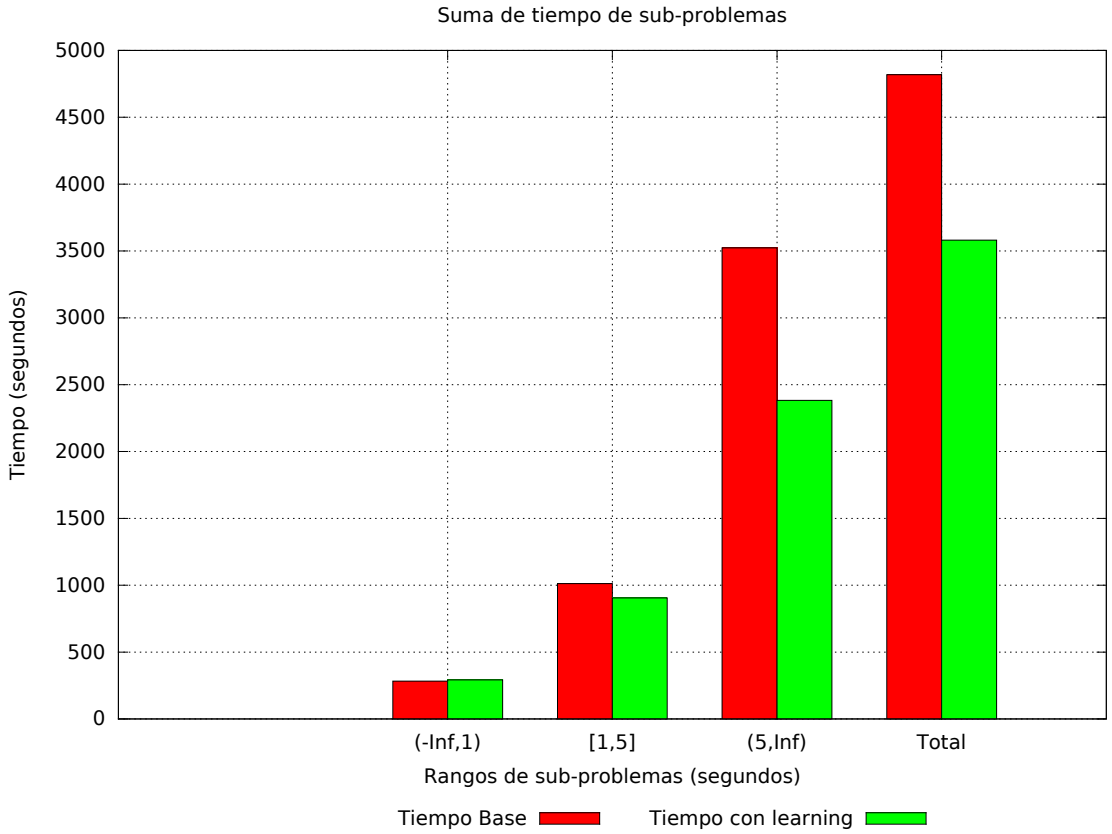

Fig. 4.7: Comparativa *offline* con un *SAT-solver* secuencial.

que fueron resueltos entre el primer y segundo *restart*, etc. En este caso las iteraciones cero y uno de *restart* resultan demasiado pequeñas en comparación con el tiempo total del sistema. No obstante en la iteración dos se puede observar que el tiempo de ejecución utilizando *learning* resultó mayor que sin utilizarlo. Esto puede ser debido al tiempo extra que se consume realizando la propagación sobre las cláusulas extra. Pero de la iteración tres en adelante se puede observar un efecto positivo en el tiempo de resolución incorporando *learning*. Si bien la ganancia neta en las iteraciones tres cuatro y cinco parece mantenerse, la ganancia relativa al tiempo de ejecución total se puede ver que decae de iteración en iteración. Para luego en la iteración seis volver a ser comparable en proporción a la iteración tres, pero mucho mayor en valor absoluto relativo a las anteriores.

En la figura 4.7 se grafica la suma de los tiempos para los mismos tres subconjuntos de sub-problemas utilizados en los gráficos comparativos (de cero a un segundo, uno a cinco y mayor a cinco segundos) y se incluye adicionalmente la suma total. En este gráfico se vuelve observar que para problemas de resolución inferior a un segundo las ganancias y perdidas de la información adicional se cancelan al tomar en cuenta la totalidad de estos problemas; mientras que para los problemas de complejidad entre uno y cinco segundos se puede observar una pequeña ganancia no concluyente, y para los problemas de costo de resolución mayor a cinco segundos se puede observar una diferencia de alrededor del 30 % del tiempo total de ejecución.

4.3.2. Análisis sobre la distribución del esfuerzo en los Workers.

El objetivo del siguiente análisis es determinar de qué manera se distribuye el esfuerzo entre cada una de las tareas que realizan los *workers*. Buscando comprender el comportamiento general de la herramienta y poder detectar, si existieran, irregularidades o alteraciones del comportamiento esperado.

Preparación del análisis funcional

Se definieron diferentes métricas sobre el tiempo que se dedica a cada una de las funcionalidades en los *workers*:

- UNSAT: El tiempo que se encuentra ejecutando un sub-problema que termina en un resultado.
- ABORT: El tiempo de ejecución un sub-problema que finalmente resulta dividido en sub-problemas.
- SPLIT: El tiempo total de generación de sub-problemas.
- LOAD: El tiempo de carga de información en un *SAT-solver*.
- LEARN: El tiempo de extracción de información luego de la ejecución de un *SAT-solver*.

Tomando en cuenta esta partición de tareas se instrumentó la herramienta para registrar estos tiempos para cada corrida. Estas mediciones se realizaron diferenciando cada uno de los *workers* del sistema, para tener una visión amplia y poder detectar problemas en cuanto a posibles diferencias de carga.

Detalle de la distribución del esfuerzo por Worker

En la figura 4.9 se puede ver el detalle del tiempo dedicado por funcionalidad para cada *worker*, notar también que el tiempo de ejecución total resulta bajo (76 segundos totales). Mientras que la figura 4.8 presenta la misma información para una resolución del mismo problema que requirió en mayor tiempo de ejecución (570 segundos totales). Ambos gráficos son de corridas de Mark & Sweep para *scope* 10, sin *restart* y con *learning* de *cláusulas aprendidas* binarias.

Análisis y comparación de la distribución del esfuerzo por Worker

Al comparar los gráficos obtenidos, se puede observar gran diferencia en cuanto a la distribución de la carga en el sistema. En la figura 4.9 se ve que la carga del sistema no resultó pareja, mostrando un desaprovechamiento de los recursos del sistema. Mientras que la figura 4.8 proveniente de una resolución más costosa en tiempo, expone un mejor aprovechamiento del sistema y una carga más pareja, siendo el tiempo dedicado a la resolución de los problemas el que gobierna la corrida.

Esta diferencia en el aprovechamiento del sistema puede ser explicada entendiendo a nivel general, el comportamiento de *Paralloy* durante la resolución de un problema. Éste puede tipificarse en las siguientes tres etapas:

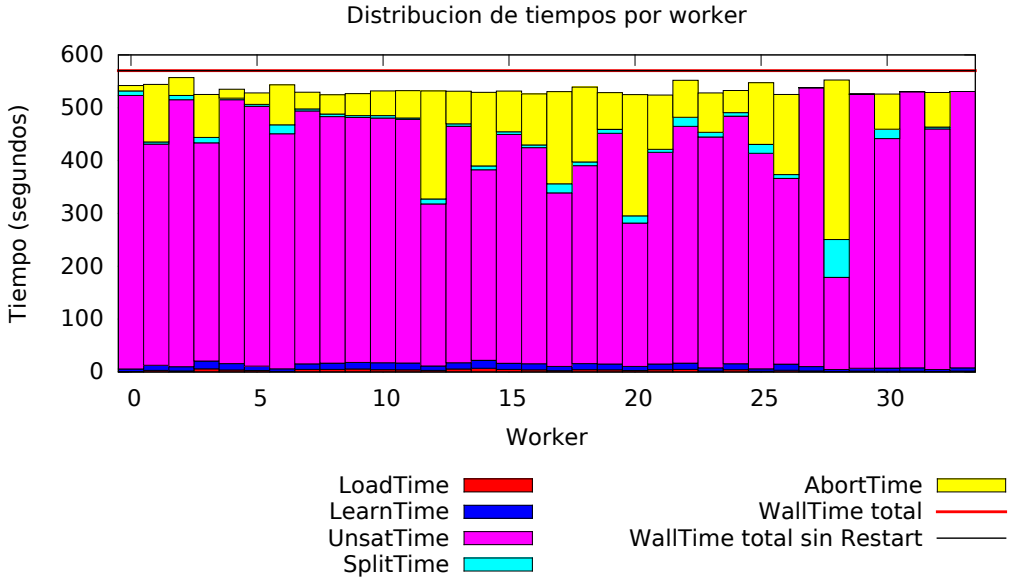

Fig. 4.8: Carga por *worker* sin *restart* con *learning* de *cláusulas aprendidas* binarias.

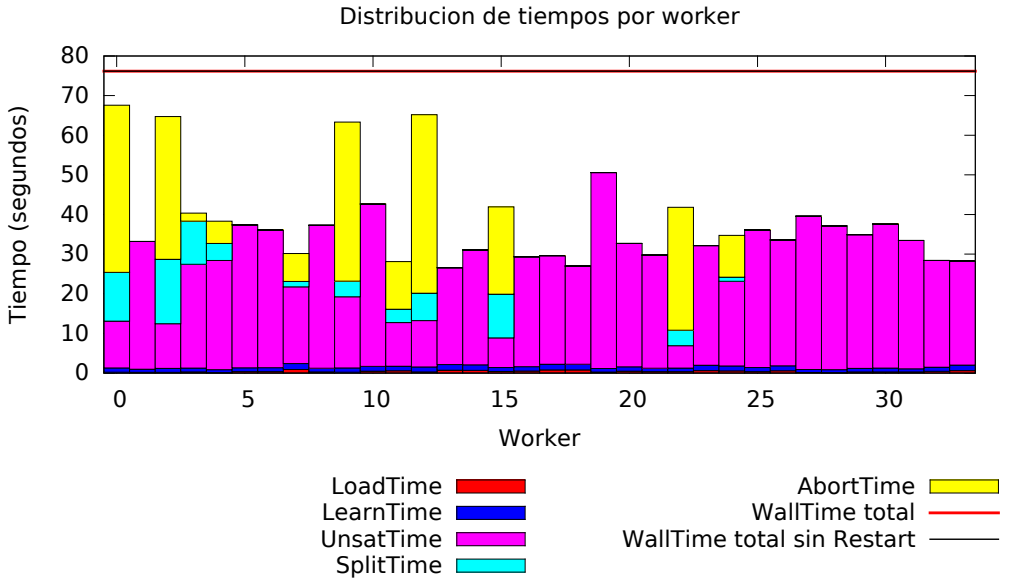

Fig. 4.9: Carga por *worker* sin *restart* con *learning* de *cláusulas aprendidas* binarias.

1. Una primer etapa, donde a partir del problema inicial se realiza una gran cantidad de *splits* hasta alimentar todas las colas de trabajos de los *workers* del sistema.

2. Una vez que se logra ese estado de carga, comienza una fase de resolución plena, donde todos los *workers* tienen trabajo para realizar y se generan *splits* más aisladamente, en general a causa de una baja frecuencia de *UNSAT*.

3. Finaliza una fase donde las colas de los *workers* se vacían y los problemas generados por *split* son los suficientemente sencillos para ser resueltos al mismo paso o aún más rápido de lo que se generan trabajos nuevos.

Tanto durante la primer como tercer fase existe una subutilización de recursos en el sistema debido a *workers* ociosos sin sub-problemas asignados. Es por esto que cuanto más predomine la fase dos, mejor será la utilización promedio del sistema.

Análisis de la carga del sistema en base a la cantidad supproblemas creados

Además del análisis funcional por *worker* presentado, se analiza el tamaño de las colas de trabajo de los *workers* a lo largo de la ejecución. Los gráficos 4.11 y 4.10 presentan esta información para las mismas corridas que generaron los gráficos 4.8 y 4.9 respectivamente. Se puede observar aquí cómo para el caso de una subutilización de recursos las colas de trabajo no logra mantenerse en un nivel suficiente para asignar trabajo a la totalidad de los *workers*[11]. Mientras que para las figuras 4.8 y 4.11, que representan un problema con eficiente utilización de los recursos, se puede observar cómo las colas de trabajo superan ampliamente la cantidad de *workers* en el sistema durante la mayoría del tiempo de ejecución. Esto explica la subutilización de los recursos del sistema para problemas que no son capaces de generar suficientes sub-problemas para mantener a todos los nodos del sistema con trabajo.

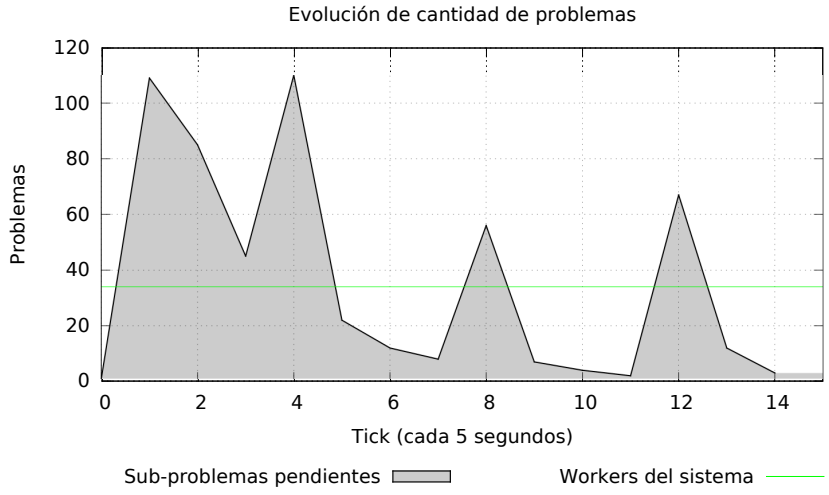

Fig. 4.10: Distribución de colas de trabajo a lo largo del tiempo.

[11] Mientras que la cantidad de trabajos se encuentre por debajo de la cantidad de *workers* activos en el sistema, existirán *workers* ociosos.

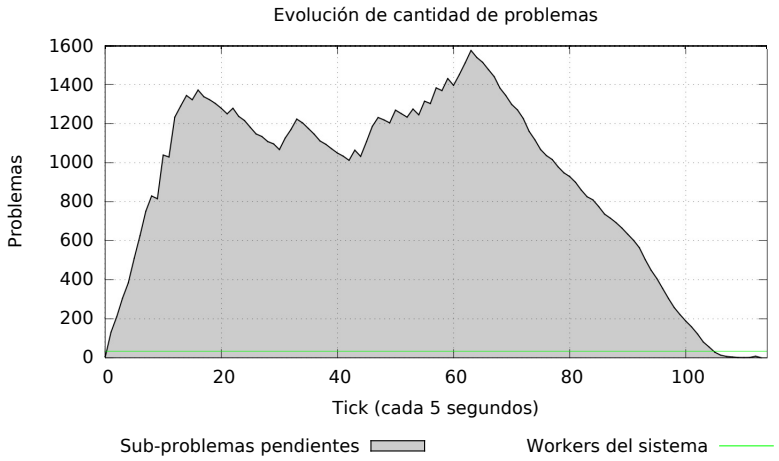

Fig. 4.11: Distribución de colas de trabajo a lo largo del tiempo.

Análisis de la carga al introducir Restart

Al incorporar *restart* a la herramienta, el proceso de detención de las resoluciones, vaciamiento de las colas de trabajos y posterior re-asignación de problemas, representa un costo adicional al tiempo requerido para la resolución. Se desea analizar el impacto de este proceso, midiendo el tiempo invertido. Con este fin se realizó una nueva ejecución para el mismo problema[12] incorporando *restart*, el resultado puede observarse en 4.12.

Se puede ver, además del tiempo total de ejecución (Total walltime), la recta *Total walltime no restart*, que representa el tiempo total descontando el invertido en el proceso de detención y re-activación del sistema para realizar los *restarts*. En esta ejecución en particular la herramienta realizó un total de 6 iteraciones de *restart*[13], sin significar el tiempo invertido en este proceso una perdida representativa.

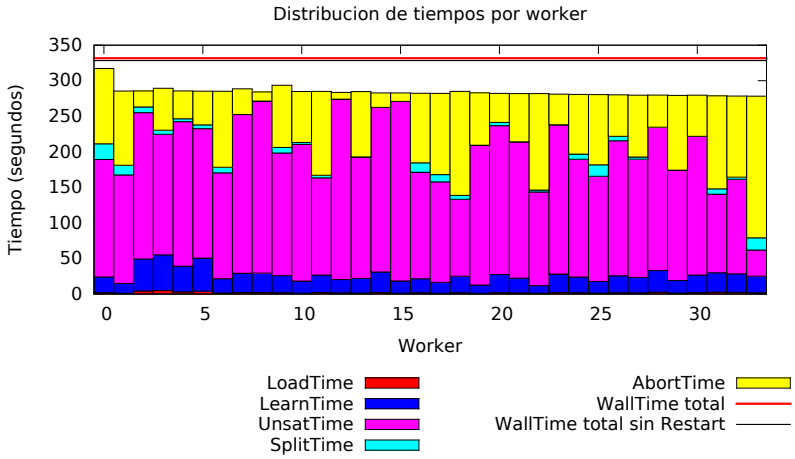

Fig. 4.12: Carga por *worker* con *restart* con *learning* de *cláusulas aprendidas* binarias.

[12] Mark & Sweep para *scope* 10, utilizado en la realización de las figuras 4.9, 4.8, 4.10 y 4.11
[13] Esta información no se observa en el gráfico.

4.3.3. Rendimiento general de la herramienta introduciendo las técnicas propuestas.

Configuración para el análisis integral al incorporar Learning y Restart.

Además de analizar las particularidades del sistema para determinar el efecto de los cambios propuestos es necesario medir el rendimiento total de la herramienta, teniendo en cuenta todas las interacciones que se dan lugar durante una resolución. Para ello se resolvieron los problemas propuestos[14] con las siguientes configuraciones:

- Configuración base (descripta es la sección 4.1.4) sin *restart* ni *learning* de ningún tipo.
- *learning*: La configuración base con *learning cláusulas aprendidas* de hasta longitud tres.[15]
- *restart*: La misma configuración que el caso de *learning*, pero utilizando también una estrategia de *restart*[16].

Además se analizó la herramienta utilizando *learning* para sólo las cláusulas de conflicto para los problemas Mark & Sweep, Network Routing y Stable Mutex Ring. Para obtener valores promedios representativos de las métricas a analizar, fue necesario repetir diez veces la resolución de cada configuración considerada.

Comparación del rendimiento integral del sistema.

Para cada resolución se registró el tiempo total de ejecución, calculando para cada configuración, el promedio y la varianza estadística de este valor (presentados en la tabla 4.1). A partir de estos valores, se realizó también el cálculo del intervalo de confianza de nivel 0,9 para el tiempo de ejecución esperado de los distintos escenarios planteados, esto se puede ver en la figura 4.13.

A partir de la experimentación se observa para todos los problemas una disminución del tiempo de ejecución esperado al incorporar *learning* con *cláusulas aprendidas* en comparación con la versión original de la herramienta. Se puede ver también que si bien el tiempo realizando también *restarts* es menor al tiempo original de la herramienta (excepto para el problema *Network Routing*), éste es mayor al obtenido utilizando sólo la propagación de las *cláusulas aprendidas*.

Por otro lado el efecto de la sóla utilización de las cláusulas de conflicto no resulta consistente, ya que para el problema *Stable Mutex Ring* el tiempo es comparable al obtenido utilizando las *cláusulas aprendidas*, mientras que para el problema *Closure* su utilización implica tiempo de ejecución incluso mayor que el original.

[14] Los cinco problemas propuestos fueron presentados en la sección 4.2.3.
[15] Ver sección 4.2.2.
[16] Ver sección 4.2.1.

Problema	Base		Learning (Sólo conflictos)		Restart				Learning (Clausulas Aprendidas)	
	Tiempo Total	Varianza	Tiempo Total	Varianza	Cantidad de Restart	Ultimo Restart	Tiempo Total	Varianza	Tiempo Total	Varianza
AVL aciclico Scope 8	7333.30	1732.96	-	-	1.00	-	357.11	92.08	430.47	83.32
Closure Scope 14	40375.64	7712.48	65965.68	33211.94	2.33	546.09	1326.93	696.74	776.42	205.82
Mark & Sweep Completeness Scope 10	98559.20	25786.30	-	-	29.80	4506.67	26517.89	25471.75	17946.45	32720.31
Network Routing Scope 10	3327.70	1262.27	-	-	12.90	2167.15	10523.10	5507.46	2847.82	759.34
Stable Mutex Ring Scope 14	8993.46	10155.94	2310.92	1402.19	5.89	1359.54	4045.21	677.49	2055.78	827.11

Tab. 4.1: Promedio de timepos de ejecuciòn para las distintas estrategias

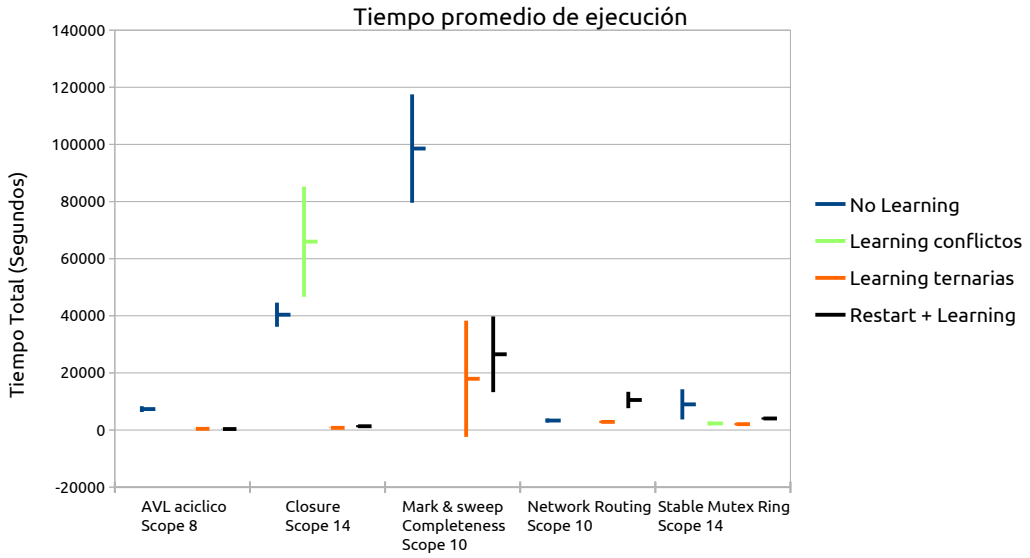

Fig. 4.13: Gráfico de tiempo total de ejecución promedio.

Efectos sobre el tiempo de ejecución promedio para las distintas estrategias

La tabla 4.2 presenta la relación entre el tiempo promedio de resolución al incorporar las estrategias propuestas con respecto al obtenido a partir de la configuración base.

Problema	Learning Conflictos	Restart + Clausulas Aprendidas	Learning Clausulas Aprendidas
AVL aciclico Scope 8	-	0.049	0.059
Closure Scope 14	1.634	0.033	0.019
Mark & Sweep Completeness Scope 10	-	0.269	0.182
Network Routing Scope 10	-	3.162	0.856
Stable Mutex Ring Scope 14	0.257	0.450	0.229

Tab. 4.2: Comparativa de promedios de tiempo de ejecución.

Cuando se analiza caso por caso, se observa que el que menos ganancia obtiene (alrededor de un 15 %) es *Network Routing*. Este problema ya había demostrado una mejora importante en el trabajo [Vis13] al ser paralelizado utilizando *Paralloy*, lo que puede justificar la pequeña diferencia al introducir *learning*. A su vez explica porqué el *restart* tuvo efectos negativos, ya que detener la resolución para volver a comenzarla cuando un problema se está resolviendo de manera eficiente resulta contraproducente. No se debe olvidar que el objetivo detrás del *restart* es evitar resoluciones deficientes que lleven a tiempos de ejecución altos para lograr resoluciones más eficientes.

Análisis de la carga durante la resolución

Se verificó también para estos problemas que la carga durante la ejecución sea homogénea y no haya desaprovechamiento de recursos, verificando también que el tiempo invertido en el proceso de *restart* sea despreciable. Se puede ver un ejemplo del comportamiento en los gráficos 4.15 y 4.14, los mismos se generaron a partir de una ejecución del problema Mark & Sweep con *restart* y *learning*[17].

En la figura 4.14 se observan los trabajos pendientes en el sistema, superando ampliamente la cantidad de *workers* en el sistema en todo momento excepto por los instantes en que se realiza un *restart*, se comienza y se finaliza la resolución. Otro punto a notar es que, si bien para este problema se realizaron veintiún iteraciones de *restart*, esto sólo resultó en un tiempo total de ejecución 1,5 veces mayor utilizando sólo *learning*.

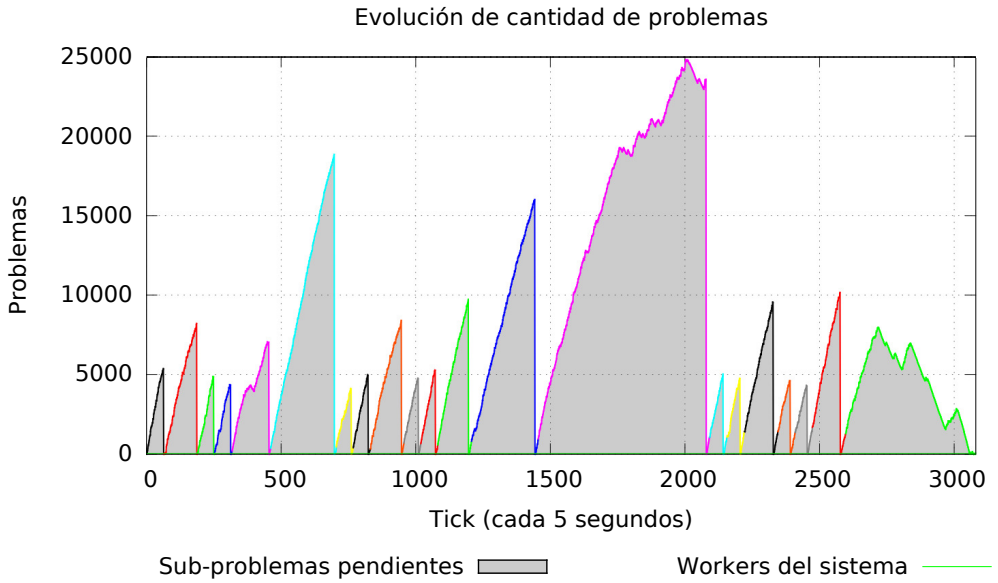

Fig. 4.14: sub-problemas creados durante la resolución de Mark & Sweep con *restart* y con *learning* de *cláusulas aprendidas* binarias.

Por otro lado la figura 4.15 muestra la carga para la misma ejecución, se puede ver en este caso una alta utilización de los recursos disponibles. También se observa que el tiempo de resolución de problemas que concluyeron es similar al dedicado a problemas que fueron finalmente abortados. A su vez, el tiempo dedicado a la extracción y procesamiento de la información representa cerca del %17 del tiempo total. Y por último se observa que, pese a la gran cantidad de *restart* que realizó la herramienta, el tiempo invertido en este proceso resulta mínimo, no así el tiempo invertido en problemas que fueron finalmente detenidos (o abortados).

[17] El resto de los gráficos no se presenta por no presentar ningún otro aspecto relevante

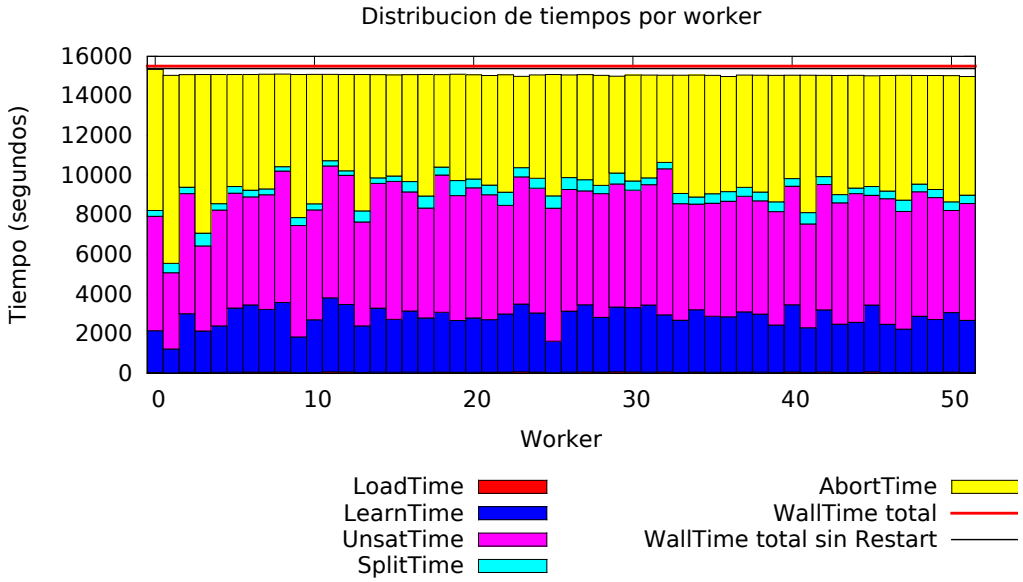

Fig. 4.15: Carga por *worker* con *restart* y con *learning* de *cláusulas aprendidas* binarias.

Análisis comparativo del tiempo por iteración de Restart.

Se ralizó también un análisis más detallado sobre el tiempo de ejecución promedio de la última iteración de *restart*. En la figura 4.16 se presenta el promedio de lo siguiente[18]:

1. El tiempo de ejecución que le toma a la herramienta comenzar de cero la resolución de un problema, pero contando con toda la información recopilada en iteraciones anteriores. Es decir, el tiempo de la última iteración de *restart*.

2. El tiempo total de resolución del problema utilizando *restart* (la suma del tiempo de todas las iteraciones de *restart*).

3. El tiempo promedio de ejecución sin utilizar *restart*.

Se puede observar claramente cómo el costo de resolución del problema, contando con la información recopilada en resoluciones anteriores es mucho menor al costo que lleva la totalidad de la resolución sin esta información. Siendo, en promedio, menor que el tiempo requerido para la resolución utilizando sólo *learning*, aunque esta diferencia para la mayoría de los casos no es suficientemente pronunciada para descartar que se trate de variaciones estadísticas.

[18] En todos los casos se utiliza *learning* de *cláusulas aprendidas*.

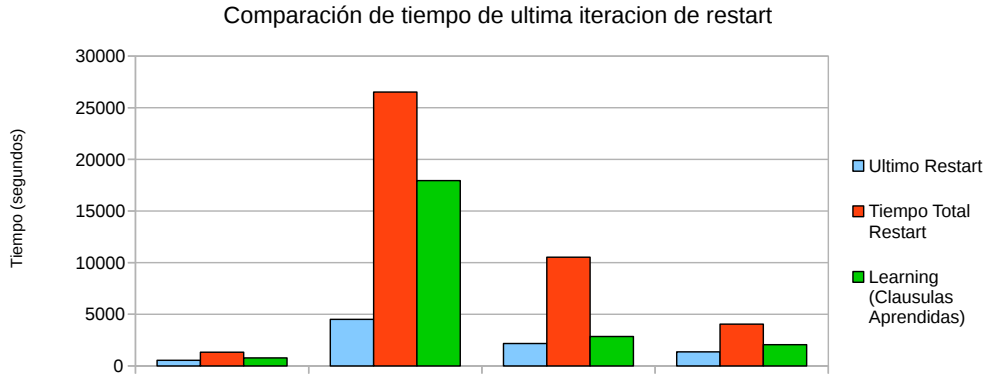

Fig. 4.16: Gráfico comparativo de tiempo de ejecución de la última iteración de *restart* contra el tiempo total de resolución y el tiempo total de resolución sin *restart*.

5. CONCLUSIONES, LIMITACIONES Y TRABAJO FUTURO

En el presente capítulo se analizan los resultados expuestos en la evaluación experimental (capítulo 4) a fin de aportar claridad al análisis se dividió en tres secciones.

- Sección 5.1: se analizan, las modificaciones realizadas a la herramienta *Paralloy* y se responden las preguntas de investigación planteadas.
- Sección 5.2: se plantean las limitaciones del trabajo y se abren nuevas preguntas que pueden abordase en futuras investigaciones.
- Sección 5.3: cierra el trabajo realizando un resumen, análisis y conclusión final.

5.1. Análisis de los resultados

A lo largo del trabajo se analiza la incorporación de *learning* y *restart* a *Paralloy*, un *SAT-solver* distribuido. Se analiza el efecto del *learning* para distintos niveles de información a distribuir por el sistema. Incorporando también la posibilidad de realizar *restarts* globales que detienen el trabajo en todos los nodos para recomenzar sin perder registro de la información antes generada.

Entre las estrategias y configuraciones consideradas se logró obtener una reducción del tiempo de resolución total de la herramienta al utilizar *learning* de *cláusulas aprendidas* de hasta longitud tres. No obstante, no se lograron obtener mejores resultados sumando iteraciones de *restart*. Si bien las pruebas muestran un aumento en el tiempo de resolución al utilizar este proceso, el tiempo de ejecución de las distintas iteraciones del *restart* arrojó resultados positivos.

Analizaremos más en detalle ambos escenarios en las siguientes secciones.

5.1.1. Impacto de la técnica de Learning aplicada

Como ya se dijo, al propagar las cláusulas de conflicto y las cláusulas aprendidas de hasta longitud tres a todos los nodos del sistema, se observa un marcado descenso en el tiempo de ejecución promedio. Esto se condice con la hipótesis inicial, ya que al distribuir el trabajo entre los diferentes *workers*, como originalmente lo realiza *Paralloy*, la perdida de información obtenida por cada unidad de cómputo resulta en detrimento del rendimiento total del sistema. Lo que comprueba que dicha información es vital para realizar una resolución eficiente.

Se pudo observar además, que la sola incorporación de las cláusulas de conflicto no resulta consistentemente favorable como se hubiera esperado. Si bien el análisis de esta estrategia no resultó tan exhaustivo como el caso de las *cláusulas aprendidas*, los resultados dieron muestras de que esta información no es suficiente para lograr una mejora general a la herramienta. Se obtuvo para el problema *Closure* un tiempo de ejecución total mayor al incorporar estas cláusulas que el obtenido sin ninguna modificación. Mientras que en el caso de *Stable Mutex Ring* el tiempo se asemeja al obtenido propagando las cláusulas aprendidas, siendo este mucho menor que el rendimiento inicial. Esto indica una dependencia del problema sobre el efecto positivo o negativo que las cláusulas de conflicto puedan tener.

Esta variación puede estar dada debido a la longitud de las cláusulas propagadas, ya que al contrario que para las cláusulas aprendidas (donde la longitud de las cláusulas a propagar es controlada y está definida), la longitud de las cláusulas de conflicto es variable y depende exclusivamente del problema y de las particiones realizadas. De esta forma, la generación de cláusulas de conflicto de gran longitud es posible, y esto se vuelve más común a medida que el camino de resolución y de particiones se extiende (cabe recordar que las cláusulas de conflicto son un subconjunto del *trail* del sub-problema). Es por esto que a medida que el *trail* crece en profundidad, la longitud de las cláusulas de conflicto también será mayor como regla general, generando cláusulas que hacen más lento el algoritmo de *BCP*.

Si bien al ser un subconjunto del *trail*, al propagarlas a un problema "hermano" un subconjunto de sus variables será automáticamente definida al fijar el *trail* del sub-problema haciendo a la cláusula superflua o reduciéndola, esto no resulta así para sub-problemas lejanos en el árbol de derivación. Ya que el conjunto de variables en dos ramas de un árbol de derivación sólo será necesariamente igual en el tramo que las ramas compartan. Siendo totalmente independientes en el resto de su extensión.

Dado que el *trail* de los sub-problemas generados se alarga a lo largo de la resolución, este fenómeno puede explicar la diferencia en cuanto al efecto sobre el tiempo de ejecución que tiene la incorporación de las cláusulas de conflicto. Siendo positivo en el caso de un problema con tiempo de resolución del orden de las decenas de minutos, mientras que para un problema cuya resolución lleva cerca de mil minutos conlleva un efecto es nocivo. Lo que deja entrever la importancia de la longitud de las cláusulas en cuanto a su pertinencia e información que aportan.

Al observar el efecto del *learning* de *cláusulas aprendidas* en la tabla 4.2, se observa que el tiempo de ejecución representa el 2 % o 6 %[1] para problemas como AVl y Clousure; mientras que para Mark & Sweep y Stable Mutex Ring es de al rededor de un 20 % del tiempo original; pero al ver observar Network Routing, se observa una mejora del sólo un 15 % (es decir, un 85 % del tiempo original). Este problema en particular, obtiene un gran beneficio de la partición y distribución del trabajo [Vis13]. Este puede ser el motivo de que el beneficio del *learning* no resulte tan importante, no obstante, aún en este caso, donde se obtiene una gran ganancia de la paralelización, se introdujo una mejora de un 20 % en el tiempo promedio de resolución.

Los resultados obtenidos al introducir la propagación de cláusulas aprendidas para los problemas elegidos muestran mejoras consistentes en cuanto al tiempo promedio de ejecución[2]. Se logró de esta forma mejorar el rendimiento del sistema para problemas cuyo tiempo de ejecución resulta muy elevado con los métodos de solving secuencial aumentando el poder de la herramienta *Paralloy*.

En cuanto a la **pregunta de investigación 1** podemos decir que fué posible definir un sistema de distribución de la información para el Sat-Solver Paralloy que mejore la eficiencia del sistema. El procedimiento planteado de distribución de la información genera para *Paralloy* una mejora en la eficiencia de resolución de problemas con requerimientos de computo mediano/grande. Pudiendo sacar provecho de la información obtenida en las distintas unidades de computo a lo largo de todo el proceso de resolución del problema.

[1] Con respecto a la versión original.

[2] Vale aclarar que estas mejoras se ven en problemas de dificultad mediana a grande, cuando se analizó el sistema para problemas de menor dificultad los resultados no fueron contundentes debido a la varianza en el tiempo final.

En cuanto a la **subpregunta 1*a***, se puede observar que la información generada en una unidad de cómputo resulta relevante para otras unidades del sistema. Si bien no se realizó un estudio minucioso sobre la relevancia de cada una de las cláusulas para los distintos sub-problemas, se observó que su tiempo de resolución cuando se introduce la información generada en resoluciones previas se ve afectado considerablemente. En el capítulo 4.3.1 se presentan los resultados encontrados sobre este fenómeno, donde se pudo observar tanto una disminución como un aumento del tiempo, habiendo una relación directa entre el tiempo requerido para un sub-problema dado y la mejora que obtenida al incorporar la información recabada, siendo superior el tiempo ganado que el perdido en conjunto.

Finalmente, con respecto a la **subpregunta 1*b***, priorizar la información a distribuir tomando en cuenta la longitud de las cláusulas resultó en un buen rendimiento del sistema. Dado que al distribuir las clausulas de hasta longitud tres se logró reducir drásticamente el tiempo promedio para la mayoría de los problemas considerados.

5.1.2. Impacto del Restart sobre la herramienta

La estrategia de *restart* que utilizan los *SAT-solver* secuenciales ha demostrado obtener menores tiempos de resolución. En el caso de la estrategia implementada para la herramienta distribuida, no se pudo lograr el mismo efecto. Si bien el tiempo de resolución de las distintas iteraciones parece disminuir, esta disminución no es suficiente para compensar el costo del *restart* mismo.

Al realizar un análisis más detallado de la carga del sistema durante una resolución con *restart* se pudo observar que el tiempo que se invierte en detener y recomenzar la resolución en cada iteración es despreciable en cuanto al tiempo total. Al descartar esto como causa del problema se analizaron los tiempos de las distintas iteraciones de *restart*; comparándose el tiempo promedio de la última iteración de *restart* con el tiempo de resolución sin *restart* (en ambos casos empleando *learning* de *cláusulas aprendidas*).

De esta comparación se observa que el tiempo invertido en la última iteración de *restart* es aproximadamente un 30 % del tiempo total sin *restart* para el caso Mark & Sweep. Este caso, al ser el que más iteraciones de *restart* promedio realizó (con casi 30 iteraciones), resultó en una mayor diferencia. Mientras que siendo Network Routing, el siguiente en cantidad de iteraciones realizadas (con un promedio de casi 13 iteraciones de *restart*), no se obtiene una variación representativa en el tiempo. Esto puede deberse a la peculiaridad de este caso de estudio que ha demostrado no ser afectado tan profundamente por el *learning*.

Al analizar la carga del sistema también se observa que el tiempo invertido en sub-problemas que terminan siendo abortados es comparable al tiempo que se dedica en resolver sub-problemas hasta obtener un resultado. Esto puede deberse a las iteraciones de *restart*, ya que en cada iteración se detiene la totalidad de los trabajos en ejecución, y si bien las cláusulas aprendidas durante la ejecución son distribuidas sin perder la totalidad del trabajo realizado, éste es un punto a tener en cuenta en trabajos futuros.

Para contestar la **pregunta de investigación 2**, sobre si es posible lograr mediante el procedimiento de *restart* una mejora en el tiempo promedio de ejecución, no se logró llegar a una conclusión cerrada. Luego de implementar dicho procedimiento, no se logró reducir el tiempo promedio. Como ya se discutió anteriormente los experimentos realizados muestran que el tiempo en cada iteración de *restart* decrece, por lo que potencialmente existe la posibilidad de explotar este fenómeno para lograr reducir el tiempo total de ejecución, pero en los experimentos realizados resultó mayor el costo que la ganancia

obtenida. Por lo que la respuesta a la pregunta $2a$ (¿Es posible lograr una mejora en el tiempo promedio para el sistema *Paralloy* aplicando esta técnica?) es que este trabajo no logró llegar a un esquema de *restart* que logre reducir el tiempo promedio, pero encontró efectos positivos sobre los tiempos parciales de ejecución, por lo que no es posible descartar la posibilidad de lograr dicha reducción.

Luego de realizar este análisis sobre el *restart* creemos que si bien no se lograron resultados consistentemente favorables con la estrategia implementada, el procedimiento ha demostrado tener potencial. Pero no por esto debe descartarse, demostrando que una mayor cantidad de iteraciones de *restart* es necesaria para lograr una diferencia representativa en el tiempo de ejecución de las iteraciones. No se debe olvidar que este proceso está altamente ligado a la estrategia de *learning* que se implemente, por lo que quizás con una estrategia de *learning* más agresiva se logren mejores resultados. Se discutirán otras opciones y propuestas para continuar con el análisis en la siguiente sección de trabajo futuro.

5.2. Limitaciones y Trabajo futuro

Durante la experimentación y análisis se detectó una fuerte correspondencia entre el uso de recursos del sistema y la "dificultad" del problema a resolver. Siendo imperioso que el problema a resolver sea de mediana a alta complejidad para una buena utilización de recursos. Si se tiene en cuenta que la herramienta ha sido desarrollada para este tipo de problemas esto es aceptable, ya que para problemas de resolución más sencilla se puede utilizar un *SAT-solver* secuencial tradicional. No obstante una asignación dinámica de recursos puede resultar beneficioso para el rendimiento general del sistema.

Esta limitación a su vez se encuentra relacionada con la estrategia de automatización con que cuenta la herramienta, ya que existen situaciones en las que a causa de recursos ociosos se generan divisiones de sub-problemas que resultan en detrimento del rendimiento general. Si bien durante la realización de este trabajo se agregaron métodos de control para mejorar esta situación, son sólo medidas de contención. Una análisis más profundo es necesario para resolver este problema, implementando una estrategia que tenga en cuenta una mayor cantidad de factores y teniendo en cuenta si el sistema se encuentra al inicio de la resolución, al final, o durante la etapa de resolución plena.

Sobre las estrategias implementadas, una estrategia de purga para las *cláusulas aprendidas* durante la ejecución quedó fuera del alcance de este trabajo. La implementación de dicho mecanismo y de un consecuente proceso de priorización de las *cláusulas aprendidas* es fundamental para aumentar el tamaño de la información que es posible distribuir en el sistema sin perjudicar el rendimiento. Esto habilitaría potencialmente una mejora para el procedimiento de *restart*, ya que cada iteración contaría con menor perdida de trabajo al mantener un volumen mayor de información. A su vez el mecanismo de distribución de la información que se planteó fue elegido entre otras cosas por el control que brinda en todo momento y para facilitar el análisis. Ahora que ya se ha demostrado que se pueden lograr efectos altamente positivos utilizando *learning* distribuido, se puede reemplazar por un esquema de distribución de información distribuido o una base de datos de acceso global asilada del proceso *client*.

Otro de las posibles mejoras a introducir a la herramienta es un sistema de *split* que logre mejor aprovechamiento de la ejecución abortada. Cuando un sub-problema es abortado y subdividido sub-problemas se cuenta con una ejecución parcial del sub-problema

original, actualmente sólo se utiliza para obtener el valor de actividad de las variables del problema y priorizarlas para la división del problema. Pero durante esta ejecución parcial, el *SAT-solver* generó un conjunto de sub-problemas que ya han sido descartados y problemas potenciales para resolver que no ha llegado a atacar (el árbol de resolución del sub-problema). Aprovechando este árbol para la generación de sub-problemas, se aprovecharía el cómputo ya realizado para reducir el costo del *split*, restando sólo el *microsolving* sobre los problemas potenciales y la generación de los archivos de sub-problemas.

5.3. Conclusiones finales

Luego de la incorporación de las técnicas de *learning* y *restart* globales al *SAT-solver* paralelo y distribuido *Paralloy* se lograron mejoras en el rendimiento de entre un 20 % y un 85 % utilizando *learning* distribuido de cláusulas de conflicto y *cláusulas aprendidas* de hasta longitud 3.

A la vez se cree que es posible mejorar este rendimiento incorporando *cláusulas aprendidas* de mayor longitud, pero será necesario un mecanismo de purga a la vez que una distribución de información descentralizada. De lograrse esto, al contar con mayor información, se cree que es potencialmente posible obtener mejoras en el tiempo de resolución utilizando *restart*.

A partir de un análisis detallado, se detectaron puntos débiles en la herramienta y se propusieron mejoras para la misma para obtener también un mayor aprovechamiento de los recursos que dispone. Para así evitar comportamientos que resulten en detrimento del rendimiento general (como ser entrar en un ciclo de *split* demasiado agresivo, sobre todo en la etapa final de resolución del problema).

Apéndice A

CONFIGURACIÓN Y USO DE LA HERRAMIENTA.

El código de la herramienta se puede encontrar en:

- https://bitbucket.org/ivissani/paralloy/branch/factLearn

Para la instalación se debe contar con

- mpich2 version 1.4 compilado con los parámetros:

  `--enable-shared --with-device=ch3:sock --disable-fc --disable-f77`

- Python 2.7.5 compilado con la opción `--enable-shared`

- Incluir la librería de python Mpi4py

Luego de instaladas las dependencias y obtenido el código del repositorio es necesario compilar las extensiones nativas con `python setup.py build_ext -i`

Para ejecutar la herramienta, primero se deben lanzar al *master* y los *workers* en un *cluster*, para esto usar el comando:

- `mpiexec -f <MACHINEFILE> python <IMPL_DIR>/backend.py`

  `MACHINEFILE`: archivo descriptor de los nodos donde se correrán los procesos, en en primer nodo se ejecutará el proceso *master* y en el resto los *workers*.

  `IMPL_DIR`: directorio donde se haya instalado la herramienta.

Una vez que se cuenta con los procesos ejecutando en el *cluster*, se debe ejecutar el proceso *client*:

- `python <IMPL_DIR>/cli.py cnf=<CNF> <PARAMS>`

  `IMPL_DIR`: directorio donde se haya instalado la herramienta.

  `PARAMS`: parámetros de configuración con los que se deseen ejecutar la herramienta.

Apéndice B

AVL HBND

```
module AvlAlturaAcotada

open util/integer

one sig null {}

one sig QF {
   thiz    : AVLTree,
   root    : AVLTree -> one ( AVLNode + null ),
   h       : AVLNode -> one Int,
   key     : AVLNode -> one ( Int ),
   left    : AVLNode -> one ( AVLNode + null ),
   right   : AVLNode -> one ( AVLNode + null ),
   N0,N1   : AVLNode,
   k       : Int
}

abstract sig Object {}
one sig AVLTree extends Object {}
sig AVLNode extends Object {}

pred Inv_AVLTree[
   thiz    : AVLTree,
   root    : AVLTree -> one (AVLNode + null),
   h       : AVLNode -> one Int,
   key     : AVLNode -> one Int,
   left    : AVLNode -> one (AVLNode + null),
   right   : AVLNode -> one (AVLNode + null)
]{
  all x: thiz.root.*(left+right) - null |
   (x !in x.^(left + right)) &&
   (all y: AVLNode | y in x.left.*(left+right) implies lt[y.key ,x.key]) &&
   (all y: AVLNode | y in x.right.*(left+right) implies lt[x.key, y.key]) &&
   (x.left=null && x.right=null => x.h=0) &&
   (x.left=null && x.right!=null => (x.h=1 && x.right.h=0)) &&
   (x.left!=null && x.right=null => (x.h=1 && x.left.h=0)) &&
   (x.left!=null && x.right!=null => (x.h=add[larger[x.left.h,x.right.h],1] &&
       lte[mul[signum[sub[x.left.h,x.right.h]],sub[x.left.h,x.right.h]],1]))
}

fact IsAnAVL { Inv_AVLTree[QF.thiz, QF.root, QF.h, QF.key, QF.left, QF.right] }

check { (QF.thiz).(QF.root).(QF.h) < 5 } for 0 but exactly __SCOPE__ AVLNode,
    exactly __SCOPE+1__ Object, 5 int
```

Apéndice C

CLOSURE

```
module closure

sig A {}

one sig Rel {
       r : A -> A
}

fact {(A <: iden) + Rel.r = Rel.r}
fact {((Rel.r).(Rel.r)) + Rel.r = Rel.r}

check {Rel.r = A<:*(Rel.r)} for exactly __SCOPE__ A
```

Apéndice D

MARK & SWEEP

```
module examples/systems/marksweepgc

/*
 * Model of mark and sweep garbage collection.
 */

// a node in the heap
sig Node {}

sig HeapState {
  left, right : Node -> lone Node,
  marked : set Node,
  freeList : lone Node
}

pred clearMarks[hs, hs' : HeapState] {
  // clear marked set
  no hs'.marked
  // left and right fields are unchanged
  hs'.left = hs.left
  hs'.right = hs.right
}

// simulate the recursion of the mark() function using transitive closure
fun reachable[hs: HeapState, n: Node] : set Node {
  n + n.^(hs.left + hs.right)
}

pred mark[hs: HeapState, from : Node, hs': HeapState] {
  hs'.marked = hs.reachable[from]
  hs'.left = hs.left
  hs'.right = hs.right
}

// complete hack to simulate behavior of code to set freeList
pred setFreeList[hs, hs': HeapState] {
  // especially hackish
  hs'.freeList.*(hs'.left) in (Node - hs.marked)
  all n: Node |
    (n !in hs.marked) => {
      no hs'.right[n]
      hs'.left[n] in (hs'.freeList.*(hs'.left))
      n in hs'.freeList.*(hs'.left)
    } else {
      hs'.left[n] = hs.left[n]
      hs'.right[n] = hs.right[n]
```

```
    }
  hs'.marked = hs.marked
}

pred GC[hs: HeapState, root : Node, hs': HeapState] {
  some hs1, hs2: HeapState |
    hs.clearMarks[hs1] && hs1.mark[root, hs2] && hs2.setFreeList[hs']
}

assert Completeness {
  all h, h' : HeapState, root : Node |
    h.GC[root, h'] =>
      (Node - h'.reachable[root]) in h'.reachable[h'.freeList]
}

check Completeness for __SCOPE__ expect 0
```

Apéndice E

NETWORK ROUTING

```
/* ============================================================================
This is the model on which the paper "Compositional Binding in Network
Domains" is based.
============================================================================ */

module alloy/fm06

/* ============================================================================
DOMAINS
============================================================================ */

sig Agent { }

sig Identifier { }

sig Name, Address extends Identifier { }
sig AddressPair extends Identifier { addr: Address, name: Name }



-- The meaning of an address pair lies in its content, so there are no
-- distinct address pairs with the same fields.
fact { all disj p1, p2: AddressPair |
   p1.addr != p2.addr || p1.name != p2.name }

sig Domain {
   endpoints: set Agent,
   space: set Address,
   routing: space -> endpoints
}

sig Path {
   source: Address,
   dest: Address,
   generator: Agent,
   absorber: Agent
}

pred DomainSupportsPath [d: Domain, p: Path] { {
-- The source address routes to the generator.
   p.source in (d.routing).(p.generator)

-- The destination address routes to the absorber.
   p.absorber in (p.dest).(d.routing)
} }
```

```
/* ==========================================================================
BINDINGS
========================================================================== */

sig Domain2 extends Domain {
   dstBinding: Identifier -> Identifier } {
   all i: Identifier | i in dstBinding.Identifier =>
   (  (i in Address => i in space) &&
      (i in AddressPair => i.addr in space)
   )
}

sig Path2 extends Path {
   origDst: Identifier }

pred Domain2SupportsPath [d: Domain2, p: Path2] {
   DomainSupportsPath[d,p] &&
   p.dest in (p.origDst).(*(d.dstBinding)) &&
   p.dest !in (d.dstBinding).Identifier
}

pred ReachableInDomain [d: Domain2, i: Identifier, g: Agent] {
   some a: Address |
      a in i.(*(d.dstBinding)) &&
      a !in (d.dstBinding).Identifier &&
      g in a.(d.routing)
}

pred DeterministicDomain [d: Domain2] {
   all i: Identifier | lone g: Agent | ReachableInDomain[d,i,g] }

pred NonloopingDomain [d: Domain2] { no ( ^(d.dstBinding) & iden ) }

pred AddBinding [d, d': Domain2, newBinding: Identifier -> Identifier] { {

-- Precondition: the new bindings can be applied in the domain.
   all i: Identifier | i in newBinding.Identifier =>
   (  (i in Address => i in d.space) &&
      (i in AddressPair => i.addr in d.space)
   )

-- Postconditions:
   d'.endpoints = d.endpoints
   d'.space = d.space
   d'.routing = d.routing
   d'.dstBinding = d.dstBinding + newBinding
} }

pred IdentifiersUnused [d: Domain2, new: Identifier ] { {
   no ((d.routing).Agent & new)
   no ((d.dstBinding).Identifier & new)
   no (Identifier.(d.dstBinding) & new)
} }
```

```
/* ========================================================================
DISTINCTIVE BINDINGS
======================================================================== */

sig Domain3 extends Domain2 {
   srcBinding: Identifier -> Identifier,
   AdstBinding: Identifier -> Identifier,
   BdstBinding: Identifier -> Identifier
} {
-- There are two kinds of destination binding.
   dstBinding = AdstBinding + BdstBinding
}

sig Path3 extends Path2 {
   finSrc: Identifier }

pred Domain3SupportsPath [d: Domain3, p: Path3] {
   Domain2SupportsPath[d,p] &&
   p.finSrc in (p.source).(*(d.srcBinding)) &&
   p.finSrc !in (d.srcBinding).Identifier
}

pred ReturnPath [p1, p2: Path3] {
   p1.absorber = p2.generator &&
   p2.source = p1.dest &&
   p2.origDst = p1.finSrc
}

pred AddABinding [d1, d2: Domain3, newBinding: Identifier -> Identifier]{ {

-- Preconditions:
   IdentifiersUnused[d1,newBinding.Identifier]
   no ( Identifier.(d1.BdstBinding) & newBinding.Identifier )
   no (Identifier.newBinding & newBinding.Identifier)

-- Postconditions:
   AddBinding[d1,d2,newBinding]
   d2.AdstBinding = d1.AdstBinding + newBinding
   d2.BdstBinding = d1.BdstBinding
   d2.srcBinding = d1.srcBinding
} }

pred AddBBinding [d1, d2: Domain3, newBinding: Identifier -> Identifier]{ {

-- Preconditions:
   IdentifiersUnused[d1,newBinding.Identifier]
   (all i: Identifier | lone i.newBinding )
   (all i: Identifier | lone (newBinding + d1.BdstBinding).i )
   no ( Identifier.newBinding & (d1.AdstBinding).Identifier )
```

```
   no (Identifier.newBinding & newBinding.Identifier)

-- Postconditions:
   AddBinding[d1,d2,newBinding]
   d2.AdstBinding = d1.AdstBinding
   d2.BdstBinding = d1.BdstBinding + newBinding
   d2.srcBinding = d1.srcBinding + ~newBinding
} }

/* ============================================================================
STRUCTURE FOR RETURNABILITY AND STRUCTURAL VERIFICATION
============================================================================ */

pred StructuredDomain [d: Domain3] {
   let ADom = (d.AdstBinding).Identifier,
       BDom = (d.BdstBinding).Identifier,
       RDom = (d.routing).Agent,
       BRan = Identifier.(d.BdstBinding) | {

   NonloopingDomain[d]

-- The two bindings and routing operate on different identifiers.
   no (ADom & BDom)
   no (ADom & RDom)
   no (BDom & RDom)

-- Except for AdstBinding, delivering a message is deterministic.
   (all i: Identifier | lone i.(d.BdstBinding) )
   (all i: Identifier | lone i.(d.routing) )

-- B bindings are invertible. Note that routing does not need to be
-- invertible because dest retains the relevant history at the absorber.
   all i: Identifier | lone (d.BdstBinding).i

-- The source binding inverts (transposes) the BdstBinding.
   d.srcBinding = ~(d.BdstBinding)

-- Pattern A bindings precede Pattern B bindings.
   no ( BRan & ADom )
} }

pred RealStructure [d: Domain3, p: Path3] {
   StructuredDomain[d] &&
   Domain3SupportsPath[d,p] &&
   p.generator != p.absorber &&
   (some i1, i2: Identifier |
      i1 = (p.source).(d.srcBinding) && p.finSrc = i1.(d.srcBinding) &&
      i2 = (p.origDst).(d.AdstBinding) && p.dest = i2.(d.BdstBinding)
   )
}
```

```
/* ========================================================================
SINGLE-DOMAIN RETURNABILITY VERIFICATION
======================================================================== */

pred ReturnableDomain [d: Domain3] {
-- If there is a terminating attempt to return a path, it must go to
-- the generator of the message being returned.
(   all p1, p2: Path3 |
      Domain3SupportsPath[d,p1] && Domain3SupportsPath[d,p2] &&
      ReturnPath[p1,p2]
   => p2.absorber = p1.generator
) &&
-- If there is an attempt to return a path, it must terminate.
   NonloopingDomain[d] &&
(  all p1: Path3 | Domain3SupportsPath[d,p1] =>
      (all a: Address |
         a in (p1.finSrc).(*(d.dstBinding)) &&
         a !in (d.dstBinding).Identifier
      => a in (d.routing).Agent )
)
}


/* ========================================================================
DOUBLE-DOMAIN RETURNABILITY VERIFICATION
======================================================================== */

pred MobileAgentMove [g: Agent, a1, a2: Address, d1, d2: Domain3] { {
-- Preconditions:
-- a1 is the result of a B binding.
   a1 in Identifier.(d1.BdstBinding)
-- a1 is not in the domain of a B binding.
   a1 !in (d1.BdstBinding).Identifier
-- a1 routes to g.
   a1.(d1.routing) = g
-- a2 is unused.
   IdentifiersUnused[d1,a2]

-- Postconditions:
-- Update the domain, assuming structure.
   (let a3 = (d1.BdstBinding).a1 |
        d2.routing = d1.routing + (a2->g) - (a1->g) &&
        d2.BdstBinding = d1.BdstBinding + (a3->a2) - (a3->a1) &&
        d2.srcBinding = d1.srcBinding + (a2->a3) - (a1->a3)
      )
-- Frame conditions on domain parts that don't change:
   d2.endpoints = d1.endpoints
   d2.space = d1.space
   d2.AdstBinding = d1.AdstBinding
} }
```

```
pred RealMove [g: Agent, a1, a2: Address, d1, d2: Domain3, p1: Path3] { {
   StructuredDomain[d1]
   MobileAgentMove[g,a1,a2,d1,d2]
   Domain3SupportsPath[d1,p1]
   p1.generator = g
   p1.generator != p1.absorber
   p1.source = a1
   (some i1, i2: Identifier |
      i1 = (p1.source).(d1.srcBinding) &&
      p1.finSrc = i1.(d1.srcBinding) &&
      i2 = (p1.origDst).(d1.AdstBinding) &&
      p1.dest = i2.(d1.BdstBinding)
   )
} }


pred MoveThenReturn [g: Agent, disj a1, a2: Address, d1, d2: Domain3, p1, p2:
    Path] { {
   StructuredDomain[d1]
   MobileAgentMove[g,a1,a2,d1,d2]
   Domain3SupportsPath[d1,p1]
   Domain3SupportsPath[d2,p2]
   ReturnPath[p1,p2]
   p1.generator = g
   p1.generator != p1.absorber
   p1.source = a1
   (some i1, i2: Identifier |
      i1 = (p1.source).(d1.srcBinding) &&
      p1.finSrc = i1.(d1.srcBinding) &&
      i2 = (p1.origDst).(d1.AdstBinding) &&
      p1.dest = i2.(d1.BdstBinding)
   )
} }

-- checked, this is the smallest possible scope



pred ReturnableDomainPair [d1, d2: Domain3] {
-- If there is a "return path", it must truly return.
(   all p1, p2: Path3 |
      Domain3SupportsPath[d1,p1] && Domain3SupportsPath[d2,p2] &&
      ReturnPath[p1,p2]
   => p2.absorber = p1.generator
) &&
-- If there is an attempt to return a path, it must terminate.
   NonloopingDomain[d2] &&
(  all p1: Path3 | Domain3SupportsPath[d1,p1] =>
      (all a: Address |
         a in (p1.finSrc).(*(d2.dstBinding)) &&
         a !in (d2.dstBinding).Identifier
      => a in (d2.routing).Agent )
)
```

```
}

assert StructureSufficientForPairReturnability {
   all g: Agent, a1, a2: Address, d1, d2: Domain3 |
      StructuredDomain[d1] &&
      MobileAgentMove[g,a1,a2,d1,d2]
      => ReturnableDomainPair[d1,d2]
}

check StructureSufficientForPairReturnability for 2 but
   2 Domain, 2 Path, 3 Agent, __SCOPE__ Identifier
```

Apéndice F

STABLE MUTEX RING

```
module examples/algorithms/stable_mutex_ring

/*
 * Dijkstra's K-state mutual exclusion algorithm for a ring
 *
 * Original paper describing the algorithm:
 *   [1] E.W.Dijkstra, "Self-Stabilizing Systems in Spite of
 *   Distributed Control", Comm. ACM, vol. 17, no. 11, pp.
 *   643-644, Nov. 1974
 *
 * Proof of algorithm's correctness:
 *   [2] E.W.Dijkstra, "A Belated Proof of Self-Stabilization",
 *   in Distributed Computing, vol. 1, no. 1, pp. 5-6, 1986
 *
 * SMV analysis of this algorithm is described in:
 *   [3] "Symbolic Model Checking for Self-Stabilizing Algorithms",
 *   by Tatsuhiro Tsuchiya, Shini'ichi Nagano, Rohayu Bt Paidi, and
 *   Tohru Kikuno, in IEEE Transactions on Parallel and Distributed
 *   Systems, vol. 12, no. 1, January 2001
 *
 * Description of algorithm (adapted from [3]):
 *
 * Consider a distributed system that consists of n processes
 * connected in the form of a ring. We assume the state-reading
 * model in which processes can directly read the state of their
 * neighbors. We define _privilege_ of a process as its ability to
 * change its current state. This ability is based on a Boolean
 * predicate that consists of its current state and the state of
 * one of its neighboring processes.
 *
 * We then define the legitimate states as those in which the
 * following two properties hold: 1) exactly one process has a
 * privilege, and 2) every process will eventually have a privilege.
 * These properties correspond to a form of mutual exclusion, because
 * the privileged process can be regarded as the only process that is
 * allowed in its critical section.
 *
 * In the K-state algorithm, the state of each process is in
 * {0,1,2,...,K-1}, where K is an integer larger than or equal to n.
 * For any process p_i, we use the symbols S and L to denote its
 * state and the state of its neighbor p_{i-1}, respectively, and
 * process p_0 is treated differently from all other processes. The
 * K-state algorithm is described below.
 *
 *   process p_0: if (L=S) { S := (S+1) mod K; }
 *   process P_i(i=1,...,n-1): if (L!=S) { S:=L; }
```

```
 */

open util/ordering[Tick] as to
open util/graph[Process] as pg
open util/graph[Val] as vg

sig Process {
  rightNeighbor: Process
}

sig Val {
  nextVal : Val
}

fact MoreValThanProcess {
  # Val > # Process
}

fact DefineRings {
  pg/ring[rightNeighbor]
  vg/ring[nextVal]
  //Val$nextVal = Ord[Val].next + (Ord[Val].last -> Ord[Val].first)
}

sig Tick {
  val: Process -> one Val,
  runs: set Process,  // processes scheduled to run on this tick
  // for visualization
  priv: set Process // the set of priviledged processes on this tick
}
{
  priv = { p : Process | Privileged[p, this] }
}

one sig FirstProc extends Process {
}


fun FirstProcTrans[curVal, neighborVal : Val]: Val {
  (curVal = neighborVal) => curVal.nextVal else curVal
}

fun RestProcTrans[curVal, neighborVal : Val]: Val {
  (curVal != neighborVal) => neighborVal else curVal
}

fact LegalTrans {
  all tp : Tick - to/last |
    let tn = to/next[tp] | {
        all p: Process |
           let curVal = tp.val[p], neighborVal = tp.val[p.rightNeighbor], newVal =
                tn.val[p] | {
               p !in tp.runs => newVal = curVal else {
```

```
                p = FirstProc =>
                    newVal = FirstProcTrans[curVal, neighborVal]
                else
                    newVal = RestProcTrans[curVal, neighborVal]
            }
        }
    }
}

pred TickTrans[tp, tn : Tick] {
  all p : Process |
    let curVal = tp.val[p], neighborVal = tp.val[p.rightNeighbor], newVal = tn.
        val[p] | {
                p = FirstProc =>
                    newVal = FirstProcTrans[curVal, neighborVal]
                else
                    newVal = RestProcTrans[curVal, neighborVal]
    }
}

pred Privileged[p : Process, t : Tick] {
  // whether this process can enter its critical section
  // on this tick
  p = FirstProc =>
    t.val[p] = t.val[p.rightNeighbor]
  else
    t.val[p] != t.val[p.rightNeighbor]
}

pred IsomorphicStates[val1, val2: Process -> one Val] {
   some processMap: Process one -> one Process,
        valMap: Val one -> one Val | {
       FirstProc.processMap = FirstProc
       all p: Process, v: Val | {
          p->v in val1 iff (p.processMap) -> (v.valMap) in val2
       }
       all v1,v2: Val | v1->v2 in nextVal iff (v1.valMap) -> (v2.valMap) in
           nextVal
       all p1,p2: Process | p1->p2 in rightNeighbor
              iff (p1.processMap) -> (p2.processMap) in rightNeighbor
   }
}

pred BadSafetyTrace {
  // Find a trace that goes into a loop
  // containing a bad tick, i.e. a tick
  // at which two distinct processes
  // try to run their critical sections
  // simultaneously. In such a trace the
  // algorithm never "stabilizes".
  let lst = to/last |
    some t : Tick - lst | {
      //IsomorphicStates(ft.val, lst.val)
```

```
      t.val = lst.val
      Process in (to/nexts[t] + t - lst).runs
      some badTick : to/nexts[t] + t |
        BadTick[badTick]
    }
}

pred BadTick[badTick : Tick] {
      // Two different processes simultaneously
      // try to run their critical sections at this tick
      some p1 , p2 : Process | {
        p1!=p2
        Privileged[p1, badTick]
        Privileged[p2, badTick]
      }
}

assert Closure {
  not BadTick[to/first] => (all t : Tick | not BadTick[t])
}

pred TwoPrivileged {
  BadTick[to/first]
  some p1, p2 : Process, t1, t2 : Tick - to/first | {
    p1!=p2
    Privileged[p1,t1]
    Privileged[p2,t2]
  }
}

pred TraceWithoutLoop {
  all t1, t2 : Tick | t1!=t2 => t1.val != t2.val
}

pred TraceShorterThanMaxSimpleLoop {
  to/first.val = to/last.val
  all t : Tick - to/first - to/last |
    !(t.val = to/first.val)
}


check Closure for 5 but __SCOPE__ Process, __SCOPE+1__ Val expect 0
```

Bibliografía

[18t15] 18th International Conference on Theory and Applications of Satisfiability Testing: *SAT-Race 2015*, 2015. http://baldur.iti.uka.de/sat-race-2015/, visitado el 2015-06-19.

[AS09] Audemard, Gilles y Laurent Simon: *Predicting Learnt Clauses Quality in Modern SAT Solvers*. En *IJCAI 2009, Proceedings of the 21st International Joint Conference on Artificial Intelligence, Pasadena, California, USA, July 11-17, 2009*, páginas 399–404, 2009. http://ijcai.org/papers09/Papers/IJCAI09-074.pdf.

[BBH+09] Biere, A., A. Biere, M. Heule, H. van Maaren y T. Walsh: *Handbook of Satisfiability: Volume 185 Frontiers in Artificial Intelligence and Applications*. IOS Press, Amsterdam, The Netherlands, The Netherlands, 2009, ISBN 1586039296, 9781586039295.

[BCCZ99] Biere, Armin, Alessandro Cimatti, Edmund M. Clarke y Yunshan Zhu: *Symbolic Model Checking Without BDDs*. En *Proceedings of the 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, TACAS '99, páginas 193–207, London, UK, UK, 1999. Springer-Verlag, ISBN 3-540-65703-7. http://dl.acm.org/citation.cfm?id=646483.691738.

[Bie10] Biere, Armin: *Lingeling, Plingeling, PicoSAT and PrecoSAT at SAT Race 2010*. Informe técnico, 2010.

[BS97] Bayardo, Jr., Roberto J. y Robert C. Schrag: *Using CSP Look-back Techniques to Solve Real-world SAT Instances*. En *Proceedings of the Fourteenth National Conference on Artificial Intelligence and Ninth Conference on Innovative Applications of Artificial Intelligence*, AAAI'97/IAAI'97, páginas 203–208. AAAI Press, 1997, ISBN 0-262-51095-2. http://dl.acm.org/citation.cfm?id=1867406.1867438.

[Coo71] Cook, Stephen A.: *The Complexity of Theorem-proving Procedures*. En *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, STOC '71, páginas 151–158, New York, NY, USA, 1971. ACM. http://doi.acm.org/10.1145/800157.805047.

[Dij74] Dijkstra, Edsger W.: *Self-stabilizing Systems in Spite of Distributed Control*. Commun. ACM, 17(11):643–644, Noviembre 1974, ISSN 0001-0782. http://doi.acm.org/10.1145/361179.361202.

[DIM93] *Satisfiability Suggested Format*, May 1993. http://www.cs.ubc.ca/~hoos/SATLIB/Benchmarks/SAT/satformat.ps.

[DLL62] Davis, Martin, George Logemann y Donald Loveland: *A Machine Program for Theorem-proving*. Commun. ACM, 5(7):394–397, Julio 1962, ISSN 0001-0782. http://doi.acm.org/10.1145/368273.368557.

[DP60] Davis, Martin y Hilary Putnam: *A Computing Procedure for Quantification Theory*. J. ACM, 7(3):201–215, Julio 1960, ISSN 0004-5411. http://doi.acm.org/10.1145/321033.321034.

[GSK98] Gomes, Carla P., Bart Selman y Henry Kautz: *Boosting Combinatorial Search Through Randomization*. En *Proceedings of the Fifteenth National/Tenth Conference on Artificial Intelligence/Innovative Applications of Artificial Intelligence*, AAAI '98/IAAI '98, páginas 431–437, Menlo Park, CA, USA, 1998. American Association for Artificial Intelligence, ISBN 0-262-51098-7. http://dl.acm.org/citation.cfm?id=295240.295710.

[HH14] Haim, Shai y Marijn Heule: *Towards Ultra Rapid Restarts*. CoRR, abs/1402.4413, 2014. http://arxiv.org/abs/1402.4413.

[HS09] Hamadi, Youssef y Lakhdar Sais: *ManySAT: a parallel SAT solver*. JOURNAL ON SATISFIABILITY, BOOLEAN MODELING AND COMPUTATION (JSAT), 6, 2009.

[Jac06] Jackson, Daniel: *Software Abstractions: Logic, Language, and Analysis*. Cambridge, MA: MIT Press, 2006. http://alloy.mit.edu.

[LSZ93] Luby, Michael, Alistair Sinclair y David Zuckerman: *Optimal Speedup of Las Vegas Algorithms*. Inf. Process. Lett., 47(4):173–180, Septiembre 1993, ISSN 0020-0190. http://dx.doi.org/10.1016/0020-0190(93)90029-9.

[Men97] Mendelson, Elliott: *Introduction to Mathematical Logic (Fourht Edition)*. Chapman & Hall, 1997.

[MF08] MPI-Forum, Message Passing Interface Forum: *MPI: A Message-Passing Interface Standard*. Informe técnico, University of Tennessee, Knoxville, TN, USA, 2008. http://www.mpi-forum.org/docs/mpi-2.1/mpi21-report.pdf.

[MFM04] Mahajan, Yogesh S., Zhaohui Fu y Sharad Malik: *Zchaff2004: An Efficient SAT Solver*. En *Theory and Applications of Satisfiability Testing, 7th International Conference, SAT 2004, Vancouver, BC, Canada, May 10-13, 2004, Revised Selected Papers*, páginas 360–375, 2004. http://dx.doi.org/10.1007/11527695_27.

[MMZ+01] Moskewicz, Matthew W., Conor F. Madigan, Ying Zhao, Lintao Zhang y Sharad Malik: *Chaff: Engineering an Efficient SAT Solver*. En *Proceedings of the 38th Annual Design Automation Conference*, DAC '01, páginas 530–535, New York, NY, USA, 2001. ACM, ISBN 1-58113-297-2. http://doi.acm.org/10.1145/378239.379017.

[MSS99] Marques-Silva, João P. y Karem A. Sakallah: *GRASP: A Search Algorithm for Propositional Satisfiability*. IEEE Trans. Comput., 48(5):506–521, Mayo 1999, ISSN 0018-9340. http://dx.doi.org/10.1109/12.769433.

[NE03] Niklas Eén, Niklas Sörensson: *An Extensible SAT-solver*. En Enrico Giunchiglia, Armando Tacchella (editor): *Theory and Applications of Satisfiability Testing*, 2003.

[SAT02] SAT Competition Organizing committee: *The international SAT Competitions web page*, 2002. `http://www.satcompetition.org/`, visitado el 2015-06-19.

[Sil99] Silva, João P. Marques: *The Impact of Branching Heuristics in Propositional Satisfiability Algorithms*. En *Proceedings of the 9th Portuguese Conference on Artificial Intelligence: Progress in Artificial Intelligence*, EPIA '99, páginas 62–74, London, UK, UK, 1999. Springer-Verlag, ISBN 3-540-66548-X. `http://dl.acm.org/citation.cfm?id=645377.651196`.

[SS96] Silva, Jo
btxfnamespacelong ao P. Marques y Karem A. Sakallah: *GRASP&Mdash;a New Search Algorithm for Satisfiability*. En *Proceedings of the 1996 IEEE/ACM International Conference on Computer-aided Design*, ICCAD '96, páginas 220–227, Washington, DC, USA, 1996. IEEE Computer Society, ISBN 0-8186-7597-7. `http://dl.acm.org/citation.cfm?id=244522.244560`.

[TNPK01] Tsuchiya, Tatsuhiro, Shin'ichi Nagano, Rohayu Bt Paidi y Tohru Kikuno: *Symbolic Model Checking for Self-Stabilizing Algorithms*. IEEE Trans. Parallel Distrib. Syst., 12(1):81–95, Enero 2001, ISSN 1045-9219. `http://dx.doi.org/10.1109/71.899941`.

[Vis13] Vissani, Ignacio: *Un sat-solver paralelo y distribuido con herencia de clausulas aprendidas*. Tesis de Licenciatura, Facultad de Ciencias Exactas y Naturales, Universidad de Buenos Aires, 2013.

[XHHLB08] Xu, Lin, Frank Hutter, Holger H. Hoos y Kevin Leyton-Brown: *SATzilla: Portfolio-based Algorithm Selection for SAT*. J. Artif. Int. Res., 32(1):565–606, Junio 2008, ISSN 1076-9757. `http://dl.acm.org/citation.cfm?id=1622673.1622687`.

[Zav06] Zave, Pamela: *Compositional Binding in Network Domains*. En *Proceedings of the 14th International Conference on Formal Methods*, FM'06, páginas 332–347, Berlin, Heidelberg, 2006. Springer-Verlag, ISBN 3-540-37215-6, 978-3-540-37215-8. `http://dx.doi.org/10.1007/11813040_23`.

[ZBP+96] Zhang, Hantao, Maria Paola Bonacina, Maria Paola, Bonacina y Jieh Hsiang: *PSATO: a Distributed Propositional Prover and Its Application to Quasigroup Problems*. Journal of Symbolic Computation, 21:543–560, 1996.