SPUR Documentation

Zayd Hammoudeh

Feb 13, 2019

Contents:

1 Library API **3**
- 1.1 Class Hierarchy . . . 3
- 1.2 File Hierarchy . . . 3
- 1.3 Full API . . . 3

2 Indices and tables **233**

Developer: Zayd Hammoudeh (zayd.hammoudeh@gmail.com)
Source Code

CHAPTER 1

Library API

1.1 Class Hierarchy

1.2 File Hierarchy

1.3 Full API

1.3.1 Classes and Structs

Struct SolverConfiguration

- Defined in *File solver_config.h*

Struct Documentation

struct SolverConfiguration

solver_config.h

Purpose: Defines the *SolverConfiguration()* class stores the state of the sampler and counter.

Author Zayd Hammoudeh zayd@ucsc.edu

Version 0.00.00

Original Author: Marc Thurley. Singleton object used for storing the configuration settings for the current execution.

Public Functions

bool **store_sampled_models**()
When random sampling is enabled, this flag indicates whether the models should currently be stored or other processing is occurring.

Return True if sample models should be stored.

bool **perform_sample_caching**()
Cache Sampling State Accessor

Returns whether the cache currently supports sampling.

Return true if samples are being stored in the cache.

void **EnableSampleCaching**()
Sample Caching Enabler

This enables storing of samples in the cache. It can only be enabled. It can never be disabled by design since once the caching stage has been enabled, it can never be disabled.

Public Members

bool **perform_component_caching** = true
Support to modify the default state of this feature is disabled in the sampler.

Forces variable backtracking in the case of a conflict being founded. Support to modify the default state of this feature is disabled in the sampler.

bool **perform_failed_lit_test** = true
Support to modify the default state of this feature is disabled in the sampler.

bool **perform_pre_processing** = true
Support to modify the default state of this feature is disabled in the sampler.

bool **perform_random_sampling_** = true
Rather than performing standard model counting, sample satisfying models uniformly at random.

bool **perform_top_tree_sampling** = false
Performs sampling of the top of the tree.

bool **disable_samples_write_** = false
Disable writing the samples file.

std::string **samples_output_file** = ""
Location to write the output sample and sampler information.

bool **debug_mode** = false
Debug mode has special settings to make debugging the program easier.

uint64_t **time_bound_seconds** = UINT64_MAX
Stores the maximum execution time of the sampler.

Default is approximately 27 hours.

bool **verbose** = false
Controls whether verbose state tracking is enabled. When verbose is enabled, a basic trace printing is also enabled.

In some versions of the console, this verbose printing may be multicolored for easier visual tracking.

bool **quiet** = false
Prevents printing to the console.

Cannot be true if verbose is true.

bool **perform_two_pass_sampling_** = false
Forces two pass sample construction. This applies only to the case where the number of samples requested is 1.

unsigned **num_samples_to_cache_** = 1
Number of samples that will be cached at once.

bool **skip_partial_assignment_fill** = false
DEBUG_ONLY MODE - This is an unsupported feature used for debug and development purposes only. Almost no one should use it beyond the core development team.

SampleSize **num_samples_** = 0
Number of satisfying assignments to randomly sample. This is a command line parameter.

TreeNodeIndex **max_top_tree_depth_** = 25
When trimming the top of the tree, this represents the maximum branch variable depth before sample building stops.

SampleSize **max_top_tree_leaf_sample_count** = 50
Maximum number of samples that can come from a single top tree node.

"__final_top_tree_samples.txt"]Location to write the top tree samples.

Struct Variable

- Defined in *File structures.h*

Struct Documentation

struct Variable
Simple variable structure used for storing variable related information including its antecedent and the decision level in the stack.

Public Members

Antecedent **ante**

long **decision_level** = -1

Class AltComponentAnalyzer

- Defined in *File alt_component_analyzer.h*

Class Documentation

class AltComponentAnalyzer
alt_component_analyzer.h

Purpose: Defines the *AltComponentAnalyzer()* class that analyzes a Boolean formula for connected components.

Author Zayd Hammoudeh zayd@ucsc.edu

Version 0.00.00

Original Author: Marc Thurley.

Public Functions

AltComponentAnalyzer(*DataAndStatistics* &*statistics*, *LiteralIndexedVector*<*TriValue*> &*lit_values*)

const unsigned **scoreOf**(*VariableIndex v*) **const**

ComponentArchetype &**current_archetype**()

void **initialize**(*LiteralIndexedVector*<*Literal*> &*literals*, std::vector<*LiteralID*> &*lit_pool*)
alt_component_analyzer.cpp

Purpose: Defines the methods for the *AltComponentAnalyzer()* class.

Author Zayd Hammoudeh zayd@ucsc.edu

Version 0.00.00

Original Author: Marc Thurley.

bool **isUnseenAndActive**(*VariableIndex v*)

bool **manageSearchOccurrenceOf**(*LiteralID lit*)

bool **manageSearchOccurrenceAndScoreOf**(*LiteralID lit*)

void **setSeenAndStoreInSearchStack**(*VariableIndex v*)

void **setupAnalysisContext**(*StackLevel* &*top*, *Component* &*super_comp*)

bool **exploreRemainingCompOf**(*VariableIndex v*)

Component ***makeComponentFromArcheType**()

unsigned **max_clause_id**()
Maximum clause ID number.

This is used to determine the packing size (m).

Return Maximum clause identification number

unsigned **max_variable_id**()
Maximum variable ID number.

This is used to determine the packing size (n).

Return Maximum variable identification number

ComponentArchetype &**getArchetype**()

Class Antecedent

- Defined in *File structures.h*

Class Documentation

class Antecedent

Antecedent can be either a clause or a literal.

Public Functions

Antecedent()

Creates an empty antecedent clause with ID zero.

Antecedent(**const** *ClauseOfs cl_ofs*)

Shifts the clause offset by 1 to the left and then adds 1.

Parameters

- `cl_ofs`: Clause offset.

Antecedent(**const** *LiteralID idLit*)

bool **isAClause**() **const**

Checks if the antecendent is a clause. This is done by checking if the least significant bit is a 0b0 or a 0b1.

Return True if the antecedent is a clause. False if it is a literal.

ClauseOfs **asCl**() **const**

Returns the antecedent information formatted as a clause

Return *Antecedent* clause

LiteralID **asLit**()

Returns the antecedent information formatted as a literal.

Return *Antecedent* literal

bool **isAnt**() **const**

MT - A NON-Antecedent will only be A NOT_A_CLAUSE Clause Id

Checks whether this is an actual antecedent.

Return true if the this is not a true antecendent (i.e., the value equals 1).

Class BasePackedComponent

- Defined in *File base_packed_component.h*

Inheritance Relationships

Derived Type

- `public DifferencePackedComponent` (*Class DifferencePackedComponent*)

Class Documentation

class BasePackedComponent

Stores the specifications for package cache components including the number of bits in a variable, clause, and other information.

Subclassed by *DifferencePackedComponent*

Public Functions

BasePackedComponent()

BasePackedComponent(unsigned *creation_time*)

Creates a packed component and records with it the creation time.

Parameters

- `creation_time`:

~BasePackedComponent()

unsigned **creation_time**()

const mpz_class &**model_count**() **const**

unsigned **alloc_of_model_count**() **const**

void **set_creation_time**(unsigned *time*)

void **set_model_count**(**const** mpz_class &*rn*, unsigned *time*)

unsigned **hashkey**() **const**

bool **modelCountFound**()

bool **isDeletable**() **const**

Determines if a cache entry is deletable. This entails that the entry is not connected to an active component in the component stack.

Return true if the cache entry is disconnected from an active component and can be deleted.

void **set_deletable**()

Mark the packed cache entry as deletable.

void **clear**()
Free the memory associated with the packed cache entry's data.

Public Static Functions

static unsigned **bits_per_variable**()
Accessor to get n := {lg |var(F)|}, i.e., the number of bits that represent a VARIABLE.

Return Number of bits to store a VARIABLE

static unsigned **variable_mask**()

static unsigned **bits_per_clause**()
Accessor to get m := {lg |cl(F)|}, i.e., the number of bits that represent a CLAUSE.

Return Number of bits to store a CLAUSE.

static unsigned **bits_per_block**()

static unsigned **bits_of_data_size**()

void **adjustPackSize**(unsigned int *maxVarId*, unsigned int *maxClId*)
Configures that variables that define the packing size variables.

Parameters

- `maxVarId`: Maximum ID number for a variable
- `maxClId`: Maximum ID number for a clause

static void **outbit**(unsigned *v*)
Prints a bit string to the console. The MSB is printed first and the LSB last.

Parameters

- `v`: Value to be printed by bit to the console.

static unsigned **log2**(unsigned *v*)

Public Static Attributes

unsigned **_debug_static_val** = 0

Protected Attributes

unsigned ***data_** = nullptr
Packed data array to be stored in memory,

data_ contains in packed form the variable indices and clause indices of the component ordered structure is

var var ... clause clause ...

clauses begin at clauses_ofs_

unsigned **hashkey_** = 0

mpz_class **model_count_**

unsigned **creation_time_** = 1

unsigned **length_solution_period_and_flags_** = 0

Protected Static Attributes

unsigned **_bits_per_clause** = 0

In the sharpSAT paper, bits per clause (m) is defined as:

m := {lg |cl(F)|}

It is the minimum number of bits per clause when packing is enabled.

The implementation is slightly different in the code. It is defined as:

m := {lg |cl(F)| + 1}

base_packed_component.cpp

Purpose: Defines the static variables and methods for the *BasePackedComponent()* class.

Author Zayd Hammoudeh zayd@ucsc.edu

Version 0.00.00

Original Author: Marc Thurley.

unsigned **_bits_per_variable** = 0

In the sharpSAT paper, bits per variable (n) is defined as:

n := {lg |var(F)|}

It is the minimum number of bits per clause when packing is enabled.

The implementation is slightly different in the code. It is defined as:

n := {lg |var(F)| + 1}

unsigned **_bits_of_data_size** = 0

unsigned **_data_size_mask** = 0

unsigned **_variable_mask** = 0

Bit mask in the form of bits of 0b1 of length "n"

See *BasePackedComponent::_bits_per_variable*.

unsigned **_clause_mask** = 0

Bit mask in the form of bits of 0b1 of length "m"

See *BasePackedComponent::_bits_per_variable*.

const unsigned **_bits_per_block** = (sizeof(unsigned) << 3)
Equals 32 (= 4 * 8 bits)

sizeof(unsigned) = 4 Three left bit shifts equals 8

Template Class BitStuffer

- Defined in *File base_packed_component.h*

Class Documentation

template <class *T*>
class BitStuffer
base_packed_component.h

Purpose: Defines the classes *BitStuffer()* and *BasePackedComponent()*.

Copyright (C) 2018 Zayd Hammoudeh. All rights reserved.

Author Zayd Hammoudeh zayd@ucsc.edu

Version 0.00.00

This software may be modified and distributed under the terms of the MIT license. See the LICENSE file for details.

Original Author: Marc Thurley. Manages the data in a byte array. It shift the bits into the array of objects of type T.

Template Parameters

- `T`: Generally the "unsigned" type.

Public Functions

BitStuffer(T **data*)

void **stuff**(**const** unsigned *val*, **const** unsigned *num_bits_val*)
Packs the specified signal "val" into the memory.

Parameters

- `val`: Binary value to pack into memory.
- `num_bits_val`: The expected bit length of val.

void **assert_size**(unsigned *size*)
Verify the specified size matches the actual contents of the BitStuff object.

Parameters

- `size`: number of bytes between the data_start_ pointer and the end of the BitStuff managed area.

Class CachedAssignment

- Defined in *File cached_assignment.h*

Class Documentation

class CachedAssignment

Public Functions

CachedAssignment()

void **Sort**()
Sort the list of literals for easier management.

void **IncreaseSize**(*VariableIndex* *size_adder*)
Increases the size of the literals list by the specified amount.

Parameters

- `size_adder`: The amount the literals list will be increased.

const std::vector<*VariableIndex*> **vars**()
Add the specified literal to the list of literals in the cache assignment.

Builds a list of variables in the cached assignment.

Parameters

- `lit`: New literal value. *Variable* List Extractor

Return List of variable numbers in this assignment.

const *VariableIndex* **num_components**() **const**
Stores the total number of components that make up this cached assignment.

Return Number of different components that make up this assignment.

const bool **empty**() **const**
An empty assignment has no associated components.

Return true if the assignment is empty.

void **ProcessComponent**(**const** *Component* *comp, mpz_class *model_count_and_assn*)
Processes a cached component and incorporates its information into the cached assignment.

Parameters

- `comp`: *Component* to be processed
- `model_count_and_assn`: *Component* unshifted model count. It contains the component assignment in the lower *n* bits where *n* is the number of bits in the component.

const std::vector<*LiteralID*> &**literals**() **const**
Accessor for the literals in the cached assignment.

Return List of literals in the cached assignment.

const std::vector<*VariableIndex*> &**emancipated_vars**() **const**
Accessor for get the set of emancipated variabes in this cached assignment.

Return Emancipated variables in the cached assignment.

void **clear**()
Resets the component and deletes all associated component assignment information including the number of components and the assigned literals.

Class ClauseHeader

- Defined in *File structures.h*

Class Documentation

class ClauseHeader

Public Functions

void **increaseScore**()

void **decayScore**()

unsigned **score**() **const**

unsigned **creation_time**()

unsigned **length**()

void **set_length**(unsigned *length*)

void **set_creation_time**(unsigned *time*)

Public Static Functions

static unsigned **overheadInLits**()

Class Component

- Defined in *File component.h*

Class Documentation

class Component
cacheable_component.h

Purpose: Defines the *Component()* class.

Author Zayd Hammoudeh zayd@ucsc.edu

Version 0.00.00

Original Author: Marc Thurley.

Public Functions

void **reserveSpace**(unsigned int *num_variables*, unsigned int *num_clauses*)
The size is increased by 2 because the objects are surrounded by two since the storage format is:

```
variables SENTINEL clauses SENTINEL
```

Parameters

- `num_variables`: Number of variables in the component
- `num_clauses`: Number of long (i.e., non-unary and non-binary) clauses in the component.

void **set_id**(*CacheEntryID id*)
Updates the component ID.

Parameters

- `id`: New component ID

CacheEntryID **id**() **const**

void **addVar**(**const** *VariableIndex var*)

void **closeVariableData**()

void **addCl**(**const** *ClauseIndex cl*)

void **closeClauseData**()

std::vector<*VariableIndex*>::const_iterator **varsBegin**() **const**

std::vector<*ClauseIndex*>::const_iterator **clsBegin**() **const**

unsigned **num_variables**() **const**

unsigned **numLongClauses**() **const**

bool **empty**() **const**
Empty *Component* Checker

Checks whether this component has any variables or clauses.

Return True if the component has at least one clause or component.

void **createAsDummyComponent**(unsigned *max_var_id*, unsigned *max_clause_id*)

void **clear**()

unsigned **clauses_ofs**() **const**
Accesor the clause offset number.

Return Clause offset number in the data structure.

const std::vector<*ComponentVarAndCls*> &**getData**() **const**
Component Data Accessor

This function exposes the internal "data_" object of the component. It is not a copy so changes to this object would corrupt the component. Care should be taken when using it.

Return Reference to the internal data of the component

Class ComponentArchetype

- Defined in *File component_archetype.h*

Class Documentation

class ComponentArchetype

Public Functions

ComponentArchetype()

ComponentArchetype(*StackLevel* &*stack_level*, *Component* &*super_comp*)

void **reInitialize**(*StackLevel* &*stack_level*, *Component* &*super_comp*)

Component &**super_comp**()

StackLevel &**stack_level**()

void **setVar_in_sup_comp_unseen**(*VariableIndex* *v*)

void **setClause_in_sup_comp_unseen**(*ClauseIndex* *cl*)

void **setVar_nil**(*VariableIndex* *v*)

void **setClause_nil**(*ClauseIndex* *cl*)

void **setVar_seen**(*VariableIndex* *v*)

void **setClause_seen**(*ClauseIndex* *cl*)

void **setClause_seen**(*ClauseIndex* *cl*, bool *all_lits_act*)

void **setVar_in_other_comp**(*VariableIndex* *v*)

void **setClause_in_other_comp**(*ClauseIndex* *cl*)

bool **var_seen**(*VariableIndex* *v*) **const**

bool **clause_seen**(*ClauseIndex* *cl*) **const**

bool **clause_all_lits_active**(*ClauseIndex* *cl*)

void **setClause_all_lits_active**(*ClauseIndex* *cl*) **const**

bool **var_nil**(*VariableIndex v*) **const**

bool **clause_nil**(*ClauseIndex cl*) **const**

bool **var_unseen_in_sup_comp**(*VariableIndex v*) **const**

bool **clause_unseen_in_sup_comp**(*ClauseIndex cl*) **const**

bool **var_seen_in_peer_comp**(*VariableIndex v*) **const**

bool **clause_seen_in_peer_comp**(*ClauseIndex cl*) **const**

Component ***makeComponentFromState**(unsigned long *stack_size*)

Public Members

Component **current_comp_for_caching_**

Public Static Functions

static void **initArrays**(unsigned *max_variable_id*, unsigned *max_clause_id*)

static void **clearArrays**()

Class ComponentCache

- Defined in *File component_cache.h*

Class Documentation

class ComponentCache

component_cache.h

Purpose: Defines the *ComponentCache()* class for storing components in the cache.

Author Zayd Hammoudeh zayd@ucsc.edu

Version 0.00.00

Original Author: Marc Thurley. Data structure that is used as the component cache for the number of solutions for different components.

Public Functions

ComponentCache(*DataAndStatistics* &*statistics*, *SolverConfiguration* &*config*)

component_cache.cpp

Purpose: Defines methods for the *ComponentCache()* class for storing components in the cache.

Author Zayd Hammoudeh zayd@ucsc.edu

Version 0.00.00

This software may be modified and distributed under the terms of the MIT license. See the LICENSE file for details.

Original Author: Marc Thurley.

~ComponentCache()

void **init**(*Component* &*super_comp*, bool *quiet* = false)

uint64_t **compute_byte_size_infrastructure**()

Rebuild the size of the cache infrastructure (in bytes) from scratch by computing the size of the individual component including:

- Size of the cache entry ID table (based on its capacity
- Size of the cacheable component area.
- Size of the cache entry slots used to store the list of free entries.

It also updates the statistics for the formula including the cache infrastructure bytes usage,

This is called when the cache entries are deleted.

Return Size of the cache infrastructure.

const *CacheableComponent* &**entry_const**(**const** *CacheEntryID id*) **const**

Access a reference to an entry in the component cache. If the specified entry ID does not exist, the program will crash. This is the constant version of the function and is safe to use in assert statements.

Return Reference to the specified cache entry ID.

Parameters

- `id`: Cache entry identification number

CacheableComponent &**entry**(**const** *CacheEntryID id*)

Access a reference to an entry in the component cache. If the specified entry ID does not exist, the program will crash.

Return Reference to the specified cache entry ID.

Parameters

- `id`: Cache entry identification number

bool **hasEntry**(*CacheEntryID id*) **const**

Uses a cached component's CacheEntryID to extract the components cached information.

Return Cache entry for the specified component. Determines whether the

Return

Parameters

- `comp`: *Component* to be extracted from the cache.

Parameters

- `id`: *removeFromHashTable(CacheEntryID id)*;

void **removeFromHashTable**(*CacheEntryID id*)
Removes the entry ID from the hash table but not from the entry base.

Parameters

- `id`: Identification number of the cache entry.

void **cleanPollutionsInvolving**(*CacheEntryID id*)

CacheEntryID **storeAsEntry**(*CacheableComponent* &*ccomp*, *CacheEntryID super_comp_id*)

bool **manageNewComponent**(*StackLevel* &*top*, *CacheableComponent* &*packed_comp*, *Component* **unpacked_comp*, *CachedAssignment* &*cached_assn*, *CacheEntryID* &*cache_entry_id*)
New *Component* Cache Checker

MT - Check quickly if the model count of the component is cached. If so, incorporate it into the model count of top If not, store the packed version of it in the entry_base of the cache

Return True if the new component is in cache.

Parameters

- `top`: Top of the literal decision stack.
- `packed_comp`: New component to check whether it is in cache.
- `cached_assn`: Cached assignment to be modified

void **eraseEntry**(*CacheEntryID id*)

void **storeValueOf**(*CacheEntryID id*, **const** mpz_class &*model_count*)
Stores the model count of the specified CacheEntryId item in the cache.

If may invoke a cache resize if the new entry cannot fit in the cache.

Parameters

- `id`: ID number of the cache entry.
- `model_count`: Number of models for the specified ID.

bool **deleteEntries**()
Purges old entries from the cache to get the cache size to roughly half of its current size.

It then rehashes the table and updates the cache size information.

Return Always true.

void **removeFromDescendantsTree**(*CacheEntryID id*)

void **test_descendantstree_consistency**()

void **debug_dump_data**()

Class ComponentManager

- Defined in *File component_management.h*

Class Documentation

class ComponentManager

Public Functions

ComponentManager(*SolverConfiguration* &*config*, *DataAndStatistics* &*statistics*, *LiteralIndexedVector*<*TriValue*> &*lit_values*)

void **initialize**(*LiteralIndexedVector*<*Literal*> &*literals*, std::vector<*LiteralID*> &*lit_pool*, bool *quiet* = false)

Initializes the cache, component stack, and component analyzer.

component_management.cpp

Parameters

- `literals`: Set of valid literals
- `lit_pool`:
- `quiet`: Quiet parameter used to set whether console printing is enabled.

Purpose: Defines methods for the ComponentManagement() class.

Author Zayd Hammoudeh zayd@ucsc.edu

Version 0.00.00

Original Author: Marc Thurley.

unsigned **scoreOf**(*VariableIndex* *v*)

void **cacheModelCountOf**(*VariableIndex* *stack_comp_id*, **const** mpz_class &*value*)

Cache Model Count Storer

If caching is enabled, this method stores the model count of the component onto the stack.

Parameters

- `stack_comp_id`: *Component* ID of the stack element
- `value`: Model count of the element

void **cacheModelCountAndAssignment**(**const** *VariableIndex* *stack_comp_id*, **const** mpz_class &*value*, **const** *SampleAssignment* &*assn*, **const** *Component* &*component*)

Cache Model Count and Assignment Encoder and Storer

Prepend the component's MPZ value with the sample assignment.

Parameters

- `stack_comp_id`: *Component* ID of the stack element.
- `value`: Model count
- `assn`: Sample assignment to encode in the cache with the model count.
- `component`: Actual component to get the variables.

Component &**superComponentOf**(*StackLevel* &*lev*)

Gets a pointer to the actual super component that corresponds to passed stack level.

Return Pointer to the super component of the stack item.

Parameters

- `lev`: Level in the stack whose super component will be extracted

const *Component* &**super_component**(*StackLevel* &*lev*) **const**

Const version of the *superComponentOf()* method.

Return Pointer to the super component of the stack item.

Parameters

- `lev`: Level in the stack whose super component will be extracted

VariableIndex **component_stack_size**() **const**

Extracts the number of components currently in the component stack.

Return Number of element in teh component stack.

void **cleanRemainingComponentsOf**(*StackLevel* &*top*)

const *Component* &**component**(unsigned long *comp_idx*) **const**

Return

Parameters

- `comp_idx`:

bool **findNextRemainingComponentOf**(*StackLevel* &*top*, **const** std::vector<*LiteralID*> &*literal_stack*, *SamplesManager* &*samples*)

Checks for the next yet to explore remaining component of top returns true if a non-trivial non-cached component has been found and is now stack_.TOS_NextComp() returns false if all components have been processed;

This function has been modified to support sampling. That includes passing the parameter "literal_stack". This is neeeded to build samples.

Return true if unprocessed components remain.

Parameters

- `top`: Top of the decision stack.
- `literal_stack_`: Current assigned literal stack.
- `samples`: Current reservoir sample set.

void **recordRemainingCompsFor**(*StackLevel* &*top*, **const** std::vector<*LiteralID*> &*literal_stack*)

Examines the top of the decision stack. The function will try to decompose the component if applicable. For any new component, the function will check if the new components exists in the cache; if it does, the results are used to update the super component.

Otherwise, the new component is added to the component stack (not the decision stack). The end of unprocessed components is also updated.

Component at the top of the stack is analyzed. If a new and unseen component has never been seen, then the new component is placed on the component stack.

Parameters

- `top`: Element at the top of the decision stack.
- `literal_stack`: Current assigned literal stack.

The component stack is then sorted so components with fewer variables are at the top of the stack.

Parameters

- `top`: Element at the top of the decision stack.
- `literal_stack_`: Current assigned literal stack.

void **buildResidualComponent**(*StackLevel* &*top*)

Build residual formula component. It uses the contents of the literal stack to decide how to construct remaining components.

Parameters

- `top`: Element at the top of the decision stack.

Component ***top_stack**()

Accessor for the top of the component stack.

Return Pointer to the component on top of the component stack.

void **sortComponentStackRange**(unsigned long *start*, unsigned long *end*)

Sorts the components stack from start (inclusive) to end (exclusive). The components with more variables are placed lower on the stack (i.e., further from the top) and processed later.

Parameters

- `start`: Start of component stack to be sorted (inclusive)
- `end`: End of the component stack to sort (exclusive)

void **gatherStatistics**()

void **removeAllCachePollutionsOf**(*StackLevel* &*top*)

std::vector<*VariableIndex*> **buildFreedVariableList**(**const** *StackLevel* &*top*, **const** std::vector<*LiteralID*> &*literal_stack*)

Freed *Variable* List Builder

When assigning variables in a formula it is common that some variables become free (i.e., no longer appear in any of the clauses). These free variables will not appear on the literal stack but must be tracked to create the sample. This function stores all such freed variables.

Return Vector of the variable indexes of all freed variables

Parameters

- `literal_stack`: Current literal stack

Class DataAndStatistics

- Defined in *File statistics.h*

Class Documentation

class DataAndStatistics

statistics.h

Purpose: Defines the *DataAndStatistics()* class.

Author Zayd Hammoudeh zayd@ucsc.edu

Version 0.00.00

Original Author: Marc Thurley.

Public Functions

DataAndStatistics()

void **UpdateNodeTypeStatistics**(*TopTreeNodeType node_type*, **const** mpz_class &*num_models*)

Updates the number of nodes and total model count based on the top tree node type.

Parameters

- `node_type`: Type of top tree node
- `num_models`: Number of new models for the specified node type.

TreeNodeIndex **num_tree_nodes**(*TopTreeNodeType node_type*)

Accesor for the number of top tree nodes of a specified type.

Return Total number of top tree nodes of the specified type.

Parameters

- `node_type`: Type of top tree node.

mpz_class **num_tree_node_models**(*TopTreeNodeType node_type*)

Accessor for the total number of models associated with a specified type of top tree node.

Return Total number of models associated with the specific top tree node type.

Parameters

- `node_type`: Type of top tree node.

double **`percent_tree_node_models`**(*TopTreeNodeType node_type*)
Calculates the percentage of all models that are of a specific type.

Return Percent of all models that fall under the specified node type

Parameters

- `node_type`: Type of top tree node (e.g., cylinder, max depth, component split, etc.)

void **`reset_statistics`**()

bool **`cache_full`**() **`const`**
Checks whether the cache is full (i.e., exceeds 100% capacity).

Return true if the cache is full and false otherwise.

uint64_t **`cache_bytes_memory_usage`**() **`const`**

Return Sum of the infrastructure bytes and the sized of the cached components.

uint64_t **`overall_cache_bytes_memory_stored`**()

void **`incorporate_cache_store`**(*CacheableComponent* &*ccomp*)

void **`incorporate_cache_erase`**(*CacheableComponent* &*ccomp*)

void **`incorporate_cache_hit`**(*CacheableComponent* &*ccomp*)

unsigned long **`cache_MB_memory_usage`**()

double **`implicitBCP_miss_rate`**()

unsigned long **`num_clauses`**() **`const`**

unsigned long **`num_conflict_clauses`**()

unsigned long **`clause_deletion_interval`**()

void **`set_final_solution_count`**(**`const`** mpz_class &*count*)
Updates the solver field "final_solution_count_" with the solution count multiplied by 2^{#unused variables}. Hence:

final_solution_count_ = count * 2^(num_variables_ - num_used_variables_)

Parameters

- `count`: model count.

`const` mpz_class &**`final_solution_count`**() **`const`**
Accessor for the final solution count

Return Final solution count.

void **`incorporateClauseData`**(**`const`** std::vector<*LiteralID*> &*clause*)

void **printShort**()
statistics.cpp

Purpose: Defines methods for the *DataAndStatistics()* class.

Author Zayd Hammoudeh zayd@ucsc.edu

Version 0.00.00

Original Author: Marc Thurley.

void **printShortFormulaInfo**()
Prints to the console information about the formula including: number of long, binary, & unit clauses, number of used variables, and total number of variables.

unsigned long **getTime**() **const**
Number of decisions (i.e., branch variable selections) made. This is used for determining when to remove components from the cache.

Return Number of branch variable selections

long double **getAvgComponentSize**()

unsigned long **cached_component_count**()
Access the number of components in the cache.

Return

unsigned long **cache_hits**()
Number of cache look-ups where the component requested was in the cache.

Return Cache look-up count.

double **cache_miss_rate**()
The rate of cache look-ups where the requested component was not in the cache.

Return 0 if a cache look-up never occurred. Otherwise, it is the number of caches misses divided by the total number of cache look-ups.

long double **getAvgCacheHitSize**()
The average size of a reused (i.e., hit) components in the cache.

Return 0 if a cache hit has never occurred. Otherwise, it is the average size of cached components that have ever been HIT.

Public Members

std::string **input_file_**
Path to the file containing the CNF formula used in the current run.

time_t **start_time_**

uint64_t **maximum_cache_size_bytes_** = 0

SolverExitState **exit_state_** = SolverExitState::NO_STATE

VariableIndex **num_original_variables_** = 0

ClauseIndex **num_original_clauses_** = 0

ClauseIndex **num_original_binary_clauses_** = 0

ClauseIndex **num_original_unit_clauses_** = 0

VariableIndex **num_variables_** = 0
number of variables remaining

VariableIndex **num_used_variables_** = 0
Number of variables that actually occurs in clauses

VariableIndex **num_free_variables_** = 0
Number of variables that do not appear in any clause.

VariableIndex **num_indep_support_variables_** = 0
Number of variables in the independent support.

ClauseIndex **num_long_clauses_** = 0
Number of long clauses (i.e., greater than 3 unique literals) after pre-processing.

ClauseIndex **num_binary_clauses_** = 0
Number of binary clauses after pre-processing.

ClauseIndex **num_long_conflict_clauses_** = 0

ClauseIndex **num_binary_conflict_clauses_** = 0

ClauseIndex **times_conflict_clauses_cleaned_** = 0

ClauseIndex **num_unit_clauses_** = 0

uint64_t **num_decisions_** = 0
number of all decisions made - MT

Number of times a branch variable is set. It does not include BCP variable setting.

uint64_t **num_failed_literals_detected_** = 0
Number of all failed literal detected.

uint64_t **num_failed_literal_tests_** = 0

uint64_t **num_conflicts_** = 0
Number of all conflicts encountered.

uint64_t **num_clauses_learned_** = 0

VariableIndex **max_component_split_depth_** = 0
Maximum number of embedded components splits.

VariableIndex **max_branch_var_depth_** = 0
Maximum depth of branch variable setting. This excludes variables implied via UP or implied BCP.

TreeNodeIndex **num_top_tree_nodes_** = 0
Number of top tree nodes observed.

TreeNodeIndex **num_tree_nodes_by_type_**[static_cast<long>(**NUM_TREE_NODE_TYPES**)]
Number of nodes of each of the top tree node types

mpz_class **num_models_by_tree_node_type_**[static_cast<long>(**NUM_TREE_NODE_TYPES**)]
Number of models for each of the top tree node types.

uint64_t **num_cache_hits_** = 0

uint64_t **num_cache_look_ups_** = 0

uint64_t **sum_cache_hit_sizes_** = 0

uint64_t **num_cached_components_** = 0

uint64_t **sum_size_cached_components_** = 0

uint64_t **sum_bytes_cached_components_** = 0
Number of bytes occupied by all cached components.

uint64_t **overall_bytes_components_stored_** = 0

uint64_t **sum_bytes_pure_cached_component_data_** = 0

uint64_t **overall_bytes_pure_stored_component_data_** = 0

uint64_t **sys_overhead_sum_bytes_cached_components_** = 0

uint64_t **sys_overhead_overall_bytes_components_stored_** = 0

uint64_t **cache_infrastructure_bytes_memory_usage_** = 0

uint64_t **overall_num_cache_stores_** = 0

mpz_class **final_solution_count_** = 0

double **sampler_time_elapsed_** = DBL_MAX
Amount of time required to complete the full sampling.

double **sampler_pass_1_time_** = DBL_MAX
Amount of time required to complete phase 1 and build all partial assignments (or just the time required to collect a single sample when two pass is disabled).

double **sampler_pass_2_time_** = DBL_MAX
Time required to convert all partial assignments into complete assignments. This represents "pass 2" of the sampler.

std::vector<*VariableIndex*> **numb_second_pass_vars_**
When performing two pass testing, this records the number of variables that remain to be set at the start of the second pass for each sample.

Class DecisionStack

- Defined in *File stack.h*

Inheritance Relationships

Base Type

- `public std::vector< StackLevel >`

Class Documentation

class DecisionStack : **public** std::vector<*StackLevel*>
Vector based decision stack. It is also used during implicit BCP (i.e., failed literal testing).

Public Functions

DecisionStack()

void **startFailedLitTest**()
Sets a flag to begin implicit BCP

void **stopFailedLitTest**()
Sets a flag to halt implicit BCP.

StackLevel &**top**()
Accessor for the top of the decision stack.

Return Top of the decision stack.

const *StackLevel* &**top_const**() **const**
Accessor for the top of the decision stack.

Return Top of the decision stack.

StackLevel &**on_deck**()
Get the element below the top, making it on deck in baseball parlance (i.e., it would be up next after the top).

Return One below the top of the stack.

DecisionLevel **get_decision_level**() **const**
Height of the decision stack. This represents "d" in CDCL.

If a failed literal test is underway, a failed literal is pushed onto the stack during testing.

Return Height of the stack.

Class DifferencePackedComponent

- Defined in *File difference_packed_component.h*

Inheritance Relationships

Base Type

- `public BasePackedComponent` (*Class BasePackedComponent*)

Class Documentation

class DifferencePackedComponent : **public** *BasePackedComponent*
difference_packed_component.h

Purpose: Defines the class *DifferencePackedComponent()*.

Author Zayd Hammoudeh zayd@ucsc.edu

Version 0.00.00

This software may be modified and distributed under the terms of the MIT license. See the LICENSE file for details.

Original Author: Marc Thurley.

Public Functions

DifferencePackedComponent()

DifferencePackedComponent(*Component* &*rComp*)

unsigned **num_variables**() **const**

unsigned **data_size**() **const**

unsigned **data_only_byte_size**() **const**

unsigned **raw_data_byte_size**() **const**

unsigned **sys_overhead_raw_data_byte_size**() **const**

bool **equals**(**const** *DifferencePackedComponent* &*comp*) **const**

Template Class GenericCacheableComponent

- Defined in *File cacheable_component.h*

Inheritance Relationships

Base Type

- `public T_Component`

Class Documentation

template <**class** T_Component>
class **GenericCacheableComponent** : **public** T_Component

Public Functions

GenericCacheableComponent()

GenericCacheableComponent(*Component* &*comp*)

unsigned long **SizeInBytes**() **const**

unsigned long **sys_overhead_SizeInBytes**() **const**

void **set_father**(*CacheEntryID f*)

const *CacheEntryID* **father**() **const**

void **set_next_sibling**(*CacheEntryID sibling*)

CacheEntryID **next_sibling**()

void **set_first_descendant**(*CacheEntryID descendant*)

const *CacheEntryID* **first_descendant**() **const**

void **set_next_bucket_element**(*CacheEntryID entry*)

CacheEntryID **next_bucket_element**()

Class Instance

- Defined in *File instance.h*

Inheritance Relationships

Derived Type

- `public Solver` (*Class Solver*)

Class Documentation

class Instance
Subclassed by *Solver*

Protected Functions

void **unSet**(*LiteralID lit*)

Antecedent &**getAntecedent**(*LiteralID lit*)

const *Antecedent* &**antecedent**(*LiteralID lit*) **const**
Accessor the antecedent object. It is a constant alternative to the method getAntecedent(LiteralID).

Return *Variable* corresponding to the literal.

Parameters

- `lit`: *Literal* identification number

bool **hasAntecedent**(*LiteralID lit*) **const**
Checks if the literal has an antecedent clause.

Return True if the literal has an antecedent.

Parameters

- `lit`: *Literal* identification object

bool **isAntecedentOf** (*ClauseOfs ante_cl*, *LiteralID lit*)

bool **isolated** (*VariableIndex v*)

bool **free** (*VariableIndex v*)

bool **deleteConflictClauses** (bool *delete_all* = false)

bool **markClauseDeleted** (*ClauseOfs cl_ofs*)

void **compactConflictLiteralPool** ()

void **compactClauses** ()

void **compactVariables** ()

void **cleanClause** (*ClauseOfs cl_ofs*)
instance.cpp

Purpose: Defines the methods for the *Instance()* class.

Author Zayd Hammoudeh zayd@ucsc.edu

Version 0.00.00

Original Author: Marc Thurley.

unsigned long **num_conflict_clauses** () **const**

const *VariableIndex* **num_variables** () **const**
Number of variables in the formula.

Return Size of the variables array

bool **createfromFile** (**const** std::string *&file_name*)
Reads the CNF formula from a file. It does minimal analysis of the clauses. It adds the clauses to the basic clause data structures including the occurence lists.

If there is a self contradictory clause, it may catch that failure but it is not guaranted.

Return "true" if the file was parsed and the formula properly built.

Parameters

- `file_name`: Location of the CNF formula in a file

void **decayActivities** ()
Decay the activity scores of the literal based on a regular decision frequency (128 decisions in the original code).

void **updateActivities** (*ClauseOfs clause_ofs*)

bool **isUnitClause** (**const** *LiteralID lit*)
Checks whether the specified literal is unit clause.

Return True if the literal is a unit clause.

Parameters

- `lit`: *Literal* identification object.

bool **existsUnitClauseOf**(*VariableIndex v*)
Checks whether a unit clause exists for the specified variable.

Return true if the specified variable is in a unit clause.

Parameters

- `v`: *Variable* identification number

void **ParseCnfCommentForSupport**(std::ifstream &*input_file*)
Parses a comment line in a DIMACs file to extract the independent support if it is present.

Parameters

- `input_file`: Input file stream of a DIMACs file.

bool **addUnitClause**(**const** *LiteralID lit*)
Attempts to add a unit clause literal to the set of unit clauses.

If the unit clause (or its complement) already exists, the function merely returns and does nothing.

Return True if the literal was added or already exists in the unit clause list A return of false implies the formula is UNSAT.

Parameters

- `lit`: Unit clause literal

long **addClause**(std::vector<*LiteralID*> &*literals*)
Adds a clause (be it unit, binary, or long) to the set of clauses for the formula.

Return Identification number of the clause. If there is an error adding the clause, it returns "INT_MIN" (i.e., the minimum integer value).

Parameters

- `literals`: Set of literals in the clause

Antecedent **addUIPConflictClause**(std::vector<*LiteralID*> &*literals*)

bool **addBinaryClause**(*LiteralID litA*, *LiteralID litB*)

Variable &**var**(**const** *LiteralID lit*)
Accessor to get a variable object from a literal.

Return *Variable* object associated with the literal

Parameters

- `lit`: *Literal* identification object.

const *Variable* &**var_const**(**const** *LiteralID lit*) **const**
Const Version of the *Variable* Function.

Return *Variable* object associated with the literal

Parameters

- `lit`: *Literal* identification object.

Literal &**literal**(*LiteralID lit*)
Accessor to extract the specified literal.

Return *Literal* with the specified ID

Parameters

- `lit`: Identification number for a positive or negative literal.

bool **isSatisfied**(**const** *LiteralID* &*lit*) **const**

bool **isResolved**(*LiteralID lit*)

bool **isActive**(*LiteralID lit*) **const**

Return true if the variable is unset.

Parameters

- `lit`: *Literal* ID number

std::vector<*LiteralID*>::const_iterator **beginOf**(*ClauseOfs cl_ofs*) **const**

std::vector<*LiteralID*>::iterator **beginOf**(*ClauseOfs cl_ofs*)

decltype(*literal_pool_*.begin()) **conflict_clauses_begin**()

ClauseHeader &**getHeaderOf**(*ClauseOfs cl_ofs*)

bool **isSatisfied**(*ClauseOfs cl_ofs*)

Protected Attributes

DataAndStatistics **statistics_**

std::vector<*LiteralID*> **literal_pool_**
literal_pool_: the literals of all clauses are stored here INVARIANT: first and last entries of literal_pool_ are a SENTINEL_LIT

Clauses begin with a *ClauseHeader* structure followed by the literals terminated by SENTINEL_LIT

unsigned long **original_lit_pool_size_**
this is to determine the starting offset of conflict clauses - MT

This contains the number of unique literals that appear in the original formula. It will be less than or equal to 2 * |variables_|

LiteralIndexedVector<*Literal*> **literals_**

LiteralIndexedVector<std::vector<*ClauseOfs*>> **occurrence_lists_**
Set of long clauses that the specified literal appears in.

std::vector<*ClauseOfs*> **conflict_clauses_**

std::vector<*LiteralID*> **unit_clauses_**
Set of all unit clauses (i.e., list of unit literals)

std::vector<*Variable*> **variables_**

LiteralIndexedVector<*TriValue*> **literal_values_**

std::vector<*VariableIndex*> **unused_vars_**
Stores the variable identification numbers of any variables that do not appear in a formula at all.

bool **has_independent_support_** = false
Stores whether the solver has an independent support. This is based on the contents of the CNF file.

std::vector<*VariableIndex*> **independent_support**
Variables that represent the independent support.

Class Literal

- Defined in *File structures.h*

Class Documentation

class Literal
Reference class storing information on each literal (i.e., both positive and negative0. The primary information this tracks is:

- binary_links_ - Represents variables linked by binary clauses
- Watch List - Clauses associated with the two watch literals principle

Public Functions

void **increaseActivity**(unsigned *u* = 1)
Increase the literal's supplied literals activity score by the specified amount.

Parameters

- `u`: Amount to increase the literals activity score by. This should generally be a positive number.

void **resetActivity**()
Resets the literals activity score which represents how frequently it appears in a clause.

void **removeWatchLinkTo**(*ClauseOfs clause_ofs*)

Parameters

- `clause_ofs`:

void **replaceWatchLinkTo**(*ClauseOfs clause_ofs*, *ClauseOfs replace_ofs*)

void **addWatchLinkTo**(*ClauseIndex clause_ofs*)
Associates the clause to be monitored with this literal.

Parameters

- `clause_ofs`: Clause to be monitored

void **addBinLinkTo**(*LiteralID lit*)
Creates a link between the implicit literal and the explicitly supplied literal. This implies the two literals are in the same binary clause.

Parameters

- `lit`: Other literal to be linked to.

void **resetWatchList**()
Resets the binary watch list for a literal. It removes all binary links from the literal to all binary clauses.

It does put the sentinel clause onto the stack.

bool **hasBinaryLinkTo**(*LiteralID lit*)
Checks if the implicit literal has a link to the explicitly supplied literal.

Return true if the two literals appear in the same binary clause.

Parameters

- `lit`: Other literal to check for a binary clause linkage to.

bool **hasBinaryLinks**()
Checks if the literal is associated with any binary links.

Return True if there is at least one binary link.

Public Members

std::vector<*LiteralID*> **binary_links_** = std::vector<*LiteralID*>(1, NOT_A_LIT)

std::vector<*ClauseOfs*> **watch_list_** = std::vector<ClauseOfs>(1, NOT_A_CLAUSE)
List of watch clauses for the two watch literals paradigm.

float **activity_score_** = 0.0f

Class LiteralID

- Defined in *File structures.h*

Class Documentation

class LiteralID

Public Functions

LiteralID()
Initializes the literal to negated and ID 0.

LiteralID(int *lit*)
Converts a literal number into a literal ID. The absolute value of the passed literal ID is the variable number. The sign represents whether the literal is negated or non-negated.

Parameters

- `lit`: *Literal* ID with a negative number representing a negated literal

LiteralID(*VariableIndex var*, bool *sign*)
Build a literal from the variable ID and a sign term.

Parameters

- `var`: *Variable* identification number.
- `sign`: true for a positive literal and false for a negative literal

VariableIndex **var**() **const**
Gets the variable identification number associated with the literal. It ignores the literal's sign (i.e., negation/positive)

Return *Variable* ID number

int **toInt**() **const**
Converts the literal into the form (sign)VariableNumber where sign is +/- depending whether the literal is negated.

Return Signed version of the variable number.

void **inc**()
Increments the literal ID number. This may cause a negated literal to become positive or cause a positive literal's ID to become the next negated literal.

void **copyRaw**(unsigned int *v*)
Sets the literal ID to the processed value. It does not process its integrity or correctness.

Parameters

- `v`: New literal value

const bool **sign**() **const**
Returns whether the literal is positive or negated.

Return True if the literal is positive (i.e., unnegated)

const bool **operator!=**(**const** *LiteralID* &*rL2*) **const**

const bool **operator==**(**const** *LiteralID* &*rL2*) **const**

const bool **operator<**(**const** *LiteralID* &*rL2*) **const**

const *LiteralID* **neg**() **const**
Creates a literal that has the opposite sign as the current literal. For example, if the implicit literal is negated, this returns a positive/unnegated literal. If it is positive, this function returns a negated literal.

Return *LiteralID* of this literal just negated.

void **print**() **const**

unsigned **raw**() **const**
Accesses the raw value of how the literal is stored in the data structure.

Return Unmodified verison of the data structure value

Template Class LiteralIndexedVector

- Defined in *File containers.h*

Inheritance Relationships

Base Type

- `protected std::vector< _T >`

Class Documentation

template <class *_T*>
class LiteralIndexedVector : **protected** std::vector<_T>

containers.h

Purpose: Defines the LiteralIndexVector() class. This stores items and access them via literal identification numbers.

Author Zayd Hammoudeh zayd@ucsc.edu

Version 0.00.00

Original Author: Marc Thurley.

Public Functions

LiteralIndexedVector(*VariableIndex size* = 0)

Creates a vector twice the specified size.

Parameters

- `size`: Expected number of variables

LiteralIndexedVector(*VariableIndex size*, **const typename** std::vector<_T>::value_type &*__value*)

Create a vector twice the specified size initialized to the specified value.

Parameters

- `size`: Expected number of variables.
- `__value`: Value to write into all locations in the vector

_T &**operator[]**(**const** *LiteralID lit*)

const _T &**operator[]**(**const** *LiteralID* &*lit*) **const**

std::vector<_T>::iterator **begin**()

Creates an iterator that points to the first element in the vector.

Return Iterator to first element

void **resize**(*VariableIndex* *_size*)

Parameters

- `_size`: Expected number of variables.

void **resize**(*VariableIndex* *_size*, **const typename** std::vector<_T>::value_type &*_value*)
Resizes the vector to twice the specified size. If a value is specified, it sets all the elements to the specified value.

Parameters

- `_size`:
- `_value`:

void **reserve**(*VariableIndex* *_size*)

LiteralID **end_lit**()
Calculated version of the last literal ID number. It is based off the size of the literal array and may have unexpected behavior if the numbers are not incrementally increasing.

Return Negated version of the largest possible literal

Class Random

- Defined in *File rand_distributions.h*

Nested Relationships

Nested Types

- *Class Random::Mpz*

Class Documentation

class Random
rand_distributions.h

Purpose: Static class for getting randomly created objects from different distributions and of different types.

Author Zayd Hammoudeh zayd@ucsc.edu

Version 0.00.00

Unnamed Group

template <**typename** T>
static T **uniform**(T *min*, T *max*)
Generates a uniform random integer in the range [min, max].

Return Uniform random integer across the full range of possible values.

Parameters

- `min`: Minimum value that can be generated inclusive.
- `max`: Maximum value that can be generated inclusively.

static int **uniform**(int *min* = INT_MIN, int *max* = INT_MAX)
Special int uniform random generator that can optionally avoid reinitialize the uniform int distribution generator.

Public Static Functions

static void **init**(*SolverConfiguration* **config*)
Random State Initializer

Creates the random state of the program. It also seeds the random number generator.

Parameters

- `config`: *Solver* configuration

static *SampleSize* **binom**(*SampleSize* *n*, double *p*)
Creates a random variable sampled from the distribution Binom(n,p)

Return Integer random variable generated according to the binomial distribution.

Parameters

- `n`: Number of Bernoulli trials in the binomial distribution
- `p`: Probability of success.

template <**typename** ListType, **typename** CountType>
static void **DownsampleList**(CountType *target_size*, std::vector<ListType> &*oversampled_vec*, bool *resize* = true)
Using a modified version of the Fisher-Yates shuffle, the first `target_size` elements of the vector `oversampled_vec` will be selected uniformly at random. All modifications are done inplace. The running time is O(`target_size`).

Template Parameters

- `ListType`: Type of the object in the oversampled vector.
- `CountType`: Type of object used to store the counts.

Parameters

- `oversampled_vec`: Vector with more elements than desired. `target_size` elements will be selected in the first [0, `target_size`) elements.
- `target_size`: Number of objects to select in `oversampled_vec`

- `resize`: True to resize `oversampled_vec` to the size `target_count`.

template <typename T>
static void **shuffle**(std::vector<T> &*vec*)
Shuffles the specified vector in place uniformly at random.

Template Parameters

- `T`: Vector element type

Parameters

- `vec`: Vector to be shuffled.

static void **SelectRangeInts**(*SampleSize* *max_id*, *SampleSize* *num_elements*, std::vector<*SampleSize*> &*samples_to_replace*)
Builds a randomly selected list of integers in the range [0, `max_id`).

This is performed WITHOUT replacement. Its running time is Theta(`max_id`)

Parameters

- `max_id`: Maximum value (exclusive) that any integer in the range can be.
- `num_elements`: Number of elements to be selected
- `samples_to_replace`: List of p num_elements randomly selected integers in the range [0, `max_id`).

template <typename T>
static void **shuffle**(**typename** std::vector<T>::iterator *begin*, **typename** std::vector<T>::iterator *end*)
Shuffles a vector using the standad random device.

Template Parameters

- `T`: Type contained in the vector

Parameters

- `begin`: Beginning of the vector to shuffle (inclusive)
- `end`: End of the vector to shuffle (exclusive)

class Mpz

Public Static Functions

static void **uniform**(mpz_class *max_z*, mpz_class &*rand_val*)
Generates and returns a random multiprecision integer. This may be used by other classes to consolidate the random MP generation.

Parameters

- `max_z`: Maximum integer value. All generated random values will be in the range [0, max_z).
- `rand_val`: Output value where the generated random number will be stored.

static *SampleSize* **binom**(*SampleSize n*, **const** mpz_class &*t*, **const** mpz_class &*a*)
Selects a binomially distributed random variable from Binom(n, a/t).

It may an approximate or exact version of the distribution depending on the state of the variable Random::Mpz::use_approx_binom_.

Return Integer random variable generated according to the binomial distribution.
Parameters
- `n`: Number of Bernoulli trials in the binomial distribution
- `t`: Total weight of all samples
- `a`: Weight of success.

Class Random::Mpz

- Defined in *File rand_distributions.h*

Nested Relationships

This class is a nested type of *Class Random*.

Class Documentation

class Mpz

Public Static Functions

static void **uniform**(mpz_class *max_z*, mpz_class &*rand_val*)
Generates and returns a random multiprecision integer. This may be used by other classes to consolidate the random MP generation.

Parameters

- `max_z`: Maximum integer value. All generated random values will be in the range [0, max_z).
- `rand_val`: Output value where the generated random number will be stored.

static *SampleSize* **binom**(*SampleSize n*, **const** mpz_class &*t*, **const** mpz_class &*a*)
Selects a binomially distributed random variable from Binom(n, a/t).

It may an approximate or exact version of the distribution depending on the state of the variable Random::Mpz::use_approx_binom_.

Return Integer random variable generated according to the binomial distribution.

Parameters

- `n`: Number of Bernoulli trials in the binomial distribution
- `t`: Total weight of all samples
- `a`: Weight of success.

Class SampleAssignment

- Defined in *File model_sampler.h*

Class Documentation

class SampleAssignment

Public Functions

SampleAssignment(*SampleSize sample_count*)
Creates a blank assignment with all variables unset.

Parameters

- `sample_count`: Number of samples this sample will represent.

SampleAssignment()
Empty constructor with zero samples. It is not valid for general use.

std::string **ToString**() **const**
Converts the sample to a string,

Return String representation of the sample.

const *AssignmentEncoding* **var_assignment**(**const** *VariableIndex* &*var*) **const**
Variable Assignment Accessor.

Extracts the value of a single variable from the partial assignment

Return Value of the specified value number

Parameters

- `var`: Identification number of the variable from 1 to num_var

void **GetPartialAssignment**(*PartialAssignment* &*all_vars*) **const**
Extracts the current assignment in vector form.

Return Assignment as a vector

void **setVarAssignment**(**const** *VariableIndex var*, **const** *AssignmentEncoding* &*val*)
Variable Value Setter

Modifies the bit values for a specific variable.

Parameters

- `var`: *Variable* identification number between 1 and num_var
- `val`: Value to encode the variable as.
- `lit`: *Literal* ID

const bool **IsComplete**() **const**
Complete Assignment Checker

Determines whether the sample model is partial or complete.

Return true if the sample model is a complete assignment.

const *SampleSize* **sample_count**() **const**
Complete Assignment Checker

Determines whether the sample model is partial or complete.

Return true if the sample model is a complete assignment. Accessor for the number of cached component IDs in this object.

Return Number of cached component IDs in this object Accessor for the number of samples in this object.

Return Number of samples represented by the object.

const *VariableIndex* **num_set_vars**()
Accesor for the number of variables set in this assignment. Note that this does not include any emancipated variables since those are free and not set.

Return Number of variables set in this assignment.

const *VariableIndex* **num_set_vars_const**() **const**
Accessor for the number of variables set in this assignment. Note that this does not include any emancipated variables since those are free and not set.

Return Number of variables set in this assignment.

const *VariableIndex* **num_unset_vars**()
Updates the assignment of the implicit assignment with that of the specified one. It does NOT update the sample count

Return Number of unset unemancipated variables.

Parameters

- `other`: Another *SampleAssignment*. Gets the number of remaining variables that are unset. This does NOT include emancipated variables.

const bool **IsVarEmancipated**(*VariableIndex var*) **const**
Builds and returns the set of unconstrained variables in this sample.

Return Identification number of the unset variables. Checks if the specified variable is emancipated.

Return True if the variable is emancipated.

Parameters

- `var`: *Variable* identification number

const std::vector<*CacheEntryID*> &**cache_comp_ids**() **const**
Cached component ID accessor.

Return All cached components identification numbers in this object.

void **clear_cache_comp_ids**()
Clears all stored cached component IDs. There is generally not a reason to call this function outside of when filling a partial assignment. Care should be taken when using this function.

Public Static Functions

static void **set_num_var**(**const** *VariableIndex* *num_var*)
Defines the number of variables in the Boolean formula. This is used to encode the recipe.

The number of variables dictates the size of the

Parameters

- `num_var`: Number of variables in the equation.

Class SamplesManager

- Defined in *File model_sampler.h*

Class Documentation

class SamplesManager

Public Functions

SamplesManager(*SampleSize* *num_samples*, *SolverConfiguration* &*config*)
Initialize the sample manager. It creates complete blank samples.

void **exportFinal**(std::ostream &*out*, **const** *DataAndStatistics* &*statistics*, **const** *SolverConfiguration* &*config*)
Copy constructor. Equality operator.

Exports the set of samples to an output stream. This can be either a file or to the console.

Return Reference to the new *SamplesManager* created. This allows for chaining equality operators. Sample Exporter

Parameters

- `other`: Object to which the implicit object will be set.

Parameters

- `output_file_path`: Path for export file
- `statistics`: CNF and runtime statistics structure.
- `config`: *Solver* configuration

void **reservoirSample**(**const** *Component* **active_comp*, **const** std::vector<*LiteralID*> &*literal_stack*, **const** mpz_class &*solution_weight*, **const** mpz_class &*weight_multiplier*, **const** *AltComponentAnalyzer* &*ana*, *VariableIndex literal_stack_ofs*, **const** std::vector<*VariableIndex*> &*freed_vars*, **const** std::vector<*CacheEntryID*> &*cached_comp_ids*, **const** *CachedAssignment* &*cached_assn*, *SampleAssignment* &*cached_sample*)

Performs reservoir sampling

Parameters

- `active_comp`: Currently active component.
- `literal_stack`: Current assigned literal stack
- `sample_weight`: Weight of the current set of samples to assign.

void **GenerateSamplesToReplace**(**const** mpz_class &*new_sample_weight*, std::vector<*SampleSize*> &*samples_to_replace*) **const**

Sample Replacement List Generator

Builds a list of the samples that will be replaced from the assignment collection. The selection of whether a sample will be replaced is based off their relative weighting of their sample counts.

Parameters

- `new_sample_weight`: Model count weight for the new samples.
- `samples_to_replace`: Contains the indices of the samples in the list that will be replaced.

void **stitch**(*SamplesManager* &*other*)

Sample *Variable* Assignment Accessor

Gets the value of the variable assignment for a specific sample in the list of samples

Debug function.

Stitches together to sample managers. This is used when combining the results after a component split.

Return *Variable*'s assigned value. Samples Manager Stitcher

Parameters

- `sample_num`: Sample number between 0 (inclusive) and num_samples (exclusive)
- `var`: *Variable* number

Parameters

- `other`: *Component* split to be merged with.

void **StitchShuffledArray**(*SamplesManager* &*other*)

Perform stitching permutation generation by creating a vector of size tot_num_samples_ (|S|) and shuffling it via a Fisher-Yates shuffle which has running time Theta(|S|).

Parameters

- `other`: *SamplesManager* that will be stitched to the implicit *SamplesManager*.

void **SplitAndStitch**(*ListOfSamples*::iterator &*this_itr*, *ListOfSamples*::iterator &*other_itr*, *SampleSize* &*num_new_samples*)

Splits the sample point to by `this_itr`. The new split sample has `num_new_samples`. This new sample is then stitched with the sample pointed to by `other_itr`. After performing the stitching, the sample count of `other_itr` is decreased by `num_new_samples`.

Note: that if `num_new_samples` and the sample count of `this_itr` are equal, the sample pointed to by this_itr is not split. All other steps proceed normally.

Parameters

- `this_itr`: Sample to be split and then merged.
- `other_itr`: Object that will be stitched with the new split sample.
- `num_new_samples`: Sample size for the new stitched object. It must be greater than zero and less than or equal to the sample count of `this_itr`.

const bool **verifyPostStitchingCorrectness**(*SamplesManager* &*other*) **const**

Debug helper function to verify that post stitching, the number of samples is correct.

Return True if the stitching appears correct.

Parameters

- `other`: Set of samples that there were stitched to the implicit sample set.

void **merge**(*SamplesManager* &*other*, **const** mpz_class &*other_multiplier*, **const** std::vector<*VariableIndex*> &*freed_vars*, **const** std::vector<*CacheEntryID*> &*cached_comp_ids*, **const** *CachedAssignment* &*cached_assn*, *SampleAssignment* &*cached_sample*)

Samples Manager Merger

After a component split, a new descendant *SamplesManager* is created. This function will merge the existing *SamplesManager* with the original one before the component split.

See other model count.

Parameters

- `other`: Sample manager created for the descendants of the component split.
- `other_multiplier`: Used to scale the model count of the

Parameters

- `freed_vars`: List of emanicipated variables that can be set to either true or false.
- `cached_assn`: Assignment stored in the cached that is used when storing samples in cache.
- `cached_sample`: Sample output when performing sample caching. This will be used to update the top of the decision stack.

bool **VerifySolutions**(**const** std::string &*input_file_path*, bool *skip_unassigned* = false) **const**

Sample Verifier

Verifies that all samples generated by the program are valid and actually satisfies the input file. This is mostly for debug and should NEVER return false. If false is ever returned, something is wrong.

Return true if all samples are valid.

Parameters

- `input_file_path`: Path to a file in CNF formula
- `skip_unassigned`: If true, skip verification of all clauses with an unassigned literal.

const mpz_class &**model_count**() **const**
Model Count Extractor

Gets the current number of solutions tracked by the manager object

Return Model count for the samples manager object

void **AddEmancipatedVars**(**const** std::vector<*VariableIndex*> &*emancipated_vars*)
Emancipated and Unused *Variable* Adder

Unused variables are any variables that do not appear at all in a given formula. This sets them at the end of the countSAT execution.

Parameters

- `emancipated_vars`: List of the IDs of unused variables.

void **AddCachedCompIds**(**const** std::vector<*CacheEntryID*> &*cached_comp_ids*)
Adds the specified cached component identification numbers to all samples in the collection.

Parameters

- `cached_comp_ids`: Cached component identification numbers for all samples.

void **TransferVariableAssignments**(*ListOfSamples* &*others*)
Transfers the variable assignments from a previous sampler run and uses them to update the implicit sample manager. This is done when filling a partial assignment since multiple samples with the same cached component are run together even if the rest of their partial assignment is different.

Parameters

- `others`: Existing samples from a previous solver run.

ListOfSamples &**samples**()
Solver Configuration Storer

This function is used to store a reference to the solver's configuration. This is useful in case any of the run parameters are used.

Return Reference to the sample managers recipe assignments.

Parameters

- `config`: *Solver*'s configuration. Accessor to a sample manager's variable assignment.

bool **IsComplete**() **const**
Sample Setter

This function replaces the sample (based off the specified number) with the new model passed to the function.

- * Extracts a reference to the specified sample from the manager.
- *

Parameters

- `sample_num`: Number of the sample to access - base 0

- *

Return Sample at the specified number Checks whether all the samples in the set are complete.

Return true if all contained samples are complete.

Parameters

- `sample_num`: Sample number to be set
- `new_model`: New sample model to be stored Sample Accessor

void **append**(*SamplesManager* &*other*)

Adds the samples from one *SamplesManager()* to that of annother. This is different from merging and stitching as the implicit *SamplesManager()*'s samples remain unchanged. The only difference is the samples from `other` are appended to the collections of samples in the implicit parameter.

Parameters

- `other`: Another SampleManager() object whose samples will be copied to the implicit parameter.

void **append**(*ListOfSamples* &*other*)

Adds the samples from one *SamplesManager()* to that of annother. This is different from merging and stitching as the implicit *SamplesManager()*'s samples remain unchanged. The only difference is the samples from `other` are appended to the collections of samples in the implicit parameter.

Parameters

- `other`: A list of samples to add.

const *SampleSize* **num_samples**() **const**

Accessor for the number of samples actually stored by the *SamplesManager()* object.

Return Number of samples actually stored by the *SamplesManager()*.

void **RemoveSamples**(std::vector<*SampleSize*> &*samples_to_remove*)

Removes the samples specified at the indices in the `samples_to_remove` input.

Parameters

- `samples_to_remove`: Identification numbers of the samples to remove.

void **KeepSamples**(std::vector<*SampleSize*> &*samples_to_keep*)

Specifies which samples from this SampleManager should be kept. All others are discarded. This is used as a compliment to the *RemoveSamples()* function since when there is a merge, one set keeps samples and the other discards them.

Parameters

- `samples_to_keep`: Identification number of the samples that will be kept.

const bool **verifySampleCount**() **const**

Verifies that the SampleManager actually has the number of samples it is supposed to.

Return True if the sample count is correct.

const *SampleSize* **GetActualSampleCount**() **const**
Gets the actual number of samples that are contained in this manager. In MOST (but not all) cases, this should equal the expected number of samples. However, that is not necessarily the case in some rare but useful exceptions.

Return Actual number of samples contained in this object.

void **clear**()
Eliminate all existing samples.

Class Solver

- Defined in *File solver.h*

Inheritance Relationships

Base Type

- `public Instance` (*Class Instance*)

Class Documentation

class Solver : **public** *Instance*
Main object managing the execution of both the sampler and sharpSAT.

Public Functions

Solver(int *argc*, char **argv*[])
Constructs a *Solver* object from command line input arguments.

Parameters

- `argc`: Number of arguments in the argv array
- `argv`: Set of input arguments

Solver(*SolverConfiguration* &*config*, *DataAndStatistics* &*statistics*, *SampleSize num_samples* = 0, bool *two_pass* = true)

Solver(**const** *Solver* &*other*)
Copy constructor

Parameters

- `other`: Another solver used as the basic of this solver.

void **solve**(**const** *PartialAssignment* &*partial_assn* = *PartialAssignment*())
Performs the main solver execution.

Additional features include printing timing and other notes during execution. Also writes the final results to an output

This solve method can override the stored input file with a custom specified one.

Parameters

- `partial_assn`: Initial partial assignment.
- `results_output_file`: Output file to write the results

void **sample_models**()
Selects uniformly at random a set of satisfying models from the satisfying set for the implicit Boolean formula.

const *SolverConfiguration* &**config**()
Accessor for the *Solver*'s configuration.

Return Reference to the solver's internal configuration object.

void **DisableTopTreeSampling**()
Modify the *Solver()*'s configuration to no longer perform top tree sampling.

const *DataAndStatistics* &**statistics**()
Accessor for the *Solver()*'s statistics information.

Return *Solver()*'s statistics information.

void **PushNewSamplesManagerOnStack**()
Creates a new (i.e., empty) *SamplesManager* for the solver and pushes it onto the stack.

Class StackLevel

- Defined in *File stack.h*

Class Documentation

class StackLevel
stack.h

Purpose: Defines the *StackLevel()* class that encapsulates a single level in the decision stack. The decision stack itself is stored in the *DecisionStack()* class.

Author Zayd Hammoudeh zayd@ucsc.edu

Version 0.00.00

Original Author: Marc Thurley. Represents a single level in the decision stack.

Public Functions

bool **hasUnprocessedComponents**() **const**

void **nextUnprocessedComponent**()
Mark the current unprocessed component as processed.

It does verify that at least one unprocessed component exists at this decision level.

This function does not return anything. Rather, the other code operates directly on the component stack and accesses the top of it via the

See *StackLevel*:top() method.

void **resetRemainingComps**()
Mark all components in this decision level as processed.

const *VariableIndex* **super_component**() **const**

const *VariableIndex* **remaining_components_ofs**() **const**
MT - the start offset in the component stack for the remaining components in this decision level all remaining components can hence be found in

[remaining_components_ofs_, "nextLevel".remaining_components_begin_)

Return Level in the component stack of the first component for this decision level.

const *VariableIndex* **unprocessed_components_end**() **const**
One greater than the location in the component stack of unprocess components for the current decision level.

Return One greater than the location in the component stack for unprocessed components for teh current point in the decision stack.

void **set_unprocessed_components_end**(*VariableIndex end*)
Update the location of the unprocessed component reference for this decision level.

Parameters

- `end`: New location of the set of unprocessed components for this decision level.

StackLevel()

StackLevel(*VariableIndex super_comp*, *VariableIndex lit_stack_ofs*, *ClauseOfs comp_stack_ofs*)

VariableIndex **currentRemainingComponent**()
Access for the total number of remaining components across all decision levels not just the current one.

Return Total number of remaining components

bool **isSecondBranch**()
Checks if the second (i.e., "true") branch is now active.

Return true if the second ("true") branch is active.

void **changeBranch**()
Set the active branch (i.e., the one being searched to the positive branch.

bool **anotherCompProcessible**()
Checks if there is another processible component left. Lack of processible components could be due to either the branch being unsatisfiable or there being no processible components left

Return true if a processible component remains.

VariableIndex **literal_stack_ofs**() **const**

void **includeSolution**(**const** mpz_class &*solutions*)
Updates the solution count. If no solution have been found, then the increase is additive. If at least one solution has already been found then the increase is multiplicative.

Parameters

- `solutions`: Number of solutions to increase by.

void **includeSolutionSampleMultiplier**(**const** mpz_class &*solutions*)
Updates the multiplier for when doing solution sampling.

Parameters

- `solutions`: Number of solutions to increase by.

void **includeSolution**(unsigned *solutions*)
Updates the solution count. If no solution have been found, then the increase is additive. If at least one solution has already been found then the increase is multiplicative.

Parameters

- `solutions`: Number of solutions to increase by.

void **configureNewLevel**(**const** *StackLevel* &*top*, **const** *DecisionLevel decision_level*)
When variables become free in a residual formula (i.e., they no longer appear in any clauses), the solution count multiplier is stored. This needs to be pushed down the stack to ensure that the sample weights are correct.

This function pushes the current scalar multiplier down the stack.

Parameters

- `top`: Current top of the decision stack.
- `decision_level`: Number of branch variables already assigned. This does not include any implicit partial assignment.

void **addFreeVariables**(**const** std::vector<*VariableIndex*> &*freed_vars*)
Freed Variables Adder

When performing variable assignments, it is common that variables will become free (i.e., no longer appear in any clauses). Those variables will not appear in the literal stack since they are not set. However, they must be assigned as part of the sample generator. This function stores those freed variables.

Parameters

- `freed_vars`: Newly freed variables

void **addCachedCompIds**(**const** std::vector<*CacheEntryID*> &*ids*)
Store the component IDs for the cached components.

Parameters

- `ids`: List of cached entry identification numbers

bool **branch_found_unsat**() **const**
Checks if the currently active branch variable has been determined to be unsatisfiable.

Return true if the currently active branch is unsatisfiable.

void **mark_branch_unsat**()
Sets the currently active branch to being unsatisfiable.

const mpz_class **getTotalModelCount**() **const**
Total number (i.e., sum of false and true) branches of this literal decision in the subtree.

Return Total model count for both the positive and negative subtrees.

const mpz_class &**getActiveModelCount**() **const**
Gets the model count for the active branch. If the active branch is false, it will return the model count for negated version of the literal. Otherwise, it return the positive (unnegated) model count.

Return Model count for the currently active literal state.

const mpz_class &**getSamplerSolutionMultiplier**() **const**
Gets the multiplier count for when running the solution sampler. It is used for the case when variables become free due to the residual formula no longer containing them.

Return Solution count multiplier for the sampler.

const bool **HasNoDescendentModels**()
Calculates the total descendent model count for this stack level. It is equal to:

FalseModelCount * FalseCountMultiplier + TrueModelCount * TrueCountMultiplier

Return Total descendent model count. Checks whether the associated decision level has any descendent models.

Return True if both the true and false descendents are UNSAT.

const std::vector<*VariableIndex*> &**emancipated_vars**() **const**
Emancipated *Variable* List Accessor

This function is used to access the freed variable list for a specific variable branch. This is used for variable assignments.

Return List of freed variables for the specified active branch.

const std::vector<*CacheEntryID*> &**cached_comp_ids**() **const**
Cached *Component* ID List Accessor

This function accesses the cached entry identification numbers for this stack level.

Return Cached component IDs

const bool **isComponentSplit**() **const**
Component Split State Accessor

Accessor for whether this stack level corresponds with a component split.

It only applies when performing random sampling. Otherwise, it merely returns false.

Return True if component split and random sampling is running.

void **setAsComponentSplit**()
Marks the decision level as a component split.

It only applies when performing random sampling.

void **unsetAsComponentSplit**()
Unmarks the decision level as a component split.

This function only has an effect when performing random sampling.

const bool **isFirstComponent**() **const**
Returns whether the first component in a component split has been processed. This is only relevant when performing random sampling.

Return true if the first component has not been processed

void **markFirstComponentComplete**()
This is used for marking when the first component has been completed. We use this for sample stitching.

void **set_cached_assn**(*CachedAssignment* &*cached_assn*)
Cached Assignment Updater

Updates the cached assignment associated with this stack level.

Parameters

- `cached_assn`: Updates the cached assignment information for this *StackLevel*.

const *CachedAssignment* &**cached_assn**() **const**

void **ClearCachedAssn**()
Removes all information associated with the cached assignment including the number of components it contains as well as the assignment itself.

void **set_cache_sample**(**const** *SampleAssignment* &*cache_sample*)
Store the cache sample assignment. It will be automatically associated with the right active branch.

Parameters

- `cache_sample`: Sample that will be used for caching this sample.

SampleAssignment **random_cache_sample**() **const**
Randomly select the cache sample between the two branches.

Return A sample that can be stored in the cache for this stack level.

Public Static Functions

static *VariableIndex* **componentSplitDepth**()
Component Split Depth Accessor

Accesses the global component split depth information.

Return Current component split depth.

static void **set_solver_config_and_statistics**(*SolverConfiguration* &*config*, *DataAndStatistics* &*statistics*)
Configuration Updater

Stores the solver configuration. This is used so that global configuration settings (e.g., verbose) can be used for printing.

Parameters

- `config`: *Solver* configuration

Class StopWatch

- Defined in *File solver.h*

Class Documentation

class StopWatch
solver.h

Purpose: Primary function is to define the *Solver()* class which is what runs for the entire program. It also

Copyright (C) 2018 Zayd Hammoudeh. All rights reserved.

Author Zayd Hammoudeh zayd@ucsc.edu

Version 0.00.00

This software may be modified and distributed under the terms of the MIT license. See the LICENSE file for details.

Original Author: Marc Thurley.

Public Functions

StopWatch()
Used to track time events in the execution of the solver.

Built using the internal C++ timer (i.e., timeval()). Resolution down to microseconds via the "tv_usec" property.

solver.cpp

See timeval

Purpose: Defines the methods for the *Solver()* class.

Author Zayd Hammoudeh zayd@ucsc.edu

Version 0.00.00

Original Author: Marc Thurley.

bool **timeBoundBroken**()
Checks whether the time bound has been exceeded.

Return true if the time bound has been exceeded.

bool **start**()

bool **stop**()
Records and saves the stop time independent of the time zone.

Return 0 for success, or -1 for failure (in which case errno is set appropriately).

double **getElapsedSeconds**()

bool **interval_tick**()

void **setTimeBound**(uint64_t *seconds*)
Updates the stopwatch's time bound.

Parameters

- `seconds`: New time bound.

1.3.2 Enums

Enum AssignmentEncoding

- Defined in *File primitive_types.h*

Enum Documentation

enum AssignmentEncoding
Values:

ASSN_F = 0x0

ASSN_T = 0x1

ASSN_U = 0x3

Enum PrintColor

- Defined in *File sampler_tools.h*

Enum Documentation

enum PrintColor

Values:

COLOR_BLACK = 30

COLOR_RED = 31

COLOR_GREEN = 32

COLOR_YELLOW = 33

COLOR_BLUE = 34

COLOR_MAGENTA = 35

COLOR_CYAN = 36

Enum SolverExitState

- Defined in *File primitive_types.h*

Enum Documentation

enum SolverExitState

Enumerated class the stores the result of a solver execution. It is returned by the Solver::countSAT method. It is also stored as part of the run statistics in

See *DataAndStatistics*#

Values:

NO_STATE

SUCCESS

Enum SolverNextAction

- Defined in *File primitive_types.h*

Enum Documentation

enum SolverNextAction

Associated with backtracking and whether a conflict is resolved or backtracking is required.

EXIT means the program is completed and should return success.

Values:

EXIT

RESOLVED

PROCESS_COMPONENT

BACKTRACK

Enum TopTreeNodeType

- Defined in *File primitive_types.h*

Enum Documentation

enum TopTreeNodeType
Enumerated type to represent each of the top tree node types.

Values:

MAX_DEPTH

CYLINDER

COMPONENT_SPLIT

NUM_TREE_NODE_TYPES

1.3.3 Functions

Function ExitInvalidParam

- Defined in *File sampler_tools.h*

Function Documentation

void **ExitInvalidParam**(**const** std::string &*msg*)
Exits the program due to an invalid input argument/

Parameters

- `msg`: Error message to be printed

Function ExitWithError

- Defined in *File sampler_tools.h*

Function Documentation

void **ExitWithError**(**const** std::string &*msg*, **const** int *err_code* = EXIT_FAILURE)
Exits the entire program after printing the associated message

Parameters

- `msg`: Error message to be printed

Function FileExists

- Defined in *File sampler_tools.h*

Function Documentation

const bool **FileExists**(**const** std::string &*file_path*)
Checks if the specified file exists on disk.

Return True if the file exists.

Parameters

- `file_path`: Path to the file of interest.

Function main

- Defined in *File main.cpp*

Function Documentation

Warning: doxygenfunction: Cannot find function "main" in doxygen xml output for project "SPUR" from directory: ./doxyoutput/xml

Function NOT_A_CLAUSE

- Defined in *File primitive_types.h*

Function Documentation

```
static const ClauseIndex NOT_A_CLAUSE(0)
```

Function NOT_A_LIT

- Defined in *File structures.h*

Function Documentation

```
static const LiteralID NOT_A_LIT(0, false)
```

Function PrintError()

- Defined in *File sampler_tools.h*

Function Documentation

Warning: doxygenfunction: Unable to resolve multiple matches for function "PrintError" with arguments () in doxygen xml output for project "SPUR" from directory: ./doxyoutput/xml. Potential matches:

```
- void PrintError(const std::string&)
- void PrintError(const std::stringstream&)
```

Function PrintError()

- Defined in *File sampler_tools.h*

Function Documentation

Warning: doxygenfunction: Unable to resolve multiple matches for function "PrintError" with arguments () in doxygen xml output for project "SPUR" from directory: ./doxyoutput/xml. Potential matches:

```
- void PrintError(const std::string&)
- void PrintError(const std::stringstream&)
```

Function PrintInColor()

- Defined in *File sampler_tools.h*

Function Documentation

Warning: doxygenfunction: Unable to resolve multiple matches for function "PrintInColor" with arguments () in doxygen xml output for project "SPUR" from directory: ./doxyoutput/xml. Potential matches:

```
- void PrintInColor(const std::string&, PrintColor, bool)
- void PrintInColor(const std::stringstream&, PrintColor, bool)
- void PrintInColor(std::ostream&, const std::string&, const PrintColor, bool)
- void PrintInColor(std::ostream&, const std::stringstream&, const PrintColor, bool)
```

Function PrintInColor()

- Defined in *File sampler_tools.h*

Function Documentation

Warning: doxygenfunction: Unable to resolve multiple matches for function "PrintInColor" with arguments () in doxygen xml output for project "SPUR" from directory: ./doxyoutput/xml. Potential matches:

```
- void PrintInColor(const std::string&, PrintColor, bool)
- void PrintInColor(const std::stringstream&, PrintColor, bool)
- void PrintInColor(std::ostream&, const std::string&, const PrintColor, bool)
- void PrintInColor(std::ostream&, const std::stringstream&, const PrintColor, bool)
```

Function PrintInColor()

- Defined in *File sampler_tools.h*

Function Documentation

Warning: doxygenfunction: Unable to resolve multiple matches for function "PrintInColor" with arguments () in doxygen xml output for project "SPUR" from directory: ./doxyoutput/xml. Potential matches:

```
- void PrintInColor(const std::string&, PrintColor, bool)
- void PrintInColor(const std::stringstream&, PrintColor, bool)
- void PrintInColor(std::ostream&, const std::string&, const PrintColor, bool)
- void PrintInColor(std::ostream&, const std::stringstream&, const PrintColor, bool)
```

Function PrintInColor()

- Defined in *File sampler_tools.h*

Function Documentation

Warning: doxygenfunction: Unable to resolve multiple matches for function "PrintInColor" with arguments () in doxygen xml output for project "SPUR" from directory: ./doxyoutput/xml. Potential matches:

```
- void PrintInColor(const std::string&, PrintColor, bool)
- void PrintInColor(const std::stringstream&, PrintColor, bool)
- void PrintInColor(std::ostream&, const std::string&, const PrintColor, bool)
- void PrintInColor(std::ostream&, const std::stringstream&, const PrintColor, bool)
```

Function printInputArgumentDescription

- Defined in *File main.cpp*

Function Documentation

Warning: doxygenfunction: Cannot find function "printInputArgumentDescription" in doxygen xml output for project "SPUR" from directory: ./doxyoutput/xml

Function PrintWarning

- Defined in *File sampler_tools.h*

Function Documentation

void **PrintWarning**(**const** std::string &*msg*)

Generic handler for printing warning messages.

Parameters

- `msg`: Warning message

Function UpdateVarDescedantsList(const std::vector<ComponentVarAndCls>&, VariableIndex, std::vector<bool>&, std::vector<ComponentVarAndCls>, VariableIndex)

- Defined in *File component_management.cpp*

Function Documentation

VariableIndex **UpdateVarDescedantsList**(**const** std::vector<*ComponentVarAndCls*> &*ref_vars*, *VariableIndex* *ref_num_vars*, std::vector<bool> &*varInDescendants*, std::vector<*ComponentVarAndCls*> *descendant_vars*, *VariableIndex* *desc_num_vars*)

Helper function for determining freed variables.

It goes through a list of variables and marks any that appears in the descendant list. If it is appears, then it is by definition not free.

See ref_vars (exclusive)

See ref_vars. This is a set of non-free vars.

See descendant_vars (exclusive)

Return Number of descendant elements marked as present

Parameters

- `ref_vars`: Superset list of reference variables.
- `ref_num_vars`: Number of variables in

Parameters

- `varInDescendants`: One to one mapping of variables currently or previous marked as non-free. A variable should only ever be marked as non-free (i.e., true).
- `descendant_vars`: Subset of

Parameters

- `desc_num_vars`: Number of variables in

Function UpdateVarDescedantsList(const std::vector<ComponentVarAndCls>&, VariableIndex, std::vector<bool>&, std::vector<ComponentVarAndCls>, VariableIndex)

- Defined in *File component_management.h*

Function Documentation

VariableIndex **UpdateVarDescedantsList**(**const** std::vector<*ComponentVarAndCls*> &*ref_vars*, *VariableIndex* *ref_num_vars*, std::vector<bool> &*varInDescendants*, std::vector<*ComponentVarAndCls*> *descendant_vars*, *VariableIndex* *desc_num_vars*)

Helper function for determining freed variables.

It goes through a list of variables and marks any that appears in the descendant list. If it is appears, then it is by definition not free.

See ref_vars (exclusive)

See ref_vars. This is a set of non-free vars.

See descendant_vars (exclusive)

Return Number of descendant elements marked as present

Parameters

- `ref_vars`: Superset list of reference variables.
- `ref_num_vars`: Number of variables in

Parameters

- `varInDescendants`: One to one mapping of variables currently or previous marked as non-free. A variable should only ever be marked as non-free (i.e., true).
- `descendant_vars`: Subset of

Parameters

- `desc_num_vars`: Number of variables in

1.3.4 Defines

Define BITS_PER_BYTE

- Defined in *File primitive_types.h*

Define Documentation

BITS_PER_BYTE

Define CA_CL_ALL_LITS_ACTIVE

- Defined in *File component_archetype.h*

Define Documentation

CA_CL_ALL_LITS_ACTIVE

Define CA_CL_IN_OTHER_COMP

- Defined in *File component_archetype.h*

Define Documentation

CA_CL_IN_OTHER_COMP

Define CA_CL_IN_SUP_COMP_UNSEEN

- Defined in *File component_archetype.h*

Define Documentation

CA_CL_IN_SUP_COMP_UNSEEN

Define CA_CL_MASK

- Defined in *File component_archetype.h*

Define Documentation

CA_CL_MASK

Define CA_CL_SEEN

- Defined in *File component_archetype.h*

Define Documentation

CA_CL_SEEN

Define CA_NIL

- Defined in *File component_archetype.h*

Define Documentation

CA_NIL

Define CA_VAR_IN_OTHER_COMP

- Defined in *File component_archetype.h*

Define Documentation

CA_VAR_IN_OTHER_COMP

Define CA_VAR_IN_SUP_COMP_UNSEEN

- Defined in *File component_archetype.h*

Define Documentation

CA_VAR_IN_SUP_COMP_UNSEEN

Define CA_VAR_MASK

- Defined in *File component_archetype.h*

Define Documentation

CA_VAR_MASK

Define CA_VAR_SEEN

- Defined in *File component_archetype.h*

Define Documentation

CA_VAR_SEEN

Define CACHED_VARIABLE_LEN

- Defined in *File cached_assignment.h*

Define Documentation

CACHED_VARIABLE_LEN

cached_assignment.h

Purpose: Stores a variable assignment retrieved from the cache.

Author Zayd Hammoudeh zayd@ucsc.edu

Version 0.00.00

Original Author: Marc Thurley.

Define CLAUSE_ADDING_ERROR

- Defined in *File instance.h*

Define Documentation

CLAUSE_ADDING_ERROR

instance.h

Purpose: Defines the "Instance()" class that is the superclass of the main "Solver()" class.

Author Zayd Hammoudeh zayd@ucsc.edu

Version 0.00.00

Original Author: Marc Thurley.

Define clsSENTINEL

- Defined in *File primitive_types.h*

Define Documentation

clsSENTINEL

Define END_ESCAPE

- Defined in *File sampler_tools.h*

Define Documentation

END_ESCAPE

Define ESCAPE_CHAR

- Defined in *File sampler_tools.h*

Define Documentation

ESCAPE_CHAR

sampler_tools.h

Purpose: Contains generic helper functions used by the solver.

Author Zayd Hammoudeh zayd@ucsc.edu

Version 0.00.00

Define EXIT_TIMEOUT

- Defined in *File primitive_types.h*

Define Documentation

EXIT_TIMEOUT

Define F_TRI

- Defined in *File structures.h*

Define Documentation

F_TRI

Macro value for an unsatisfied LITERAL.

Define FILE_PATH_SEPARATOR

- Defined in *File sampler_tools.h*

Define Documentation

FILE_PATH_SEPARATOR

Define FIRST_VAR

- Defined in *File primitive_types.h*

Define Documentation

FIRST_VAR

Define INVALID_DL

- Defined in *File structures.h*

Define Documentation

INVALID_DL

structures.h

Purpose: Defines a set of classes originally created by Marc Thurley include *LiteralID*, *Literal*, *Antecedent*, *Variable*, and *ClauseHeader*.

Author Zayd Hammoudeh zayd@ucsc.edu

Version 0.00.00

Original Author: Marc Thurley.

Define ITEM_SEP

- Defined in *File sampler_tools.h*

Define Documentation

ITEM_SEP

Define LT

- Defined in *File base_packed_component.h*

Define Documentation

LT(n)

Define NIL_ENTRY

- Defined in *File cacheable_component.h*

Define Documentation

NIL_ENTRY

Define PRINT_BOLD

- Defined in *File sampler_tools.h*

Define Documentation

PRINT_BOLD

Define RESET_CONSOLE_COLOR

- Defined in *File sampler_tools.h*

Define Documentation

RESET_CONSOLE_COLOR

Define SENTINEL_CL

- Defined in *File primitive_types.h*

Define Documentation

SENTINEL_CL

Define SENTINEL_LIT

- Defined in *File structures.h*

Define Documentation

SENTINEL_LIT

Define STR_DECIMAL_BASE

- Defined in *File sampler_tools.h*

Define Documentation

STR_DECIMAL_BASE

Define T_TRI

- Defined in *File structures.h*

Define Documentation

T_TRI
Macro value for a satisfied LITERAL.

Define toDEBUGOUT

- Defined in *File primitive_types.h*

Define Documentation

toDEBUGOUT(X)

Define varsSENTINEL

- Defined in *File primitive_types.h*

Define Documentation

varsSENTINEL
primitive_type.h

Purpose: Base class and type definitions used throughout this project.

Author Zayd Hammoudeh zayd@ucsc.edu

Version 0.00.00

Original Author: Marc Thurley.

Define X_TRI

- Defined in *File structures.h*

Define Documentation

X_TRI
Macro value for an unassigned/free LITERAL.

1.3.5 Typedefs

Typedef AssignmentContainer

- Defined in *File model_sampler.h*

Typedef Documentation

typedef std::vector<uint32_t> **AssignmentContainer**
model_sampler.h

Purpose: Classes for storing and managing partial and complete SAT assignments.

Copyright (C) 2018 Zayd Hammoudeh. All rights reserved.

Author Zayd Hammoudeh zayd@ucsc.edu

Version 0.00.00

This software may be modified and distributed under the terms of the MIT license. See the LICENSE file for details.

Original Author: Marc Thurley. Defines the encoding for variable states in the assignment.

Typedef CA_SearchState

- Defined in *File component_archetype.h*

Typedef Documentation

typedef unsigned char **CA_SearchState**
component_archetype.h

Purpose: Defines the *ComponentArchetype()* class.

Copyright (C) 2018 Zayd Hammoudeh. All rights reserved.

Author Zayd Hammoudeh zayd@ucsc.edu

Version 0.00.00

This software may be modified and distributed under the terms of the MIT license. See the LICENSE file for details.

Original Author: Marc Thurley.

Typedef CacheableComponent

- Defined in *File cacheable_component.h*

Typedef Documentation

typedef *GenericCacheableComponent*<*DifferencePackedComponent*> **CacheableComponent**

Typedef CacheEntryID

- Defined in *File primitive_types.h*

Typedef Documentation

typedef unsigned **CacheEntryID**
Unsigned Int.

Typedef ClauseIndex

- Defined in *File primitive_types.h*

Typedef Documentation

typedef *VariableIndex* **ClauseIndex**
Identification number for clauses.

Typedef ClauseOfs

- Defined in *File primitive_types.h*

Typedef Documentation

typedef *VariableIndex* **ClauseOfs**
Unsigned Int.

Typedef ComponentAnalyzer

- Defined in *File component_management.h*

Typedef Documentation

typedef *AltComponentAnalyzer* **ComponentAnalyzer**
component_management.h

Purpose: Defines the *ComponentManager()* class for managing the Boolean formula components.

Author Zayd Hammoudeh zayd@ucsc.edu

Version 0.00.00

Original Author: Marc Thurley.

Typedef ComponentVarAndCls

- Defined in *File primitive_types.h*

Typedef Documentation

typedef *VariableIndex* **ComponentVarAndCls**

Typedef DecisionLevel

- Defined in *File primitive_types.h*

Typedef Documentation

typedef int64_t **DecisionLevel**
May be negative due to Thurley's implementation.

Typedef ListOfSamples

- Defined in *File model_sampler.h*

Typedef Documentation

typedef std::list<*SampleAssignment*> **ListOfSamples**

Typedef PartialAssignment

- Defined in *File primitive_types.h*

Typedef Documentation

typedef std::vector<*AssignmentEncoding*> **PartialAssignment**

Typedef SampleSize

- Defined in *File primitive_types.h*

Typedef Documentation

typedef uint32_t **SampleSize**
Defines the number of samples that can be requested.

Typedef TopTreeLiteral

- Defined in *File primitive_types.h*

Typedef Documentation

typedef int32_t **TopTreeLiteral**
Type used for encoding literals in the top tree analyzer.

Typedef TreeNodeIndex

- Defined in *File primitive_types.h*

Typedef Documentation

typedef uint64_t **TreeNodeIndex**
Index used to represent top tree nodes.

Typedef TriValue

- Defined in *File structures.h*

Typedef Documentation

typedef unsigned char **TriValue**

Typedef VariableIndex

- Defined in *File primitive_types.h*

Typedef Documentation

typedef uint64_t **VariableIndex**
Identification numbers for variables.

1.3.6 Directories

Directory src

Directory path: `src`

Subdirectories

- *Directory types*

Files

- *File alt_component_analyzer.cpp*
- *File alt_component_analyzer.h*
- *File cached_assignment.h*
- *File component_cache.cpp*
- *File component_cache.h*
- *File component_cache.inc*
- *File component_management.cpp*
- *File component_management.h*
- *File containers.h*
- *File instance.cpp*
- *File instance.h*
- *File main.cpp*
- *File model_sampler.cpp*
- *File model_sampler.h*
- *File primitive_types.h*
- *File rand_distributions.cpp*
- *File rand_distributions.h*
- *File sampler_tools.h*
- *File solver.cpp*
- *File solver.h*
- *File solver_config.h*
- *File stack.cpp*
- *File stack.h*
- *File statistics.cpp*
- *File statistics.h*
- *File structures.h*

Directory types

Parent directory (`src`)

Directory path: `src/component_types`

Files

- *File base_packed_component.cpp*
- *File base_packed_component.h*

- *File cacheable_component.h*
- *File component.h*
- *File component_archetype.cpp*
- *File component_archetype.h*
- *File difference_packed_component.h*

1.3.7 Files

File alt_component_analyzer.cpp

Parent directory (`src`)

Contents

- *Definition (`src/alt_component_analyzer.cpp`)*
- *Includes*

Definition (`src/alt_component_analyzer.cpp`)

Program Listing for File alt_component_analyzer.cpp

Return to documentation for file (`src/alt_component_analyzer.cpp`)

```
#include "stack.h"
#include "alt_component_analyzer.h"

void AltComponentAnalyzer::initialize(LiteralIndexedVector<Literal> & literals,
                                      std::vector<LiteralID> &lit_pool) {
  max_variable_id_ = literals.end_lit().var() - 1;

  search_stack_.reserve(max_variable_id_ + 1);
  var_frequency_scores_.resize(max_variable_id_ + 1, 0);
  variable_occurrence_lists_pool_.clear();
  variable_link_list_offsets_.clear();
  variable_link_list_offsets_.resize(max_variable_id_ + 1, 0);

  std::vector<std::vector<ClauseOfs> > occs(max_variable_id_ + 1);
  std::vector<std::vector<unsigned> > occ_long_clauses(max_variable_id_ + 1);
  std::vector<std::vector<unsigned> > occ_ternary_clauses(max_variable_id_ + 1);

  std::vector<unsigned> tmp;
  max_clause_id_ = 0;
  unsigned curr_clause_length = 0;
  auto it_curr_cl_st = lit_pool.begin();

  for (auto it_lit = lit_pool.begin(); it_lit < lit_pool.end(); it_lit++) {
    if (*it_lit == SENTINEL_LIT) {
      if (it_lit + 1 == lit_pool.end())
        break;
```

(continues on next page)

(continued from previous page)

```
      max_clause_id_++;
      it_lit += ClauseHeader::overheadInLits();
      it_curr_cl_st = it_lit + 1;
      curr_clause_length = 0;
    } else {
      assert(it_lit->var() <= max_variable_id_);
      curr_clause_length++;

      getClause(tmp, it_curr_cl_st, *it_lit);

      assert(tmp.size() > 1);

      if (tmp.size() == 2) {
//      if (false) {
        occ_ternary_clauses[it_lit->var()].push_back(max_clause_id_);
        occ_ternary_clauses[it_lit->var()].insert(occ_ternary_clauses[it_lit->var()].
↪end(),
            tmp.begin(), tmp.end());
      } else {
        occs[it_lit->var()].push_back(max_clause_id_);
        occs[it_lit->var()].push_back(occ_long_clauses[it_lit->var()].size());
        occ_long_clauses[it_lit->var()].insert(occ_long_clauses[it_lit->var()].end(),
            tmp.begin(), tmp.end());
        occ_long_clauses[it_lit->var()].push_back(SENTINEL_LIT.raw());
      }
    }
  }

  ComponentArchetype::initArrays(max_variable_id_, max_clause_id_);
  // the unified link list
  unified_variable_links_lists_pool_.clear();
  unified_variable_links_lists_pool_.push_back(0);
  unified_variable_links_lists_pool_.push_back(0);
  for (unsigned v = 1; v < occs.size(); v++) {
    // BEGIN data for binary clauses
    variable_link_list_offsets_[v] = unified_variable_links_lists_pool_.size();
    for (auto l : literals[LiteralID(v, false)].binary_links_)
      if (l != SENTINEL_LIT)
        unified_variable_links_lists_pool_.push_back(l.var());

    for (auto l : literals[LiteralID(v, true)].binary_links_)
      if (l != SENTINEL_LIT)
        unified_variable_links_lists_pool_.push_back(l.var());

    unified_variable_links_lists_pool_.push_back(0);

    // BEGIN data for ternary clauses
    unified_variable_links_lists_pool_.insert(
        unified_variable_links_lists_pool_.end(),
        occ_ternary_clauses[v].begin(),
        occ_ternary_clauses[v].end());

    unified_variable_links_lists_pool_.push_back(0);

    // BEGIN data for long clauses
    for (auto it = occs[v].begin(); it != occs[v].end(); it+=2) {
```

(continues on next page)

(continued from previous page)

```
      unified_variable_links_lists_pool_.push_back(*it);
      unified_variable_links_lists_pool_.push_back(*(it + 1) +(occs[v].end() - it));
    }

    unified_variable_links_lists_pool_.push_back(0);

    unified_variable_links_lists_pool_.insert(
        unified_variable_links_lists_pool_.end(),
        occ_long_clauses[v].begin(),
        occ_long_clauses[v].end());
  }
}


//void AltComponentAnalyzer::recordComponentOf(const VariableIndex var) {
//
//  search_stack_.clear();
//  setSeenAndStoreInSearchStack(var);
//
//  for (auto vt = search_stack_.begin(); vt != search_stack_.end(); vt++) {
//    //BEGIN traverse binary clauses
//    assert(isActive(*vt));
//    unsigned *p = beginOfLinkList(*vt);
//    for (; *p; p++) {
//      if (isUnseenAndActive(*p)) {
//        setSeenAndStoreInSearchStack(*p);
//        var_frequency_scores_[*p]++;
//        var_frequency_scores_[*vt]++;
//      }
//    }
//    //END traverse binary clauses
//    auto s = p;
//    for ( p++; *p ; p+=3) {
//    }
//    //END traverse ternary clauses
//
//    for (p++; *p ; p +=2) {
//      if (archetype_.clause_unseen_in_sup_comp(*p)) {
//        LiteralID * pstart_cls = reinterpret_cast<LiteralID *>(p + 1 + *(p+1));
//        searchClause(*vt,*p, pstart_cls);
//      }
//    }
//
//    for ( s++; *s ; s+=3) {
//          if (archetype_.clause_unseen_in_sup_comp(*s)) {
//            LiteralID * pstart_cls = reinterpret_cast<LiteralID *>(s + 1);
//            searchThreeClause(*vt,*s, pstart_cls);
//          }
//        }
//  }
//}

void AltComponentAnalyzer::recordComponentOf(const VariableIndex var) {
  search_stack_.clear();
  setSeenAndStoreInSearchStack(var);

  for (auto vt = search_stack_.begin(); vt != search_stack_.end(); vt++) {
```

(continues on next page)

(continued from previous page)

```
    // BEGIN traverse binary clauses
    assert(isActive(*vt));
    unsigned *p = beginOfLinkList(*vt);
    for (; *p; p++) {
      if (manageSearchOccurrenceOf(LiteralID(*p, true))) {
        var_frequency_scores_[*p]++;
        var_frequency_scores_[*vt]++;
      }
    }
    // END traverse binary clauses

    for (p++; *p ; p+=3) {
      if (archetype_.clause_unseen_in_sup_comp(*p)) {
        LiteralID litA = *reinterpret_cast<const LiteralID *>(p + 1);
        LiteralID litB = *(reinterpret_cast<const LiteralID *>(p + 1) + 1);
        if (isSatisfied(litA) || isSatisfied(litB)) {
          archetype_.setClause_nil(*p);
        } else {
          var_frequency_scores_[*vt]++;
          manageSearchOccurrenceAndScoreOf(litA);
          manageSearchOccurrenceAndScoreOf(litB);
          archetype_.setClause_seen(*p, isActive(litA) &
              isActive(litB));
        }
      }
    }
    // END traverse ternary clauses

    for (p++; *p ; p +=2)
      if (archetype_.clause_unseen_in_sup_comp(*p))
        searchClause(*vt, *p, reinterpret_cast<LiteralID *>(p + 1 + *(p+1)));
  }
}


bool AltComponentAnalyzer::exploreRemainingCompOf(VariableIndex v) {
  assert(archetype_.var_unseen_in_sup_comp(v));
  recordComponentOf(v);

  if (search_stack_.size() == 1) {
    archetype_.stack_level().includeSolution(2);
    archetype_.setVar_in_other_comp(v);
    return false;
  }
  return true;
}
```

Includes

- `alt_component_analyzer.h` (*File alt_component_analyzer.h*)
- `stack.h` (*File stack.h*)

File alt_component_analyzer.h

Parent directory (`src`)

Contents

- *Definition (`src/alt_component_analyzer.h`)*
- *Includes*
- *Included By*
- *Classes*

Definition (`src/alt_component_analyzer.h`)

Program Listing for File alt_component_analyzer.h

Return to documentation for file (`src/alt_component_analyzer.h`)

```
#ifndef ALT_COMPONENT_ANALYZER_H_
#define ALT_COMPONENT_ANALYZER_H_

#include <gmpxx.h>
#include <vector>
#include <cmath>

#include "statistics.h"
#include "component_types/component.h"
#include "component_types/base_packed_component.h"
#include "component_types/component_archetype.h"
#include "containers.h"


class AltComponentAnalyzer {
 public:
  AltComponentAnalyzer(DataAndStatistics &statistics,
        LiteralIndexedVector<TriValue> & lit_values) :
        literal_values_(lit_values) {
  }

  const unsigned scoreOf(VariableIndex v) const {
    return var_frequency_scores_[v];
  }

  ComponentArchetype &current_archetype() {
    return archetype_;
  }

  void initialize(LiteralIndexedVector<Literal> & literals,
      std::vector<LiteralID> &lit_pool);


  bool isUnseenAndActive(VariableIndex v) {
    assert(v <= max_variable_id_);
```

(continues on next page)

(continued from previous page)

```
    return archetype_.var_unseen_in_sup_comp(v);
  }

  // manages the literal whenever it occurs in component analysis
  // returns true iff the underlying variable was unseen before
  //
  bool manageSearchOccurrenceOf(LiteralID lit) {
      if (archetype_.var_unseen_in_sup_comp(lit.var())) {
        search_stack_.push_back(lit.var());
        archetype_.setVar_seen(lit.var());
        return true;
      }
      return false;
    }
  bool manageSearchOccurrenceAndScoreOf(LiteralID lit) {
    var_frequency_scores_[lit.var()]+= isActive(lit);
    return manageSearchOccurrenceOf(lit);
  }

  void setSeenAndStoreInSearchStack(VariableIndex v) {
    assert(isActive(v));
    search_stack_.push_back(v);
    archetype_.setVar_seen(v);
  }


  void setupAnalysisContext(StackLevel &top, Component & super_comp) {
     archetype_.reInitialize(top, super_comp);

     for (auto vt = super_comp.varsBegin(); *vt != varsSENTINEL; vt++)
       if (isActive(*vt)) {
         archetype_.setVar_in_sup_comp_unseen(*vt);
         var_frequency_scores_[*vt] = 0;
       }

     for (auto itCl = super_comp.clsBegin(); *itCl != clsSENTINEL; itCl++)
         archetype_.setClause_in_sup_comp_unseen(*itCl);
  }

  // returns true, iff the component found is non-trivial
  bool exploreRemainingCompOf(VariableIndex v);

  inline Component *makeComponentFromArcheType() {
    return archetype_.makeComponentFromState(search_stack_.size());
  }
  unsigned max_clause_id() {
     return max_clause_id_;
  }
  unsigned max_variable_id() {
    return max_variable_id_;
  }

  ComponentArchetype &getArchetype() {
    return archetype_;
  }

 private:
```

(continues on next page)

(continued from previous page)

```
//  /**
//   * statistics_ object is not used.  It is left for backward compatibility.
//   *
//   * ZH - Pragma added to prevent compiler warnings.
//   */
//  DataAndStatistics &statistics_;

  // the id of the last clause
  // note that clause ID is the clause number,
  // different from the offset of the clause in the literal pool
  unsigned max_clause_id_ = 0;
  unsigned max_variable_id_ = 0;

  // this contains clause offsets of the clauses
  // where each variable occurs in;
  std::vector<ClauseOfs> variable_occurrence_lists_pool_;

  std::vector<unsigned> unified_variable_links_lists_pool_;

  std::vector<unsigned> variable_link_list_offsets_;

  LiteralIndexedVector<TriValue> & literal_values_;

  std::vector<unsigned> var_frequency_scores_;

  ComponentArchetype archetype_;

  std::vector<VariableIndex> search_stack_;

  bool isResolved(const LiteralID lit) const {
    return literal_values_[lit] == F_TRI;
  }

  bool isSatisfied(const LiteralID lit) const {
    return literal_values_[lit] == T_TRI;
  }
  bool isActive(const LiteralID lit) const {
    return literal_values_[lit] == X_TRI;
  }

  bool isActive(const VariableIndex v) const {
    return literal_values_[LiteralID(v, true)] == X_TRI;
  }

  unsigned *beginOfLinkList(VariableIndex v) {
    return &unified_variable_links_lists_pool_[variable_link_list_offsets_[v]];
  }

  // stores all information about the component of var
  // in variables_seen_, clauses_seen_ and
  // component_search_stack
  // we have an isolated variable iff
  // after execution component_search_stack.size()==1
  void recordComponentOf(VariableIndex var);


  void getClause(std::vector<unsigned> &tmp,
```

(continues on next page)

(continued from previous page)

```
              std::vector<LiteralID>::iterator & it_start_of_cl,
              LiteralID & omitLit) {
    tmp.clear();
    for (auto it_lit = it_start_of_cl; *it_lit != SENTINEL_LIT; it_lit++) {
      if (it_lit->var() != omitLit.var())
        tmp.push_back(it_lit->raw());
    }
  }


  void searchClause(VariableIndex vt, ClauseIndex clID, LiteralID * pstart_cls) {
    auto itVEnd = search_stack_.end();
    bool all_lits_active = true;
    for (auto itL = pstart_cls; *itL != SENTINEL_LIT; itL++) {
      assert(itL->var() <= max_variable_id_);
      if (!archetype_.var_nil(itL->var())) {
        manageSearchOccurrenceAndScoreOf(*itL);
      } else {
        assert(!isActive(*itL));
        all_lits_active = false;
        if (isResolved(*itL))
          continue;
        //BEGIN accidentally entered a satisfied clause: undo the search process
        while (search_stack_.end() != itVEnd) {
          assert(search_stack_.back() <= max_variable_id_);
          archetype_.setVar_in_sup_comp_unseen(search_stack_.back());
          search_stack_.pop_back();
        }
        archetype_.setClause_nil(clID);
        while (*itL != SENTINEL_LIT)
          if (isActive(*(--itL)))
            var_frequency_scores_[itL->var()]--;
        //END accidentally entered a satisfied clause: undo the search process
        break;
      }
    }

    if (!archetype_.clause_nil(clID)) {
      var_frequency_scores_[vt]++;
      archetype_.setClause_seen(clID, all_lits_active);
    }
  }


//  void searchThreeClause(VariableIndex vt, ClauseIndex clID, LiteralID * pstart_
↪cls) {
//      auto itVEnd = search_stack_.end();
//      bool all_lits_active = true;
//      // LiteralID * pstart_cls = reinterpret_cast<LiteralID *>(p + 1 + *(p+1));
//      for (auto itL = pstart_cls; itL != pstart_cls+2; itL++) {
//        assert(itL->var() <= max_variable_id_);
//        if (archetype_.var_nil(itL->var())) {
//          assert(!isActive(*itL));
//          all_lits_active = false;
//          if (isResolved(*itL))
//            continue;
//          //BEGIN accidentally entered a satisfied clause: undo the search process
```

(continues on next page)

(continued from previous page)

```
//           while (search_stack_.end() != itVEnd) {
//             assert(search_stack_.back() <= max_variable_id_);
//             archetype_.setVar_in_sup_comp_unseen(search_stack_.back());
//             search_stack_.pop_back();
//           }
//           archetype_.setClause_nil(clID);
//           while (itL != pstart_cls - 1)
//             if (isActive(*(--itL)))
//               var_frequency_scores_[itL->var()]--;
//           //END accidentally entered a satisfied clause: undo the search process
//           break;
//         } else {
//           assert(isActive(*itL));
//           var_frequency_scores_[itL->var()]++;
//           if (isUnseenAndActive(itL->var()))
//             setSeenAndStoreInSearchStack(itL->var());
//         }
//       }
//
//       if (!archetype_.clause_nil(clID)) {
//         var_frequency_scores_[vt]++;
//         archetype_.setClause_seen(clID,all_lits_active);
//       }
//     }
};


#endif // ALT_COMPONENT_ANALYZER_H_
```

Includes

- `cmath`
- `component_types/base_packed_component.h` (*File base_packed_component.h*)
- `component_types/component.h` (*File component.h*)
- `component_types/component_archetype.h` (*File component_archetype.h*)
- `containers.h` (*File containers.h*)
- `gmpxx.h`
- `statistics.h` (*File statistics.h*)
- `vector`

Included By

- *File model_sampler.h*
- *File alt_component_analyzer.cpp*
- *File component_management.h*

Classes

- *Class AltComponentAnalyzer*

File base_packed_component.cpp

Parent directory (`src/component_types`)

Contents

- *Definition (src/component_types/base_packed_component.cpp)*
- *Includes*

Definition (`src/component_types/base_packed_component.cpp`)

Program Listing for File base_packed_component.cpp

Return to documentation for file (`src/component_types/base_packed_component.cpp`)

```
#include "base_packed_component.h"
#include <math.h>  // Needed since taking log based 2.
#include <iostream>

unsigned BasePackedComponent::_bits_per_clause = 0;
unsigned BasePackedComponent::_bits_per_variable = 0;  // bitsperentry
unsigned BasePackedComponent::_variable_mask = 0;
unsigned BasePackedComponent::_clause_mask = 0;  // bitsperentry
unsigned BasePackedComponent::_debug_static_val = 0;
unsigned BasePackedComponent::_bits_of_data_size = 0;
unsigned BasePackedComponent::_data_size_mask = 0;


void BasePackedComponent::adjustPackSize(unsigned int maxVarId, unsigned int maxClId)
↪{
  _bits_per_variable = log2(maxVarId) + 1;
  _bits_per_clause   = log2(maxClId) + 1;

  _bits_of_data_size = log2(maxVarId + maxClId) + 1;

  _variable_mask = _clause_mask = _data_size_mask = 0;
  // Iteratively bit shift to create the mask variable
  for (unsigned int i = 0; i < _bits_per_variable; i++)
    _variable_mask = (_variable_mask << 1) + 1;
  for (unsigned int i = 0; i < _bits_per_clause; i++)
    _clause_mask = (_clause_mask << 1) + 1;
  for (unsigned int i = 0; i < _bits_of_data_size; i++)
    _data_size_mask = (_data_size_mask << 1) + 1;
}
```

Includes

- base_packed_component.h (*File base_packed_component.h*)
- iostream
- math.h

File base_packed_component.h

Parent directory (src/component_types)

Contents

- *Definition (src/component_types/base_packed_component.h)*
- *Includes*
- *Included By*
- *Classes*
- *Defines*

Definition (src/component_types/base_packed_component.h)

Program Listing for File base_packed_component.h

Return to documentation for file (src/component_types/base_packed_component.h)

```
#ifndef BASE_PACKED_COMPONENT_H_
#define BASE_PACKED_COMPONENT_H_

#include <assert.h>
#include <gmpxx.h>
#include <iostream>

//using namespace std;

template <class T> class BitStuffer {
 public:
  explicit BitStuffer(T *data) : data_start_(data), p(data) {
    *p = 0;
  }

  void stuff(const unsigned val, const unsigned num_bits_val) {
    assert(num_bits_val > 0);  // Verify a number of bits is specified
    assert((val >> num_bits_val) == 0);  // Verify no hanging over bits in the passed␣
↪in value

    // Clear the memory at the specified location to make sure it is empty
    if (end_of_bits_ == 0)
      *p = 0;

    // Ensure there are no 1 bits after the end_of_bits_ bit location.
```

(continues on next page)

(continued from previous page)

```
    assert((*p >> end_of_bits_) == 0);
    // Insert val into the memory location pointed to by p.
    *p |= val << end_of_bits_;
    // Shift the bit location by the number of bits in the vlue
    end_of_bits_ += num_bits_val;
    if (end_of_bits_ > _bits_per_block) {
      //assert(*p);
      // Roll over the bit count
      end_of_bits_ -= _bits_per_block;
      *(++p) = val >> (num_bits_val - end_of_bits_);
      assert(end_of_bits_ != 0 || (*p == 0));
    } else if (end_of_bits_ == _bits_per_block) {
      end_of_bits_ -= _bits_per_block;
      p++;
    }
  }
  void assert_size(unsigned size) {
    if (end_of_bits_ == 0)
       p--;
    assert(p - data_start_ == size - 1);
  }

 private:
  T *data_start_ = nullptr;
  T *p = nullptr;
  unsigned end_of_bits_ = 0;
  static const unsigned _bits_per_block = (sizeof(T) << 3);
};


class BasePackedComponent {
 public:
  static unsigned bits_per_variable() {
    return _bits_per_variable;
  }
  static unsigned variable_mask() {
      return _variable_mask;
  }
  static unsigned bits_per_clause() {
    return _bits_per_clause;
  }

  static unsigned bits_per_block() {
    return _bits_per_block;
  }

  static unsigned bits_of_data_size() {
    return _bits_of_data_size;
  }

  static void adjustPackSize(unsigned int maxVarId, unsigned int maxClId);

  BasePackedComponent() = default;

  explicit BasePackedComponent(unsigned creation_time) : creation_time_(creation_
↪time) {}
```

(continues on next page)

(continued from previous page)

```
  ~BasePackedComponent() {
//    if (data_)  // Deleting nullptr has no effect so no need for if
//      delete data_;
    delete data_;
  }
  static void outbit(unsigned v) {
    for (auto i=0; i < 32; i++) {
      std::cout << ((v&2147483648)?"1":"0");  // 2147483648 == 2^31
      v&=2147483648-1;
      v <<= 1;
    }
  }


  static unsigned log2(unsigned v) {
    // taken from
    // http://graphics.stanford.edu/~seander/bithacks.html#IntegerLogLookup
    static const char LogTable256[256] = {
      #define LT(n) n, n, n, n, n, n, n, n, n, n, n, n, n, n, n, n
      -1, 0, 1, 1, 2, 2, 2, 2, 3, 3, 3, 3, 3, 3, 3, 3,
      LT(4), LT(5), LT(5), LT(6), LT(6), LT(6), LT(6),
      LT(7), LT(7), LT(7), LT(7), LT(7), LT(7), LT(7), LT(7)
    };

    unsigned r;     // r will be lg(v)
    unsigned int t, tt;  // temporaries

    if ((tt = (v >> 16))) {
      r = (t = (tt >> 8)) ? 24 + LogTable256[t] : 16 + LogTable256[tt];
    } else {
      r = (t = (v >> 8)) ? 8 + LogTable256[t] : LogTable256[v];
    }
    return r;
  }

  unsigned creation_time() {
    return creation_time_;
  }

  const mpz_class &model_count() const {
    return model_count_;
  }

  unsigned alloc_of_model_count() const {
    return sizeof(mpz_class) + model_count_.get_mpz_t()->_mp_alloc * sizeof(mp_limb_
↪t);
  }

  void set_creation_time(unsigned time) {
    creation_time_ = time;
  }

  void set_model_count(const mpz_class &rn, unsigned time) {
    model_count_ = rn;
    length_solution_period_and_flags_ = (time - creation_time_)
                                        | (length_solution_period_and_flags_ & 1);
  }
```

(continues on next page)

(continued from previous page)

```
  unsigned hashkey() const  {
    return hashkey_;
  }

  bool modelCountFound() {
    return (length_solution_period_and_flags_ >> 1);
  }

  bool isDeletable() const {
    return length_solution_period_and_flags_ & 1;
  }
  void set_deletable() {
    length_solution_period_and_flags_ |= 1;
  }

  void clear() {
    // before deleting the contents of this component,
    // we should make sure that this component is not present in the component stack
↪anymore!
    assert(isDeletable());
//    if (data_)  // If statement unnecessary as deleting nullptr as no effect
//      delete data_;
    delete data_;
    data_ = nullptr;
  }

  static unsigned _debug_static_val;

 protected:
  unsigned* data_ = nullptr;

  unsigned hashkey_ = 0;

  mpz_class model_count_;

  unsigned creation_time_ = 1;


  // this is:  length_solution_period = length_solution_period_and_flags_ >> 1
  // length_solution_period == 0 means unsolved
  // and the first bit is "delete_permitted"
  unsigned length_solution_period_and_flags_ = 0;

  // deletion is permitted only after
  // the copy of this component in the stack
  // does not exist anymore

 protected:
  static unsigned _bits_per_clause;
  static unsigned _bits_per_variable;  // bitsperentry
  static unsigned _bits_of_data_size;  // number of bits needed to store the data
↪size.
  static unsigned _data_size_mask;
  static unsigned _variable_mask;
  static unsigned _clause_mask;
  static const unsigned _bits_per_block = (sizeof(unsigned) << 3);
```

(continues on next page)

(continued from previous page)

```
};

#endif /* BASE_PACKED_COMPONENT_H_ */
```

Includes

- assert.h
- gmpxx.h
- iostream

Included By

- *File difference_packed_component.h*
- *File alt_component_analyzer.h*
- *File base_packed_component.cpp*

Classes

- *Class BasePackedComponent*
- *Template Class BitStuffer*

Defines

- *Define LT*

File cacheable_component.h

Parent directory (src/component_types)

Contents

- *Definition (src/component_types/cacheable_component.h)*
- *Includes*
- *Included By*
- *Classes*
- *Defines*
- *Typedefs*

Definition (`src/component_types/cacheable_component.h`)

Program Listing for File cacheable_component.h

Return to documentation for file (`src/component_types/cacheable_component.h`)

```
/*
 * cacheable_component.h
 *
 *  Created on: Feb 21, 2013
 *      Author: mthurley
 */

#ifndef CACHEABLE_COMPONENT_H_
#define CACHEABLE_COMPONENT_H_

#include <assert.h>
#include <vector>

#include "../primitive_types.h"

#include "difference_packed_component.h"
//#include "simple_unpacked_component.h"


//using namespace std;

#define NIL_ENTRY 0

class Component;
class ComponentArchetype;

// GenericCacheableComponent Adds Structure to PackedComponent that is
// necessary to store it in the cache
// namely, the descendant tree structure that
// allows for the removal of cache pollutions

template< class T_Component>
class GenericCacheableComponent: public T_Component {
 public:
  GenericCacheableComponent() = default;

  explicit GenericCacheableComponent(Component &comp) :
    T_Component(comp) {
  }

  unsigned long SizeInBytes() const {
    return sizeof(GenericCacheableComponent<T_Component>) + T_Component::raw_data_
↪byte_size();
  }

  // the 48 = 16*3 in overhead stems from the three parts of the component
  // being dynamically allocated (i.e. the GenericCacheableComponent itself,
  // the data_ and the model_count data
  unsigned long sys_overhead_SizeInBytes() const {
    return sizeof(GenericCacheableComponent<T_Component>)
           + T_Component::sys_overhead_raw_data_byte_size()
```

(continues on next page)

(continued from previous page)

```
            // + 24;
            +48;
  }

  // BEGIN Cache Pollution Management

  void set_father(CacheEntryID f) {
    father_ = f;
  }
  const CacheEntryID father() const {
    return father_;
  }

  void set_next_sibling(CacheEntryID sibling) {
    next_sibling_ = sibling;
  }
  CacheEntryID next_sibling() {
    return next_sibling_;
  }

  void set_first_descendant(CacheEntryID descendant) {
    first_descendant_ = descendant;
  }
  const CacheEntryID first_descendant() const {
    return first_descendant_;
  }

  void set_next_bucket_element(CacheEntryID entry) {
    next_bucket_element_ = entry;
  }

  CacheEntryID next_bucket_element() {
    return next_bucket_element_;
  }

 private:
  CacheEntryID next_bucket_element_ = 0;

  // theFather and theDescendants:
  // each CCacheEntry is a Node in a tree which represents the relationship
  // of the components stored
  CacheEntryID father_ = 0;
  CacheEntryID first_descendant_ = 0;
  CacheEntryID next_sibling_ = 0;
};



typedef GenericCacheableComponent<DifferencePackedComponent> CacheableComponent;
//typedef GenericCacheableComponent<SimplePackedComponent> CacheableComponent;

#endif /* CACHEABLE_COMPONENT_H_ */
```

Includes

- ../primitive_types.h

- assert.h
- difference_packed_component.h (*File difference_packed_component.h*)
- vector

Included By

- *File statistics.h*
- *File component_archetype.h*
- *File component_cache.h*

Classes

- *Template Class GenericCacheableComponent*

Defines

- *Define NIL_ENTRY*

Typedefs

- *Typedef CacheableComponent*

File cached_assignment.h

Parent directory (src)

Contents

- *Definition (src/cached_assignment.h)*
- *Includes*
- *Included By*
- *Classes*
- *Defines*

Definition (src/cached_assignment.h)

Program Listing for File cached_assignment.h

Return to documentation for file (src/cached_assignment.h)

```
#ifndef SHARPSAT_CACHED_ASSIGNMENT_H
#define SHARPSAT_CACHED_ASSIGNMENT_H

#include <algorithm>
#include <vector>
#include "structures.h"

#define CACHED_VARIABLE_LEN 2 // In Bits

class CachedAssignment{
 public:
  CachedAssignment() {
    assigned_literals_.reserve(1);
    emancipated_vars_.reserve(1);
  }
  inline void Sort() {
    std::sort(assigned_literals_.begin(), assigned_literals_.end());
    std::sort(emancipated_vars_.begin(), emancipated_vars_.begin());
  }
  inline void IncreaseSize(VariableIndex size_adder) {
    VariableIndex new_size = assigned_literals_.capacity() + size_adder;
    assigned_literals_.reserve(new_size);
  }
//  /**
//   * Add the specified literal to the list of literals in the cache
//   * assignment.
//   *
//   * @param lit New literal value.
//   */
//  inline void AddLiteral(LiteralID lit) {assigned_literals_.push_back(lit);}
  const std::vector<VariableIndex> vars() {
    std::vector<VariableIndex> cached_vars;
    cached_vars.reserve(assigned_literals_.size());

    for (auto lit : assigned_literals_)
      cached_vars.push_back(lit.var());
    cached_vars.insert(cached_vars.end(), emancipated_vars_.begin(), emancipated_vars_
↪.end());
    return cached_vars;
  }
  const VariableIndex num_components() const { return num_cached_components; }
  const bool empty() const {return num_cached_components == 0; }
  void ProcessComponent(const Component * comp, mpz_class model_count_and_assn) {
    // Nothing to store in the UNSAT case
    if (model_count_and_assn == 0)
      return;
    IncreaseSize(comp->num_variables());  // Increase the capacity

    // Extract the literal assignments from the MPZ object
    std::vector<ComponentVarAndCls> comp_data = comp->getData();
    for (VariableIndex i = 0; i < comp->num_variables(); i++) {
      VariableIndex bit_index = CACHED_VARIABLE_LEN * i;
      // Check if the variable is emancipated
      if (mpz_tstbit(model_count_and_assn.get_mpz_t(), bit_index + 1) == 1) {
        emancipated_vars_.emplace_back(comp_data[i]);
      } else {
        bool sign = (mpz_tstbit(model_count_and_assn.get_mpz_t(), bit_index) == 1);
```

(continues on next page)

(continued from previous page)

```
        assigned_literals_.emplace_back(LiteralID(comp_data[i], sign));
      }
    }
    num_cached_components++;
  }
  const std::vector<LiteralID>& literals() const { return assigned_literals_; }
  const std::vector<VariableIndex>& emancipated_vars() const { return emancipated_
↪vars_; }
  void clear() {
    assigned_literals_.clear();
    emancipated_vars_.clear();
    num_cached_components = 0;
  }
//  /**
//   * Component Split Accessor
//   *
//   * Accessor for whether this cached assignment corresponds with a component
//   * split.
//   *
//   * @return true if this cached assignment corresponds to a component split.
//   */
//  const bool is_component_split() const { return is_component_split_;}

 private:
  std::vector<VariableIndex> emancipated_vars_;
  std::vector<LiteralID> assigned_literals_;
  unsigned long num_cached_components = 0;
};

#endif // SHARPSAT_CACHED_ASSIGNMENT_H
```

Includes

- `algorithm`
- `structures.h` (*File structures.h*)
- `vector`

Included By

- *File model_sampler.h*
- *File stack.h*
- *File component_cache.h*
- *File solver.h*

Classes

- *Class CachedAssignment*

Defines

- *Define CACHED_VARIABLE_LEN*

File component.h

Parent directory (`src/component_types`)

Contents

- *Definition (`src/component_types/component.h`)*
- *Includes*
- *Included By*
- *Classes*

Definition (`src/component_types/component.h`)

Program Listing for File component.h

Return to documentation for file (`src/component_types/component.h`)

```
#ifndef COMPONENT_H_
#define COMPONENT_H_

#include <assert.h>
#include <vector>

#include "../primitive_types.h"

//using namespace std;

class Component {
 public:
  void reserveSpace(unsigned int num_variables, unsigned int num_clauses) {
    data_.reserve(num_variables + num_clauses + 2);
  }
  void set_id(CacheEntryID id) {
    id_ = id;
  }

  CacheEntryID id() const {
    return id_;
  }

  void addVar(const VariableIndex var) {
    // the only time a varsSENTINEL is added should be in a
    // call to closeVariableData(..)
    assert(var != varsSENTINEL);
    data_.push_back(var);
  }
```

(continues on next page)

(continued from previous page)

```
  void closeVariableData() {
    data_.push_back(varsSENTINEL);
    clauses_ofs_ = (unsigned) data_.size();
  }

  void addCl(const ClauseIndex cl) {
    // the only time a clsSENTINEL is added should be in a
    // call to closeClauseData(..)
    assert(cl != clsSENTINEL);
    data_.push_back(cl);
  }

  void closeClauseData() {
    data_.push_back(clsSENTINEL);
    assert(*(clsBegin()-1) == 0);
  }

  std::vector<VariableIndex>::const_iterator varsBegin() const {
    return data_.begin();
  }

  std::vector<ClauseIndex>::const_iterator clsBegin() const {
    return data_.begin() + clauses_ofs_;
  }

  unsigned num_variables() const {
    return clauses_ofs_ - 1;
  }

  unsigned numLongClauses() const {
    return (unsigned) data_.size() - clauses_ofs_ - 1;
  }
  bool empty() const {
    return data_.empty();
  }

  void createAsDummyComponent(unsigned max_var_id, unsigned max_clause_id) {
    data_.clear();
    clauses_ofs_ = 1;
    for (unsigned idvar = 1; idvar <= max_var_id; idvar++)
      addVar(idvar);
    closeVariableData();
    if (max_clause_id > 0) {
      for (unsigned idcl = 1; idcl <= max_clause_id; idcl++)
        addCl(idcl);
    }
    closeClauseData();
  }

  void clear() {
    clauses_ofs_ = 0;
    data_.clear();
  }
  unsigned clauses_ofs() const {
    return clauses_ofs_;
  }
```

(continues on next page)

(continued from previous page)

```
  const std::vector<ComponentVarAndCls >& getData() const { return data_; }

 private:
  // data_ stores the component data:
  // for better cache performance the
  // clause and variable data are stored in
  // a contiguous piece of memory
  // variables SENTINEL clauses SENTINEL
  // this order has to be taken care of on filling
  // in the data!
  std::vector<ComponentVarAndCls> data_;
  unsigned clauses_ofs_ = 0;
  // id_ will identify denote the entry in the cacheable component database,
  // where a Packed version of this component is stored
  // yet this does not imply that the model count of this component is already known
  // once the model count is known, a link to the packed component will be stored
  // in the hash table
  CacheEntryID id_ = 0;
};



#endif /* COMPONENT_H_ */
```

Includes

- ../primitive_types.h
- assert.h
- vector

Included By

- *File model_sampler.h*
- *File difference_packed_component.h*
- *File alt_component_analyzer.h*
- *File component_archetype.h*
- *File component_cache.h*
- *File component_management.h*

Classes

- *Class Component*

File component_archetype.cpp

Parent directory (src/component_types)

Contents

- *Definition (src/component_types/component_archetype.cpp)*
- *Includes*

Definition (src/component_types/component_archetype.cpp)

Program Listing for File component_archetype.cpp

Return to documentation for file (src/component_types/component_archetype.cpp)

```
#include "component_archetype.h"

CA_SearchState *ComponentArchetype::seen_ = nullptr;
unsigned ComponentArchetype::seen_byte_size_ = 0;
```

Includes

- component_archetype.h (*File component_archetype.h*)

File component_archetype.h

Parent directory (src/component_types)

Contents

- *Definition (src/component_types/component_archetype.h)*
- *Includes*
- *Included By*
- *Classes*
- *Defines*
- *Typedefs*

Definition (src/component_types/component_archetype.h)

Program Listing for File component_archetype.h

Return to documentation for file (src/component_types/component_archetype.h)

```
#ifndef COMPONENT_ARCHETYPE_H_
#define COMPONENT_ARCHETYPE_H_
```

(continues on next page)

(continued from previous page)

```
#include <cstring>
#include <algorithm>
#include <iostream>

#include "../primitive_types.h"
#include "component.h"
#include "cacheable_component.h"

// State values for variables found during component
// analysis (CA)
typedef unsigned char CA_SearchState;
#define   CA_NIL  0
#define   CA_VAR_IN_SUP_COMP_UNSEEN  1
#define   CA_VAR_SEEN 2
#define   CA_VAR_IN_OTHER_COMP  4

#define   CA_VAR_MASK  7

#define   CA_CL_IN_SUP_COMP_UNSEEN  8
#define   CA_CL_SEEN 16
#define   CA_CL_IN_OTHER_COMP  32
#define   CA_CL_ALL_LITS_ACTIVE  64

#define   CA_CL_MASK  120

class StackLevel;

class ComponentArchetype {
 public:
  ComponentArchetype() = default;

  ComponentArchetype(StackLevel &stack_level, Component &super_comp) :
      p_super_comp_(&super_comp), p_stack_level_(&stack_level) {
  }

  void reInitialize(StackLevel &stack_level, Component &super_comp) {
    p_super_comp_ = &super_comp;
    p_stack_level_ = &stack_level;
    clearArrays();
    current_comp_for_caching_.reserveSpace(super_comp.num_variables(), super_comp.
↪numLongClauses());
  }

  Component &super_comp() {
    return *p_super_comp_;
  }

  StackLevel & stack_level() {
    return *p_stack_level_;
  }

  void setVar_in_sup_comp_unseen(VariableIndex v) {
    seen_[v] = CA_VAR_IN_SUP_COMP_UNSEEN | (seen_[v] & CA_CL_MASK);
  }

  void setClause_in_sup_comp_unseen(ClauseIndex cl) {
    seen_[cl] = CA_CL_IN_SUP_COMP_UNSEEN | (seen_[cl] & CA_VAR_MASK);
```

(continues on next page)

(continued from previous page)

```
  }

  void setVar_nil(VariableIndex v) {
    seen_[v] &= CA_CL_MASK;
  }

  void setClause_nil(ClauseIndex cl) {
    seen_[cl] &= CA_VAR_MASK;
  }

  void setVar_seen(VariableIndex v) {
    seen_[v] = CA_VAR_SEEN | (seen_[v] & CA_CL_MASK);
  }

  void setClause_seen(ClauseIndex cl) {
    setClause_nil(cl);
    seen_[cl] = CA_CL_SEEN | (seen_[cl] & CA_VAR_MASK);
  }

  void setClause_seen(ClauseIndex cl, bool all_lits_act) {
    setClause_nil(cl);
    seen_[cl] = CA_CL_SEEN | (all_lits_act ? CA_CL_ALL_LITS_ACTIVE : 0) | (seen_[cl] &
↪ CA_VAR_MASK);
  }

  void setVar_in_other_comp(VariableIndex v) {
    seen_[v] = CA_VAR_IN_OTHER_COMP | (seen_[v] & CA_CL_MASK);
  }

  void setClause_in_other_comp(ClauseIndex cl) {
    seen_[cl] = CA_CL_IN_OTHER_COMP | (seen_[cl] & CA_VAR_MASK);
  }

  bool var_seen(VariableIndex v) const {
    return seen_[v] & CA_VAR_SEEN;
  }

  bool clause_seen(ClauseIndex cl) const {
    return seen_[cl] & CA_CL_SEEN;
  }

  bool clause_all_lits_active(ClauseIndex cl) {
    return seen_[cl] & CA_CL_ALL_LITS_ACTIVE;
  }
  void setClause_all_lits_active(ClauseIndex cl) const {
    seen_[cl] |= CA_CL_ALL_LITS_ACTIVE;
  }

  bool var_nil(VariableIndex v) const {
    return (seen_[v] & CA_VAR_MASK) == 0;
  }

  bool clause_nil(ClauseIndex cl) const {
    return (seen_[cl] & CA_CL_MASK) == 0;
  }

  bool var_unseen_in_sup_comp(VariableIndex v) const {
```

(continues on next page)

(continued from previous page)

```
    return seen_[v] & CA_VAR_IN_SUP_COMP_UNSEEN;
  }

  bool clause_unseen_in_sup_comp(ClauseIndex cl) const {
    return seen_[cl] & CA_CL_IN_SUP_COMP_UNSEEN;
  }

  bool var_seen_in_peer_comp(VariableIndex v) const {
    return seen_[v] & CA_VAR_IN_OTHER_COMP;
  }

  bool clause_seen_in_peer_comp(ClauseIndex cl) const {
    return seen_[cl] & CA_CL_IN_OTHER_COMP;
  }

  static void initArrays(unsigned max_variable_id, unsigned max_clause_id) {
    unsigned seen_size = std::max(max_variable_id, max_clause_id)  + 1;
    seen_ = new CA_SearchState[seen_size];
    seen_byte_size_ = sizeof(CA_SearchState) * (seen_size);
    clearArrays();
  }

  static void clearArrays() {
    memset(seen_, CA_NIL, seen_byte_size_);
  }


  Component *makeComponentFromState(unsigned long stack_size) {
    Component *p_new_comp = new Component();
    p_new_comp->reserveSpace(stack_size, super_comp().numLongClauses());
    current_comp_for_caching_.clear();

    for (auto v_it = super_comp().varsBegin(); *v_it != varsSENTINEL;  v_it++)
      if (var_seen(*v_it)) {  //we have to put a var into our component
        p_new_comp->addVar(*v_it);
        current_comp_for_caching_.addVar(*v_it);
        setVar_in_other_comp(*v_it);
      }
    p_new_comp->closeVariableData();
    current_comp_for_caching_.closeVariableData();

    for (auto it_cl = super_comp().clsBegin(); *it_cl != clsSENTINEL; it_cl++)
      if (clause_seen(*it_cl)) {
        p_new_comp->addCl(*it_cl);
           if (!clause_all_lits_active(*it_cl))
             current_comp_for_caching_.addCl(*it_cl);
        setClause_in_other_comp(*it_cl);
      }
    p_new_comp->closeClauseData();
    current_comp_for_caching_.closeClauseData();
    return p_new_comp;
  }
//  Component *makeComponentFromState(unsigned stack_size) {
//      Component *p_new_comp = new Component();
//      p_new_comp->reserveSpace(stack_size, super_comp().numLongClauses());
//
//      for (auto v_it = super_comp().varsBegin(); *v_it != varsSENTINEL;  v_it++)
```

(continues on next page)

(continued from previous page)

```
//        if (var_seen(*v_it)) {  //we have to put a var into our component
//          p_new_comp->addVar(*v_it);
//          setVar_in_other_comp(*v_it);
//        }
//      p_new_comp->closeVariableData();
//
//      for (auto it_cl = super_comp().clsBegin(); *it_cl != clsSENTINEL; it_cl++)
//        if (clause_seen(*it_cl)) {
//          p_new_comp->addCl(*it_cl);
//          setClause_in_other_comp(*it_cl);
//        }
//      p_new_comp->closeClauseData();
//      return p_new_comp;
//    }

//  inline void createComponents(Component &ret_comp, CacheableComponent ret_cache_
↪comp,
//                               unsigned stack_size);

  Component current_comp_for_caching_;

 private:
  Component *p_super_comp_;
  StackLevel *p_stack_level_;

  static CA_SearchState *seen_;
  static unsigned seen_byte_size_;
};




//void ComponentArchetype::createComponents(Component &ret_comp,
//                                          CacheableComponent ret_cache_comp,␣
↪unsigned stack_size) {
//}


#endif /* COMPONENT_ARCHETYPE_H_ */
```

Includes

- ../primitive_types.h
- algorithm
- cacheable_component.h (*File cacheable_component.h*)
- component.h (*File base_packed_component.h*)
- cstring
- iostream

Included By

- *File alt_component_analyzer.h*
- *File component_archetype.cpp*

Classes

- *Class ComponentArchetype*

Defines

- *Define CA_CL_ALL_LITS_ACTIVE*
- *Define CA_CL_IN_OTHER_COMP*
- *Define CA_CL_IN_SUP_COMP_UNSEEN*
- *Define CA_CL_MASK*
- *Define CA_CL_SEEN*
- *Define CA_NIL*
- *Define CA_VAR_IN_OTHER_COMP*
- *Define CA_VAR_IN_SUP_COMP_UNSEEN*
- *Define CA_VAR_MASK*
- *Define CA_VAR_SEEN*

Typedefs

- *Typedef CA_SearchState*

File component_cache.cpp

Parent directory (`src`)

Contents

- *Definition (`src/component_cache.cpp`)*
- *Includes*

Definition (`src/component_cache.cpp`)

Program Listing for File component_cache.cpp

Return to documentation for file (`src/component_cache.cpp`)

```
#include <algorithm>
#include <vector>
#include "component_cache.h"
#include "sampler_tools.h"

#ifdef __linux__

  #include <sys/sysinfo.h>
  #include <cstdint>

  uint64_t freeram() {
    struct sysinfo info;
        sysinfo(&info);

    return info.freeram *(uint64_t) info.mem_unit;
  }

#elif __APPLE__ && __MACH__

  #include <sys/sysctl.h>

  uint64_t freeram() {
    int mib[2];
    int64_t physical_memory;
    mib[0] = CTL_HW;
    mib[1] = HW_MEMSIZE;
    size_t length = sizeof(int64_t);
    sysctl(mib, 2, &physical_memory, &length, NULL, 0);

    return physical_memory;
  }

#else

#endif



ComponentCache::ComponentCache(DataAndStatistics &statistics, SolverConfiguration &
↪config) :
    statistics_(statistics), config_(config) {
}

void ComponentCache::init(Component &super_comp, bool quiet) {
  if (!quiet) {
    std::cout << "Initialize cache\n"
              << "Size of Cacheable Component:\t" << sizeof(CacheableComponent) << "\n
↪"
              << "Size of MPZ Class:\t" << sizeof(mpz_class) << std::endl;
  }
  CacheableComponent &packed_super_comp = *new CacheableComponent(super_comp);
  my_time_ = 1;

  entry_base_.clear();
  entry_base_.reserve(2000000);
  entry_base_.push_back(new CacheableComponent());  // dummy Element
  table_.clear();
```

(continues on next page)

(continued from previous page)

```
  table_.resize(1024*1024, 0);
  table_size_mask_ = table_.size() - 1;

  free_entry_base_slots_.clear();
  free_entry_base_slots_.reserve(10000);

  uint64_t free_ram = freeram();
  uint64_t max_cache_bound = 95 * (free_ram / 100);

  if (statistics_.maximum_cache_size_bytes_ == 0) {
    statistics_.maximum_cache_size_bytes_ = max_cache_bound;
  }

  if (!quiet) {
    if (statistics_.maximum_cache_size_bytes_ > free_ram) {
      std::cout << "\n";
      //PrintWarning("Maximum cache size larger than free RAM available");
      std::cout << " Free RAM " << free_ram / 1000000 << "MB\n";
    }
    std::cout << "Maximum cache size:\t"
         << statistics_.maximum_cache_size_bytes_ / 1000000 << " MB\n"
         << std::endl;
  }

  assert(!statistics_.cache_full());

  if (entry_base_.capacity() == entry_base_.size())
    entry_base_.reserve(2 * entry_base_.size());

  entry_base_.push_back(&packed_super_comp);

  statistics_.incorporate_cache_store(packed_super_comp);

  super_comp.set_id(1);
}

void ComponentCache::test_descendantstree_consistency() {
  for (unsigned id = 2; id < entry_base_.size(); id++)
    if (entry_base_[id] != nullptr) {
      CacheEntryID act_child = entry(id).first_descendant();
      while (act_child) {
        CacheEntryID next_child = entry(act_child).next_sibling();
        assert(entry(act_child).father() == id);

        act_child = next_child;
      }
      CacheEntryID father = entry(id).father();
      CacheEntryID act_sib = entry(father).first_descendant();

      bool found = false;
      while (act_sib && !found) {
        CacheEntryID next_sib = entry(act_sib).next_sibling();
        if (act_sib == id)
          found = true;
        act_sib = next_sib;
      }
      assert(found);
```

(continues on next page)

(continued from previous page)

```
    }
}

bool ComponentCache::deleteEntries() {
  assert(statistics_.cache_full());

  // Build a list of all the scores, sorts the list, then delete all
  // entries with a score below the median.
  std::vector<double> scores;
  for (auto it = entry_base_.begin() + 1; it != entry_base_.end(); it++)
    if (*it != nullptr && (*it)->isDeletable()) {
      scores.push_back(static_cast<double>((*it)->creation_time()));
    }
  std::sort(scores.begin(), scores.end());
  double cutoff = scores[scores.size() / 2];

  //std::cout << "cutoff" << cutoff  << " entries: "<< entry_base_.size()<< std::endl;

  // first : go through the EntryBase and mark the entries to be deleted as deleted␣
↪(i.e. EMPTY
  // note we start at index 2,
  // since index 1 is the whole formula,
  // should always stay here!
  for (unsigned id = 2; id < entry_base_.size(); id++)
    if (entry_base_[id] != nullptr &&
        entry_base_[id]->isDeletable() &&
        static_cast<double>(entry_base_[id]->creation_time()) <= cutoff) {
      removeFromDescendantsTree(id);
      eraseEntry(id);
    }
  // then go through the Hash Table and erase all Links to empty entries


#ifdef DEBUG
  test_descendantstree_consistency();
#endif

  reHashTable(table_.size());
  statistics_.sum_size_cached_components_ = 0;
  statistics_.sum_bytes_cached_components_ = 0;
  statistics_.sys_overhead_sum_bytes_cached_components_ = 0;

  statistics_.sum_bytes_pure_cached_component_data_ = 0;

  for (unsigned id = 2; id < entry_base_.size(); id++)
    if (entry_base_[id] != nullptr) {
      statistics_.sum_size_cached_components_ +=
          entry_base_[id]->num_variables();
      statistics_.sum_bytes_cached_components_ +=
          entry_base_[id]->SizeInBytes();
      statistics_.sum_bytes_pure_cached_component_data_ +=
          entry_base_[id]->data_only_byte_size();
       statistics_.sys_overhead_sum_bytes_cached_components_ +=
           entry_base_[id]->sys_overhead_SizeInBytes();
    }

  statistics_.num_cached_components_ = entry_base_.size();
```

(continues on next page)

(continued from previous page)

```
  compute_byte_size_infrastructure();

  //std::cout << " \t entries: "<< entry_base_.size() - free_entry_base_slots_.size()<
↪< std::endl;
  return true;
}


uint64_t ComponentCache::compute_byte_size_infrastructure() {
  statistics_.cache_infrastructure_bytes_memory_usage_ =
      sizeof(ComponentCache)
      + sizeof(CacheEntryID)* table_.capacity()
      + sizeof(CacheableComponent *)* entry_base_.capacity()
      + sizeof(CacheEntryID) * free_entry_base_slots_.capacity();
  return statistics_.cache_infrastructure_bytes_memory_usage_;
}

void ComponentCache::debug_dump_data() {
  std::cout << "sizeof (CacheableComponent *, CacheEntryID) "
       << sizeof(CacheableComponent *) << ", "
       << sizeof(CacheEntryID) << std::endl;
  std::cout << "table (size/capacity) " << table_.size()
       << "/" << table_.capacity() << std::endl;
  std::cout << "entry_base_ (size/capacity) " << entry_base_.size()
       << "/" << entry_base_.capacity() << std::endl;
  std::cout << "free_entry_base_slots_ (size/capacity) " << free_entry_base_slots_.
↪size()
       << "/" << free_entry_base_slots_.capacity() << std::endl;

//  uint64_t size_model_counts = 0;
  uint64_t alloc_model_counts = 0;
  for (auto &pentry : entry_base_)
    if (pentry != nullptr) {
//      size_model_counts += pentry->size_of_model_count();
      alloc_model_counts += pentry->alloc_of_model_count();
    }
  std::cout << "model counts size " << alloc_model_counts << std::endl;
}
```

Includes

- `algorithm`
- `component_cache.h` (*File component_cache.h*)
- `sampler_tools.h` (*File sampler_tools.h*)
- `vector`

File component_cache.h

Parent directory (`src`)

Contents

- *Definition (src/component_cache.h)*
- *Includes*
- *Included By*
- *Classes*

Definition (src/component_cache.h)

Program Listing for File component_cache.h

Return to documentation for file (src/component_cache.h)

```
#ifndef COMPONENT_CACHE_H_
#define COMPONENT_CACHE_H_

#include <gmpxx.h>

#include <vector>
#include <sstream>

#include "component_types/cacheable_component.h"
#include "statistics.h"
#include "component_types/component.h"
#include "stack.h"
#include "solver_config.h"
#include "cached_assignment.h"

class ComponentCache {
 public:
  ComponentCache(DataAndStatistics &statistics, SolverConfiguration &config);

  ~ComponentCache() {
    // debug_dump_data();
    for (auto &pentry : entry_base_) {
//      if (pentry != nullptr)
//        delete pentry;
      delete pentry;  // Do NOT need the check as deleting nullptr has no effect
    }
  }

  void init(Component &super_comp, bool quiet = false);

  // compute the size in bytes of the component cache from scratch
  // the value is stored in bytes_memory_usage_
  uint64_t compute_byte_size_infrastructure();
  const CacheableComponent &entry_const(const CacheEntryID id) const {
    assert(entry_base_.size() > id);  // Verify the ID is value for the size.
    assert(entry_base_[id] != nullptr);
    return *entry_base_[id];
  }
  CacheableComponent &entry(const CacheEntryID id) {
    return const_cast<CacheableComponent&>(entry_const(id));
```

(continues on next page)

(continued from previous page)

```
  }
//  /**
//   * Uses a cached component's CacheEntryID to extract the components
//   * cached information.
//   *
//   * @param comp Component to be extracted from the cache.
//   * @return Cache entry for the specified component.
//   */
//  CacheableComponent &entry(const Component& comp) {
//    return entry(comp.id());
//  }
  bool hasEntry(CacheEntryID id) const {
    assert(entry_base_.size() > id);
    return static_cast<bool>(entry_base_[id]);
  }
  inline void removeFromHashTable(CacheEntryID id);

  // we delete the Component with ID id
  // and all its descendants from the cache
  inline void cleanPollutionsInvolving(CacheEntryID id);

  // creates a CCacheEntry in the entry base
  // which contains a packed copy of comp
  // returns the id of the entry created
  // stores in the entry the position of
  // comp which is a part of the component stack
  inline CacheEntryID storeAsEntry(CacheableComponent &ccomp,
                                   CacheEntryID super_comp_id);
  bool manageNewComponent(StackLevel &top, CacheableComponent &packed_comp,
                          Component * unpacked_comp, CachedAssignment &cached_assn,
                          CacheEntryID &cache_entry_id) {
    statistics_.num_cache_look_ups_++;
    unsigned table_ofs =  packed_comp.hashkey() & table_size_mask_;

    CacheEntryID act_id = table_[table_ofs];
    while (act_id) {
      if (entry(act_id).equals(packed_comp)) {
        if (config_.verbose) {
          std::stringstream ss;
          ss << "\tCache hit for component #" << top.super_component() << " with␣
↪model count "
             << entry(act_id).model_count() << " for cache ID #" << act_id;
          PrintInColor(ss, COLOR_YELLOW);
        }
//        if (config_.perform_random_sampling_ && config_.perform_sample_caching()) {
        if (config_.perform_sample_caching()) {
          // ToDo Add support for multiple samples in cache.
          assert(config_.num_samples_to_cache_ == 1);
          VariableIndex num_vars = unpacked_comp->num_variables();
          VariableIndex bit_shift = CACHED_VARIABLE_LEN * num_vars;
          mpz_class actual_model_count = entry(act_id).model_count() >> bit_shift;
          top.includeSolution(actual_model_count);
          cached_assn.ProcessComponent(unpacked_comp, entry(act_id).model_count());
        } else {
          top.includeSolution(entry(act_id).model_count());
        }
        cache_entry_id = act_id;  // Store the cache ID
```

(continues on next page)

(continued from previous page)

```
        statistics_.incorporate_cache_hit(packed_comp);
        return true;
      }
      act_id = entry(act_id).next_bucket_element();
    }
    return false;
  }


  // unchecked erase of an entry from entry_base_
  void eraseEntry(CacheEntryID id) {
    statistics_.incorporate_cache_erase(*entry_base_[id]);
    delete entry_base_[id];
    entry_base_[id] = nullptr;
    free_entry_base_slots_.push_back(id);
  }

  // store the number in model_count as the model count of CacheEntryID id
  inline void storeValueOf(CacheEntryID id, const mpz_class &model_count);
  bool deleteEntries();

  // delete entries, keeping the descendants tree consistent
  inline void removeFromDescendantsTree(CacheEntryID id);

  // test function to ensure consistency of the descendant tree
  inline void test_descendantstree_consistency();

  void debug_dump_data();

private:
  void considerCacheResize() {
    if (entry_base_.size() > table_.size()) {
      reHashTable(2*table_.size());
    }
  }

  void reHashTable(unsigned long size) {
    table_.clear();
    table_.resize(size, 0);
    // we assert that table size is a power of 2
    // otherwise the table_size_mask_ doesn't work
    assert((table_.size() & (table_.size() - 1)) == 0);
    table_size_mask_ = table_.size() - 1;
    std::cout << "ZH - Rehash the table.\n";
    std::cout << "ts " << table_.size() << " " << table_size_mask_ << std::endl;
    unsigned collisions = 0;
    for (unsigned id = 2; id < entry_base_.size(); id++) {
      if (entry_base_[id] != nullptr) {
        entry_base_[id]->set_next_bucket_element(0);
        if (entry_base_[id]->modelCountFound()) {
          unsigned table_ofs = tableEntry(id);
          collisions += (table_[table_ofs] > 0 ? 1 : 0);
          entry_base_[id]->set_next_bucket_element(table_[table_ofs]);
          table_[table_ofs] = id;
        }
      }
    }
```

(continues on next page)

(continued from previous page)

```
    std::cout << "ZH - Number of collisions.\n" << "coll " << collisions << std::endl;
  }

  unsigned tableEntry(CacheEntryID id) {
    return entry(id).hashkey() & table_size_mask_;
  }

  void add_descendant(CacheEntryID compid, CacheEntryID descendantid) {
    assert(descendantid != entry_const(compid).first_descendant());
    entry(descendantid).set_next_sibling(entry_const(compid).first_descendant());
    entry(compid).set_first_descendant(descendantid);
  }

  void remove_firstdescendantOf(CacheEntryID compid) {
    CacheEntryID desc = entry(compid).first_descendant();
    if (desc != 0)
      entry(compid).set_first_descendant(entry(desc).next_sibling());
  }

  std::vector<CacheableComponent *> entry_base_;
  std::vector<CacheEntryID> free_entry_base_slots_;

  // the actual hash table
  // by means of which the cache is accessed
  std::vector<CacheEntryID> table_;

  unsigned table_size_mask_ = INT_MAX;

  DataAndStatistics &statistics_;

  unsigned long my_time_ = 0;

  SolverConfiguration &config_;
};


#include "component_cache.inc"


#endif /* COMPONENT_CACHE_H_ */
```

Includes

- `cached_assignment.h` (*File cached_assignment.h*)
- `component_cache.inc` (*File component_cache.inc*)
- `component_types/cacheable_component.h` (*File cacheable_component.h*)
- `component_types/component.h` (*File component.h*)
- `gmpxx.h`
- `solver_config.h` (*File solver_config.h*)
- `sstream`
- `stack.h` (*File stack.h*)

- statistics.h (*File statistics.h*)
- vector

Included By

- *File component_cache.cpp*
- *File component_management.h*

Classes

- *Class ComponentCache*

File component_cache.inc

Parent directory (src)

Contents

- *Definition (src/component_cache.inc)*
- *Included By*

Definition (src/component_cache.inc)

Program Listing for File component_cache.inc

Return to documentation for file (src/component_cache.inc)

```
#ifndef COMPONENT_CACHE_INL_H_
#define COMPONENT_CACHE_INL_H_

CacheEntryID ComponentCache::storeAsEntry(CacheableComponent &ccomp, CacheEntryID␣
↪super_comp_id) {
    CacheEntryID id;

    if (statistics_.cache_full())
        deleteEntries();

    assert(!statistics_.cache_full());

    ccomp.set_creation_time(my_time_++);

    if (free_entry_base_slots_.empty()) {
        if (entry_base_.capacity() == entry_base_.size()) {
            entry_base_.reserve(2 * entry_base_.size());
        }
        entry_base_.push_back(&ccomp);
        id = entry_base_.size() - 1;
    } else {
```

(continues on next page)

(continued from previous page)

```
        id = free_entry_base_slots_.back();
        assert(id < entry_base_.size());
        assert(entry_base_[id] == nullptr);
        free_entry_base_slots_.pop_back();
        entry_base_[id] = &ccomp;
    }

    entry(id).set_father(super_comp_id);
    add_descendant(super_comp_id, id);

    assert(hasEntry(id));
    assert(hasEntry(super_comp_id));

    statistics_.incorporate_cache_store(ccomp);

  #ifdef DEBUG
      for (unsigned u = 2; u < entry_base_.size(); u++)
            if (entry_base_[u] != nullptr) {
              assert(entry_base_[u]->father() != id);
              assert(entry_base_[u]->first_descendant() != id);
              assert(entry_base_[u]->next_sibling() != id);
            }
  #endif
    return id;
}

void ComponentCache::cleanPollutionsInvolving(CacheEntryID id) {
  CacheEntryID father = entry(id).father();
  if (entry(father).first_descendant() == id) {
    entry(father).set_first_descendant(entry(id).next_sibling());
  } else {
    CacheEntryID act_sibl = entry(father).first_descendant();
    while (act_sibl) {
      CacheEntryID next_sibl = entry(act_sibl).next_sibling();
      if (next_sibl == id) {
        entry(act_sibl).set_next_sibling(entry(next_sibl).next_sibling());
        break;
      }
      act_sibl = next_sibl;
    }
  }
  CacheEntryID next_child = entry(id).first_descendant();
  entry(id).set_first_descendant(0);
  while (next_child) {
    CacheEntryID act_child = next_child;
    next_child = entry(act_child).next_sibling();
    cleanPollutionsInvolving(act_child);
  }
  removeFromHashTable(id);
  eraseEntry(id);
}

void ComponentCache::removeFromHashTable(CacheEntryID id) {
  //assert(false);
  unsigned act_id = table_[tableEntry(id)];
  if (act_id == id) {
    table_[tableEntry(id)] = entry(act_id).next_bucket_element();
```

(continues on next page)

(continued from previous page)

```
  } else {
    while (act_id) {
      CacheEntryID next_id = entry(act_id).next_bucket_element();
      if (next_id == id) {
        entry(act_id).set_next_bucket_element(entry(next_id).next_bucket_element());
        break;
      }
      act_id = next_id;
    }
  }
//  CacheBucket *p_bucket = bucketOf(entry(id));
//  if (p_bucket)
//    for (auto it = p_bucket->begin(); it != p_bucket->end(); it++)
//      if (*it == id) {
//        *it = p_bucket->back();
//        p_bucket->pop_back();
//        break;
//      }
}

void ComponentCache::removeFromDescendantsTree(CacheEntryID id) {
  assert(hasEntry(id));
  // we need a father for this all to work
  assert(entry(id).father());
  assert(hasEntry(entry(id).father()));
  // two steps
  // 1. remove id from the siblings list
  CacheEntryID father = entry(id).father();
  if (entry(father).first_descendant() == id) {
    entry(father).set_first_descendant(entry(id).next_sibling());
  } else {
    CacheEntryID act_sibl = entry(father).first_descendant();
    while (act_sibl) {
      CacheEntryID next_sibl = entry(act_sibl).next_sibling();
      if (next_sibl == id) {
        entry(act_sibl).set_next_sibling(entry(next_sibl).next_sibling());
        break;
      }
      act_sibl = next_sibl;
    }
  }

  // 2. add the children of this one as
  //    siblings to the current siblings
  CacheEntryID act_child = entry(id).first_descendant();
  while (act_child) {
    CacheEntryID next_child = entry(act_child).next_sibling();
    entry(act_child).set_father(father);
    entry(act_child).set_next_sibling(entry(father).first_descendant());
    entry(father).set_first_descendant(act_child);
    act_child = next_child;
  }
}

void ComponentCache::storeValueOf(CacheEntryID id, const mpz_class &model_count) {
  considerCacheResize();
  unsigned table_ofs = tableEntry(id);
```

(continues on next page)

(continued from previous page)

```
  // when storing the new model count the size of the model count
  // and hence that of the component will change
  statistics_.sum_bytes_cached_components_ -= entry(id).SizeInBytes();
  statistics_.overall_bytes_components_stored_ -= entry(id).SizeInBytes();

  statistics_.sys_overhead_sum_bytes_cached_components_ -= entry(id).sys_overhead_
↪SizeInBytes();
  statistics_.sys_overhead_overall_bytes_components_stored_ -= entry(id).sys_overhead_
↪SizeInBytes();

  entry(id).set_model_count(model_count, my_time_);
  entry(id).set_creation_time(my_time_);

  entry(id).set_next_bucket_element(table_[table_ofs]);
  table_[table_ofs] = id;

  statistics_.sum_bytes_cached_components_ += entry(id).SizeInBytes();
  statistics_.overall_bytes_components_stored_ += entry(id).SizeInBytes();

  statistics_.sys_overhead_sum_bytes_cached_components_ += entry(id).sys_overhead_
↪SizeInBytes();
  statistics_.sys_overhead_overall_bytes_components_stored_ += entry(id).sys_overhead_
↪SizeInBytes();
}




#endif /* COMPONENT_CACHE_INL_H_ */
```

Included By

- *File component_cache.h*

File component_management.cpp

Parent directory (`src`)

Contents

- *Definition (`src/component_management.cpp`)*
- *Includes*
- *Functions*

Definition (`src/component_management.cpp`)

Program Listing for File component_management.cpp

Return to documentation for file (src/component_management.cpp)

```
#include <algorithm>
#include "component_management.h"

void ComponentManager::initialize(LiteralIndexedVector<Literal> & literals,
                                  std::vector<LiteralID> &lit_pool, bool quiet) {
  ana_.initialize(literals, lit_pool);
  // BEGIN CACHE INIT
  CacheableComponent::adjustPackSize(ana_.max_variable_id(), ana_.max_clause_id());
  // Prevent a memory leak
  for (auto &i : component_stack_)
    delete i;
  component_stack_.clear();
  component_stack_.reserve(ana_.max_variable_id() + 2);
  component_stack_.push_back(new Component());
  component_stack_.push_back(new Component());
  assert(component_stack_.size() == 2);
  component_stack_.back()->createAsDummyComponent(ana_.max_variable_id(),
      ana_.max_clause_id());
  if (config_.perform_random_sampling_)
    cached_vars_.clear();

  cache_.init(*component_stack_.back(), quiet);
}


void ComponentManager::removeAllCachePollutionsOf(StackLevel &top) {
  // all processed components are found in
  // [top.currentRemainingComponent(), component_stack_.size())
  // first, remove the list of descendants from the father
  assert(top.remaining_components_ofs() <= component_stack_.size());
  assert(top.super_component() != 0);
  assert(cache_.hasEntry(super_component(top).id()));

  if (top.remaining_components_ofs() == component_stack_.size())
    return;

  for (unsigned u = top.remaining_components_ofs(); u < component_stack_.size();
      u++) {
    assert(cache_.hasEntry(component_stack_[u]->id()));
    cache_.cleanPollutionsInvolving(component_stack_[u]->id());
  }

#ifdef DEBUG
  cache_.test_descendantstree_consistency();
#endif
}

std::vector<VariableIndex> ComponentManager::buildFreedVariableList(const StackLevel &
↪ top,
                                                                    const std::vector<LiteralID> &
↪literal_stack) {
  Component * descendant_comp, *ref_comp = component_stack_[top.super_component()];
  const VariableIndex original_var_count = ref_comp->num_variables();
```

(continues on next page)

(continued from previous page)

```
  VariableIndex num_unused_vars = original_var_count;
  std::vector<bool> var_in_descendants(original_var_count, false);
  std::vector<ComponentVarAndCls> descendant_vars, ref_vars = ref_comp->getData();

  for (unsigned comp_id=top.remaining_components_ofs();
       comp_id < top.unprocessed_components_end(); comp_id++) {
    descendant_comp = component_stack_[comp_id];
    descendant_vars = descendant_comp->getData();
    UpdateVarDescedantsList(ref_vars, ref_comp->num_variables(), var_in_descendants,
                            descendant_vars, descendant_comp->num_variables());
    num_unused_vars -= descendant_comp->num_variables();
  }

  // Build a list of all literals assigned in the last round
  const long list_stack_ofs = top.literal_stack_ofs();
  VariableIndex num_lits_assigned = literal_stack.size() - list_stack_ofs;

  std::vector<ComponentVarAndCls> assigned_stack_vars;
  assigned_stack_vars.reserve(num_lits_assigned);
  for (VariableIndex stack_i = 0; stack_i < num_lits_assigned; stack_i++)
    assigned_stack_vars.push_back(literal_stack[list_stack_ofs + stack_i].var());
  // Sort the vector and exclude the assigned literals
  sort(assigned_stack_vars.begin(), assigned_stack_vars.end());
  num_unused_vars -= UpdateVarDescedantsList(ref_vars, ref_comp->num_variables(),
                                             var_in_descendants, assigned_stack_vars,
                                             num_lits_assigned);

  // Skip the cached variables
  num_unused_vars -= UpdateVarDescedantsList(ref_vars, ref_comp->num_variables(),
                                             var_in_descendants,
                                             cached_vars_, cached_vars_.size());

  // If a variable does not appear in the descendant nor in the cache and is not on
  // top of the literal stack, it was freed.
  std::vector<VariableIndex> freed_vars;
  freed_vars.reserve(num_unused_vars);
  for (VariableIndex ref_i = 0; ref_i < ref_comp->num_variables(); ref_i++) {
    if (!var_in_descendants[ref_i])
      freed_vars.push_back(ref_vars[ref_i]);
  }
  assert(num_unused_vars == freed_vars.size());
  return freed_vars;
}


VariableIndex UpdateVarDescedantsList(const std::vector<ComponentVarAndCls> &ref_vars,
                                      VariableIndex ref_num_vars,
                                      std::vector<bool> &varInDescendants,
                                      std::vector<ComponentVarAndCls> descendant_vars,
                                      VariableIndex desc_num_vars) {
  if (desc_num_vars == 0)
    return 0;
  VariableIndex varsRemoved = 0;
  VariableIndex ref_i, desc_i = 0;
  for (ref_i = 0; ref_i < ref_num_vars; ref_i++) {
    // Reference list is always a superset and since both are sorted, go to the next
```

(continues on next page)

(continued from previous page)

```
    if (ref_vars[ref_i] < descendant_vars[desc_i])
      continue;
    if (ref_vars[ref_i] > descendant_vars[desc_i])
      desc_i++;
    if (desc_i >= desc_num_vars)
      break;

    // If the descendant and reference match, the var is present
    if (ref_vars[ref_i] == descendant_vars[desc_i]) {
      // DebugZH - A variable should only be in a single descedent
      assert(!varInDescendants[ref_i]);
      varInDescendants[ref_i] = true;
      varsRemoved++;
    }
  }
  return varsRemoved;
}
```

Includes

- `algorithm`
- `component_management.h` (*File component_management.h*)

Functions

- *Function UpdateVarDescedantsList(const std::vector<ComponentVarAndCls>&, VariableIndex, std::vector<bool>&, std::vector<ComponentVarAndCls>, VariableIndex)*

File component_management.h

Parent directory (`src`)

Contents

- *Definition (`src/component_management.h`)*
- *Includes*
- *Included By*
- *Classes*
- *Functions*
- *Typedefs*

Definition (`src/component_management.h`)

Program Listing for File component_management.h

Return to documentation for file (src/component_management.h)

```
#ifndef COMPONENT_MANAGEMENT_H_
#define COMPONENT_MANAGEMENT_H_

#include <gmpxx.h>

#include <utility>
#include <algorithm>
#include <string>
#include <vector>

#include "component_types/component.h"
#include "component_cache.h"
#include "alt_component_analyzer.h"
#include "containers.h"
#include "stack.h"
#include "solver_config.h"
#include "model_sampler.h"
//#include "top_tree_sampler.h"

typedef AltComponentAnalyzer ComponentAnalyzer;

class ComponentManager {
 public:
  ComponentManager(SolverConfiguration &config, DataAndStatistics &statistics,
                   LiteralIndexedVector<TriValue> & lit_values) :
    config_(config), /*statistics_(statistics),*/ cache_(statistics, config),
    ana_(statistics, lit_values) {
  }
  void initialize(LiteralIndexedVector<Literal> & literals,
                  std::vector<LiteralID> &lit_pool, bool quiet = false);

  unsigned scoreOf(VariableIndex v) {
    return ana_.scoreOf(v);
  }
  void cacheModelCountOf(VariableIndex stack_comp_id, const mpz_class &value) {
    if (config_.perform_component_caching)
      cache_.storeValueOf(component_stack_[stack_comp_id]->id(), value);
  }
  void cacheModelCountAndAssignment(const VariableIndex stack_comp_id, const mpz_
↪class &value,
                                    const SampleAssignment &assn, const Component &
↪component) {
    mpz_class combined_value = 0;
    mpz_class temp_value = 0;
    // If the component is UNSAT do not push any assignment
    if (value > 0) {
      VariableIndex last_shift_len = 0, offset_amount = 0;
      uint64_t cached_word = 0;
      const VariableIndex MAX_OFFSET_AMOUNT = 28;

      std::vector<ComponentVarAndCls> comp_data = component.getData();
      for (VariableIndex i = 0; i < component.num_variables(); i++) {
        AssignmentEncoding var_assn = assn.var_assignment(comp_data[i]);
```

(continues on next page)

(continued from previous page)

```
        // Make sure the assignment is valid
        assert((var_assn != ASSN_U && !assn.IsVarEmancipated(comp_data[i]))
               || (var_assn == ASSN_U && assn.IsVarEmancipated(comp_data[i])));

        cached_word += var_assn << (offset_amount);
        if (offset_amount == MAX_OFFSET_AMOUNT) {
          mpz_import(temp_value.get_mpz_t(), 1, 1, sizeof(cached_word), 0, 0, &cached_
↪word);
          combined_value += temp_value << last_shift_len;
          last_shift_len = CACHED_VARIABLE_LEN * (i + 1);
          offset_amount = 0;
          cached_word = 0;
        } else {
          offset_amount += CACHED_VARIABLE_LEN;
        }
      }
      if (offset_amount > 0) {
        mpz_import(temp_value.get_mpz_t(), 1, 1, sizeof(cached_word), 0, 0, &cached_
↪word);
        combined_value += temp_value << last_shift_len;
      }
      combined_value += value << (CACHED_VARIABLE_LEN * component.num_variables());
    }
    cacheModelCountOf(stack_comp_id, combined_value);
  }
  Component & superComponentOf(StackLevel &lev) {
    assert(component_stack_.size() > lev.super_component());
    return *component_stack_[lev.super_component()];
  }
  const Component & super_component(StackLevel &lev) const {
    assert(component_stack_.size() > lev.super_component());
    return *component_stack_[lev.super_component()];
  }
  VariableIndex component_stack_size() const {
    return static_cast<VariableIndex>(component_stack_.size());
  }

  void cleanRemainingComponentsOf(StackLevel &top) {
    while (component_stack_.size() > top.remaining_components_ofs()) {
      if (cache_.hasEntry(component_stack_.back()->id()))
        cache_.entry(component_stack_.back()->id()).set_deletable();
      delete component_stack_.back();
      component_stack_.pop_back();
    }
    assert(top.remaining_components_ofs() <= component_stack_.size());
  }
  inline const Component &component(unsigned long comp_idx) const {
    return *component_stack_[comp_idx];
  }
//  Component & currentRemainingComponentOf(StackLevel &top) {
//    assert(component_stack_.size() > top.currentRemainingComponent());
//    return *component_stack_[top.currentRemainingComponent()];
//  }
  inline bool findNextRemainingComponentOf(StackLevel &top,
                                           const std::vector<LiteralID> &literal_
↪stack,
                                           SamplesManager &samples);
```

(continues on next page)

(continued from previous page)

```
  inline void recordRemainingCompsFor(StackLevel &top,
                                      const std::vector<LiteralID> &literal_stack);
  inline void buildResidualComponent(StackLevel &top);
  inline Component* top_stack() {
    return component_stack_.back();
  }

  inline void sortComponentStackRange(unsigned long start, unsigned long end);

  void gatherStatistics() {
//     statistics_.cache_bytes_memory_usage_ =
//       cache_.recompute_bytes_memory_usage();
    cache_.compute_byte_size_infrastructure();
  }

  void removeAllCachePollutionsOf(StackLevel &top);
  std::vector<VariableIndex> buildFreedVariableList(const StackLevel & top,
                                                    const std::vector<LiteralID> &
↪literal_stack);

 private:
  std::vector<VariableIndex> cached_vars_;
  std::vector<CacheEntryID> cached_comp_ids_;

  SolverConfiguration &config_;

//  DataAndStatistics &statistics_;

  std::vector<Component *> component_stack_;
  ComponentCache cache_;
  ComponentAnalyzer ana_;
};

void ComponentManager::sortComponentStackRange(unsigned long start, unsigned long␣
↪end) {
  assert(start <= end);
  // sort the remaining components for processing
  for (unsigned long i = start; i < end; i++)
    for (unsigned long j = i + 1; j < end; j++) {
      if (component_stack_[i]->num_variables()
          < component_stack_[j]->num_variables())
        std::swap(component_stack_[i], component_stack_[j]);
    }
}

bool ComponentManager::findNextRemainingComponentOf(StackLevel &top,
                                                    const std::vector<LiteralID> &
↪literal_stack,
                                                    SamplesManager &samples) {
  // record Remaining Components if there are none!
  if (component_stack_.size() <= top.remaining_components_ofs())
    recordRemainingCompsFor(top, literal_stack);

  assert(!top.branch_found_unsat());
  if (top.hasUnprocessedComponents()) {
    // Perform reservoir sampling if the model count is greater than zero
    if (config_.perform_random_sampling_ && top.getActiveModelCount() > 0) {
```

(continues on next page)

(continued from previous page)

```
      if (!top.isComponentSplit() || top.isFirstComponent()) {
        top.includeSolutionSampleMultiplier(top.getActiveModelCount());
        std::vector<VariableIndex> freed_vars = buildFreedVariableList(top, literal_
↪stack);
        top.addFreeVariables(freed_vars);
        top.addCachedCompIds(cached_comp_ids_);
      }
    }
    return true;
  }

  // if no component remains
  // make sure, at least that the current branch is considered SAT
  top.includeSolution(1);

  // Perform reservoir sampling if the model count is greater than zero
  if (config_.store_sampled_models()) {
    if (config_.verbose) {
      std::cout << "\t# Solutions Found: " << top.getActiveModelCount()
                << ". Total multiplied weight is "
                << top.getActiveModelCount() * top.getSamplerSolutionMultiplier()
                << std::endl;
    }
    SampleAssignment cached_sample(config_.num_samples_to_cache_);
    if (!top.cached_comp_ids().empty())
      cached_comp_ids_.insert(cached_comp_ids_.end(), top.cached_comp_ids().begin(),
                              top.cached_comp_ids().end());
    samples.reservoirSample(component_stack_[top.super_component()], literal_stack,
                            top.getActiveModelCount(), top.
↪getSamplerSolutionMultiplier(),
                            ana_, top.literal_stack_ofs(), top.emancipated_vars(),
                            cached_comp_ids_,  top.cached_assn(), cached_sample);
    if (config_.perform_sample_caching()) {
      top.ClearCachedAssn();
      top.set_cache_sample(cached_sample);
    }
    // DebugZH
//    assert(samples.VerifySolutions(statistics_.input_file_, true));
  }
  return false;
}
void ComponentManager::recordRemainingCompsFor(StackLevel &top,
                                               const std::vector<LiteralID> &literal_
↪stack) {
  Component & super_comp = superComponentOf(top);
  VariableIndex new_comps_start_ofs = component_stack_.size();  // Location to store␣
↪new comp if any

  // Clear the variables considered cached.
  cached_vars_.clear();
  cached_comp_ids_.clear();
  CachedAssignment cached_assn;

  // Initialize data structures for the component analyzer
  ana_.setupAnalysisContext(top, super_comp);

  // Go through each variable and checks its component to see if cached or new
```

(continues on next page)

(continued from previous page)

```
  for (auto vt = super_comp.varsBegin(); *vt != varsSENTINEL; vt++) {
    if (ana_.isUnseenAndActive(*vt) && ana_.exploreRemainingCompOf(*vt)) {
      Component *p_new_comp = ana_.makeComponentFromArcheType();
      auto *packed_comp = new CacheableComponent(ana_.getArchetype().current_comp_for_
↪caching_);
      CacheEntryID cache_entry_id;
      if (!cache_.manageNewComponent(top, *packed_comp, p_new_comp, cached_assn,␣
↪cache_entry_id)) {
        component_stack_.push_back(p_new_comp);
        p_new_comp->set_id(cache_.storeAsEntry(*packed_comp, super_comp.id()));
      } else {
        if (config_.perform_random_sampling_) {
          // Store any cached vars in case of a split.
          std::vector<ComponentVarAndCls> cached_comp_data = p_new_comp->getData();
          VariableIndex comp_i = 0;
          // Store the variables in the component for proper processing of␣
↪emancipated variables
          while (cached_comp_data[comp_i] != varsSENTINEL) {
            cached_vars_.push_back(cached_comp_data[comp_i]);
            comp_i++;
          }
          cached_comp_ids_.push_back(cache_entry_id);
        }
        // Delete component as it was found in the cache
        delete packed_comp;
        delete p_new_comp;
      }
    }
  }

  top.set_unprocessed_components_end(component_stack_.size());
  sortComponentStackRange(new_comps_start_ofs, component_stack_.size());
  if (config_.perform_random_sampling_) {
    long split_width = component_stack_.size() - new_comps_start_ofs;
    if (split_width > 1 || (split_width == 1 && !cached_assn.empty())) {
      top.setAsComponentSplit();
      if (config_.verbose) {
        std::stringstream ss;
        ss << "Component #" << (new_comps_start_ofs - 1) << " split into " << (split_
↪width)
           << " pieces at depth " << StackLevel::componentSplitDepth() << ".";
        PrintInColor(ss, COLOR_CYAN);
      }
    }
    if (!cached_assn.empty())
      top.set_cached_assn(cached_assn);
  }
  // For simplicity, sort the cached variable list
  if (cached_vars_.size() > 1) {
    sort(cached_vars_.begin(), cached_vars_.end());
    sort(cached_comp_ids_.begin(), cached_comp_ids_.end());
  }
}


void ComponentManager::buildResidualComponent(StackLevel &top) {
  Component & super_comp = superComponentOf(top);
```

(continues on next page)

(continued from previous page)

```
  std::vector<VariableIndex> comp_vars;
  std::vector<ClauseIndex> comp_cls;

  // Initialize data structures for the component analyzer
  ana_.setupAnalysisContext(top, super_comp);

  // Go through all components to be spawned and build a list of vars and comp
  for (auto vt = super_comp.varsBegin(); *vt != varsSENTINEL; vt++) {
    if (ana_.isUnseenAndActive(*vt) && ana_.exploreRemainingCompOf(*vt)) {
      Component *p_new_comp = ana_.makeComponentFromArcheType();
      auto comp_data = p_new_comp->getData();

      for (VariableIndex i = 0; i < p_new_comp->num_variables(); i++)
        comp_vars.push_back(comp_data[i]);
      for (ClauseIndex i = 0; i < p_new_comp->numLongClauses(); i++) {
        ClauseIndex cls = comp_data[p_new_comp->clauses_ofs() + i];
        comp_cls.push_back(cls);
      }
      delete p_new_comp;
    }
  }
  delete component_stack_.back();
  component_stack_.pop_back();

  // Sort the lists as components are build from only sorted lists
  std::sort(comp_vars.begin(), comp_vars.end());
  std::sort(comp_cls.begin(), comp_cls.end());

  auto * initial_component = new Component();
  for (auto var : comp_vars)
    initial_component->addVar(var);
  initial_component->closeVariableData();
  for (auto cls : comp_cls)
    initial_component->addCl(cls);
  initial_component->closeClauseData();
  component_stack_.push_back(initial_component);
}
VariableIndex UpdateVarDescedantsList(const std::vector<ComponentVarAndCls> &ref_vars,
                                      VariableIndex ref_num_vars,
                                      std::vector<bool> &varInDescendants,
                                      std::vector<ComponentVarAndCls> descendant_vars,
                                      VariableIndex desc_num_vars);
//bool varInVector(long var_num, std::vector<unsigned int> vec);

#endif /* COMPONENT_MANAGEMENT_H_ */
```

Includes

- `algorithm`
- `alt_component_analyzer.h` (*File alt_component_analyzer.h*)
- `component_cache.h` (*File component_cache.h*)
- `component_types/component.h` (*File component.h*)

- `containers.h` (*File containers.h*)
- `gmpxx.h`
- `model_sampler.h` (*File model_sampler.h*)
- `solver_config.h` (*File solver_config.h*)
- `stack.h` (*File stack.h*)
- `string`
- `utility`
- `vector`

Included By

- *File component_management.cpp*
- *File solver.h*

Classes

- *Class ComponentManager*

Functions

- *Function UpdateVarDescedantsList(const std::vector<ComponentVarAndCls>&, VariableIndex, std::vector<bool>&, std::vector<ComponentVarAndCls>, VariableIndex)*

Typedefs

- *Typedef ComponentAnalyzer*

File containers.h

Parent directory (`src`)

Contents

- *Definition (`src/containers.h`)*
- *Includes*
- *Included By*
- *Classes*

Definition (`src/containers.h`)

Program Listing for File containers.h

Return to documentation for file (`src/containers.h`)

```
#ifndef CONTAINERS_H_
#define CONTAINERS_H_

#include <vector>

#include "structures.h"

template<class _T>
/***
 * @tparam _T Type of element to store in the vector.
 */
class LiteralIndexedVector: protected std::vector<_T> {
 public:
  explicit LiteralIndexedVector(VariableIndex size = 0) :
      std::vector<_T>(size * 2) {
  }
  LiteralIndexedVector(VariableIndex size, const typename std::vector<_T>::value_type&
↪ __value)
      : std::vector<_T>(size * 2, __value) {
  }
  inline _T &operator[](const LiteralID lit) {
    return *(std::vector<_T>::begin() + lit.raw());
  }

  inline const _T &operator[](const LiteralID &lit) const {
    return *(std::vector<_T>::begin() + lit.raw());
  }
  inline typename std::vector<_T>::iterator begin() {
    return std::vector<_T>::begin() + 2;
  }
  void resize(VariableIndex _size) {
    std::vector<_T>::resize(_size * 2);
  }
  void resize(VariableIndex _size, const typename std::vector<_T>::value_type& _
↪value) {
    std::vector<_T>::resize(_size * 2, _value);
  }

  void reserve(VariableIndex _size) {
    std::vector<_T>::reserve(_size * 2);
  }
  LiteralID end_lit() {
    return LiteralID(size() / 2, false);
  }

  // Methods reused from the imported class
  using std::vector<_T>::end;
  using std::vector<_T>::size;
  using std::vector<_T>::clear;
  using std::vector<_T>::push_back;
};
```

(continues on next page)

(continued from previous page)

```
#endif /* CONTAINERS_H_ */
```

Includes

- `structures.h` (*File structures.h*)
- `vector`

Included By

- *File alt_component_analyzer.h*
- *File component_management.h*
- *File instance.h*

Classes

- *Template Class LiteralIndexedVector*

File difference_packed_component.h

Parent directory (`src/component_types`)

Contents

- *Definition (`src/component_types/difference_packed_component.h`)*
- *Includes*
- *Included By*
- *Classes*

Definition (`src/component_types/difference_packed_component.h`)

Program Listing for File difference_packed_component.h

Return to documentation for file (`src/component_types/difference_packed_component.h`)

```
#ifndef DIFFERENCE_PACKED_COMPONENT_H_
#define DIFFERENCE_PACKED_COMPONENT_H_

#include <math.h>

#include "base_packed_component.h"
#include "component.h"
```

(continues on next page)

(continued from previous page)

```
class DifferencePackedComponent:public BasePackedComponent {
 public:
  DifferencePackedComponent() = default;

  inline explicit DifferencePackedComponent(Component &rComp);

  unsigned num_variables() const {
    uint64_t *p = reinterpret_cast<uint64_t *>(data_);
    // ToDo Thurley is doing something suspect here - Types dont have any casting
    return (*p >> bits_of_data_size()) & (uint64_t) variable_mask();
  }

  unsigned data_size() const {
    return *data_ & _data_size_mask;
  }

  unsigned data_only_byte_size() const {
    return data_size()* sizeof(unsigned);
  }

  unsigned raw_data_byte_size() const {
    return data_size()* sizeof(unsigned) + model_count_.get_mpz_t()->_mp_alloc *␣
↪sizeof(mp_limb_t);
  }

  // raw data size with the overhead
  // for the supposed 16byte alignment of malloc
  unsigned sys_overhead_raw_data_byte_size() const {
    unsigned ds = data_size()* sizeof(unsigned);
    unsigned ms = model_count_.get_mpz_t()->_mp_alloc * sizeof(mp_limb_t);
    //      unsigned mask = 0xfffffff8;
    //      return (ds & mask) + ((ds & 7)?8:0)
    //           +(ms & mask) + ((ms & 7)?8:0);
    unsigned mask = 0xfffffff0;
    return (ds & mask) + ((ds & 15)?16:0) +(ms & mask) + ((ms & 15)?16:0);
  }

  bool equals(const DifferencePackedComponent &comp) const {
    if (hashkey_ != comp.hashkey())
      return false;
    unsigned* p = data_;
    unsigned* r = comp.data_;
    while (p != data_ + data_size()) {
      if (*(p++) != *(r++))
        return false;
    }
    return true;
  }

 private:
};



DifferencePackedComponent::DifferencePackedComponent(Component &rComp) {
  unsigned max_var_diff = 0;
  unsigned hashkey_vars = *rComp.varsBegin();
```

(continues on next page)

(continued from previous page)

```
  for (auto it = rComp.varsBegin() + 1; *it != varsSENTINEL; it++) {
    hashkey_vars = (hashkey_vars * 3) + *it;
    if ((*it - *(it - 1)) - 1 > max_var_diff)
      max_var_diff = (*it - *(it - 1)) - 1;
  }

  unsigned hashkey_clauses = *rComp.clsBegin();
  unsigned max_clause_diff = 0;
  if (*rComp.clsBegin()) {
    for (auto jt = rComp.clsBegin() + 1; *jt != clsSENTINEL; jt++) {
      hashkey_clauses = hashkey_clauses*3 + *jt;
      if (*jt - *(jt - 1) - 1 > max_clause_diff)
        max_clause_diff = *jt - *(jt - 1) - 1;
    }
  }

  hashkey_ = hashkey_vars + (hashkey_clauses << 11) + (hashkey_clauses >> 23);

  //VERIFIED the definition of bits_per_var_diff and bits_per_clause_diff
  unsigned bits_per_var_diff = log2(max_var_diff) + 1;
  unsigned bits_per_clause_diff = log2(max_clause_diff) + 1;

  assert(bits_per_var_diff <= 31);
  assert(bits_per_clause_diff <= 31);

  unsigned data_size_vars = bits_of_data_size() + 2*bits_per_variable() + 5;

  data_size_vars += (rComp.num_variables() - 1) * bits_per_var_diff;

  unsigned data_size_clauses = 0;
  if (*rComp.clsBegin())
    data_size_clauses += bits_per_clause() + 5
       + (rComp.numLongClauses() - 1) * bits_per_clause_diff;

  unsigned data_size = (data_size_vars + data_size_clauses)/bits_per_block();
    data_size+=  ((data_size_vars + data_size_clauses) % bits_per_block())? 1 : 0;

  data_ = new unsigned[data_size];

  assert((data_size >> bits_of_data_size()) == 0);
  BitStuffer<unsigned> bs(data_);

  bs.stuff(data_size, bits_of_data_size());
  bs.stuff(rComp.num_variables(), bits_per_variable());
  bs.stuff(bits_per_var_diff, 5);
  bs.stuff(*rComp.varsBegin(), bits_per_variable());

  if (bits_per_var_diff) {
    for (auto it = rComp.varsBegin() + 1; *it != varsSENTINEL; it++)
      bs.stuff(*it - *(it - 1) - 1, bits_per_var_diff);
  }

  if (*rComp.clsBegin()) {
    bs.stuff(bits_per_clause_diff, 5);
    bs.stuff(*rComp.clsBegin(), bits_per_clause());
    if (bits_per_clause_diff) {
      for (auto jt = rComp.clsBegin() + 1; *jt != clsSENTINEL; jt++)
```

(continues on next page)

(continued from previous page)

```
        bs.stuff(*jt - *(jt - 1) - 1, bits_per_clause_diff);
    }
  }

  // to check wheter the "END" block of bits_per_clause()
  // many zeros fits into the current
//  bs.end_check(bits_per_clause());
  // this will tell us if we computed the data_size
  // correctly
  bs.assert_size(data_size);
}



#endif /* DIFFERENCE_PACKED_COMPONENT_H_ */
```

Includes

- `base_packed_component.h` (*File base_packed_component.h*)
- `component.h` (*File base_packed_component.h*)
- `math.h`

Included By

- *File cacheable_component.h*

Classes

- *Class DifferencePackedComponent*

File instance.cpp

Parent directory (`src`)

Contents

- *Definition (`src/instance.cpp`)*
- *Includes*

Definition (`src/instance.cpp`)

Program Listing for File instance.cpp

Return to documentation for file (`src/instance.cpp`)

```cpp
#include <sys/stat.h>
#include <algorithm>
#include <fstream>
#include <string>
#include <vector>

#include "instance.h"
#include "model_sampler.h"

void Instance::cleanClause(ClauseOfs cl_ofs) {
  bool satisfied = false;
  for (auto it = beginOf(cl_ofs); *it != SENTINEL_LIT; it++)
    if (isSatisfied(*it)) {
      satisfied = true;
      break;
    }
  // mark the clause as empty if satisfied
  if (satisfied) {
    *beginOf(cl_ofs) = SENTINEL_LIT;
    return;
  }
  auto jt = beginOf(cl_ofs);
  auto it = beginOf(cl_ofs);
  // from now, all inactive literals are resolved
  for (; *it != SENTINEL_LIT; it++, jt++) {
    while (*jt != SENTINEL_LIT && !isActive(*jt))
      jt++;
    *it = *jt;
    if (*jt == SENTINEL_LIT)
      break;
  }
  unsigned length = it - beginOf(cl_ofs);
  // if it has become a unit clause, it should have already been asserted
  if (length == 1) {
    *beginOf(cl_ofs) = SENTINEL_LIT;
    // if it has become binary, transform it to binary and delete it
  } else if (length == 2) {
    addBinaryClause(*beginOf(cl_ofs), *(beginOf(cl_ofs) + 1));
    *beginOf(cl_ofs) = SENTINEL_LIT;
  }
}

void Instance::compactClauses() {
  std::vector<ClauseOfs> clause_ofs;
  clause_ofs.reserve(statistics_.num_long_clauses_);

  // clear watch links and occurrence lists
  for (auto it_lit = literal_pool_.begin(); it_lit != literal_pool_.end();
       it_lit++) {
    if (*it_lit == SENTINEL_LIT) {
      if (it_lit + 1 == literal_pool_.end())
        break;
      it_lit += ClauseHeader::overheadInLits();
      clause_ofs.push_back(1 + it_lit - literal_pool_.begin());
    }
  }
```

(continues on next page)

(continued from previous page)

```
  for (auto ofs : clause_ofs)
    cleanClause(ofs);

  for (auto &l : literals_)
    l.resetWatchList();

  occurrence_lists_.clear();
  occurrence_lists_.resize(variables_.size());

  std::vector<LiteralID> tmp_pool = literal_pool_;
  literal_pool_.clear();
  literal_pool_.push_back(SENTINEL_LIT);
  ClauseOfs new_ofs;
  unsigned num_clauses = 0;
  for (auto ofs : clause_ofs) {
    auto it = (tmp_pool.begin() + ofs);
    if (*it != SENTINEL_LIT) {
      for (unsigned i = 0; i < ClauseHeader::overheadInLits(); i++)
        literal_pool_.emplace_back(0);
      new_ofs = literal_pool_.size();
      literal(*it).addWatchLinkTo(new_ofs);
      literal(*(it + 1)).addWatchLinkTo(new_ofs);
      num_clauses++;
      for (; *it != SENTINEL_LIT; it++) {
        literal_pool_.push_back(*it);
        occurrence_lists_[*it].push_back(new_ofs);
      }
      literal_pool_.push_back(SENTINEL_LIT);
    }
  }

  std::vector<LiteralID> tmp_bin;
  unsigned bin_links = 0;
  for (auto &l : literals_) {
    tmp_bin.clear();
    for (auto it = l.binary_links_.begin(); *it != SENTINEL_LIT; it++)
      if (isActive(*it))
        tmp_bin.push_back(*it);
    bin_links += tmp_bin.size();
    tmp_bin.push_back(SENTINEL_LIT);
    l.binary_links_ = tmp_bin;
  }
  statistics_.num_long_clauses_ = num_clauses;
  statistics_.num_binary_clauses_ = bin_links >> 1;
}

void Instance::compactVariables() {
  std::vector<unsigned> var_map(variables_.size(), 0);
  unsigned last_ofs = 0;
  unsigned num_isolated = 0;  // Emancipated/freed variables no longer in any clause
  LiteralIndexedVector<std::vector<LiteralID> > _tmp_bin_links(1);
  LiteralIndexedVector<TriValue> _tmp_values = literal_values_;

  for (auto l : literals_)
    _tmp_bin_links.push_back(l.binary_links_);

  assert(_tmp_bin_links.size() == literals_.size());
```

(continues on next page)

(continued from previous page)

```
  for (unsigned v = 1; v < variables_.size(); v++)
    if (isActive(v)) {
      if (isolated(v)) {
        std::cout << "Variable " << v << " is isolated." << std::endl;
        num_isolated++;
        continue;
      }
      last_ofs++;
      var_map[v] = last_ofs;
    }

  variables_.clear();
  variables_.resize(last_ofs + 1);
  occurrence_lists_.clear();
  occurrence_lists_.resize(variables_.size());
  literals_.clear();
  literals_.resize(variables_.size());
  literal_values_.clear();
  literal_values_.resize(variables_.size(), X_TRI);

  unsigned bin_links = 0;
  LiteralID newlit;
  for (auto l = LiteralID(0, false); l != _tmp_bin_links.end_lit(); l.inc()) {
    if (var_map[l.var()] != 0) {
      newlit = LiteralID(var_map[l.var()], l.sign());
      for (auto it = _tmp_bin_links[l].begin(); *it != SENTINEL_LIT; it++) {
        assert(var_map[it->var()] != 0);
        literals_[newlit].addBinLinkTo(
            LiteralID(var_map[it->var()], it->sign()));
      }
      bin_links += literals_[newlit].binary_links_.size() - 1;
    }
  }

  std::vector<ClauseOfs> clause_ofs;
  clause_ofs.reserve(statistics_.num_long_clauses_);
  // clear watch links and occurrence lists
  for (auto it_lit = literal_pool_.begin(); it_lit != literal_pool_.end();
      it_lit++) {
    if (*it_lit == SENTINEL_LIT) {
      if (it_lit + 1 == literal_pool_.end())
        break;
      it_lit += ClauseHeader::overheadInLits();
      clause_ofs.push_back(1 + it_lit - literal_pool_.begin());
    }
  }

  for (auto ofs : clause_ofs) {
    literal(LiteralID(var_map[beginOf(ofs)->var()], beginOf(ofs)->sign())).
↪addWatchLinkTo(
        ofs);
    literal(LiteralID(var_map[(beginOf(ofs) + 1)->var()],
            (beginOf(ofs) + 1)->sign())).addWatchLinkTo(ofs);
    for (auto it_lit = beginOf(ofs); *it_lit != SENTINEL_LIT; it_lit++) {
      *it_lit = LiteralID(var_map[it_lit->var()], it_lit->sign());
      occurrence_lists_[*it_lit].push_back(ofs);
    }
```

(continues on next page)

(continued from previous page)

```
  }

  literal_values_.clear();
  literal_values_.resize(variables_.size(), X_TRI);
  unit_clauses_.clear();

  statistics_.num_variables_ = variables_.size() - 1 + num_isolated;

  statistics_.num_used_variables_ = num_variables();
  statistics_.num_free_variables_ = num_isolated;
}

void Instance::compactConflictLiteralPool() {
  auto write_pos = conflict_clauses_begin();
  std::vector<ClauseOfs> tmp_conflict_clauses = conflict_clauses_;
  conflict_clauses_.clear();
  for (auto clause_ofs : tmp_conflict_clauses) {
    auto read_pos = beginOf(clause_ofs) - ClauseHeader::overheadInLits();
    for (unsigned i = 0; i < ClauseHeader::overheadInLits(); i++)
      *(write_pos++) = *(read_pos++);
    ClauseOfs new_ofs =  write_pos - literal_pool_.begin();
    conflict_clauses_.push_back(new_ofs);
    // first substitute antecedent if clause_ofs implied something
    if (isAntecedentOf(clause_ofs, *beginOf(clause_ofs)))
      var(*beginOf(clause_ofs)).ante = Antecedent(new_ofs);

    // now redo the watches
    literal(*beginOf(clause_ofs)).replaceWatchLinkTo(clause_ofs, new_ofs);
    literal(*(beginOf(clause_ofs)+1)).replaceWatchLinkTo(clause_ofs, new_ofs);
    // next, copy clause data
    assert(read_pos == beginOf(clause_ofs));
    while (*read_pos != SENTINEL_LIT)
      *(write_pos++) = *(read_pos++);
    *(write_pos++) = SENTINEL_LIT;
  }
  literal_pool_.erase(write_pos, literal_pool_.end());
}


//bool Instance::deleteConflictClauses() {
//  statistics_.times_conflict_clauses_cleaned_++;
//  std::vector<ClauseOfs> tmp_conflict_clauses = conflict_clauses_;
//  conflict_clauses_.clear();
//  std::vector<double> tmp_ratios;
//  double score, lifetime;
//  for (auto clause_ofs: tmp_conflict_clauses) {
//    score = getHeaderOf(clause_ofs).score();
//    lifetime = statistics_.num_conflicts_ - getHeaderOf(clause_ofs).creation_time();
//    tmp_ratios.push_back(score/lifetime/(getHeaderOf(clause_ofs).length()));
//  }
//  std::vector<double> tmp_ratiosB = tmp_ratios;
//
//  sort(tmp_ratiosB.begin(), tmp_ratiosB.end());
//
//  double cutoff = tmp_ratiosB[tmp_ratiosB.size()/2];
//
//  for (unsigned i = 0; i < tmp_conflict_clauses.size(); i++) {
```

(continues on next page)

(continued from previous page)

```
//    if (tmp_ratios[i] < cutoff) {
//      if (!markClauseDeleted(tmp_conflict_clauses[i]))
//        conflict_clauses_.push_back(tmp_conflict_clauses[i]);
//    } else
//      conflict_clauses_.push_back(tmp_conflict_clauses[i]);
//  }
//  return true;
//}

bool Instance::deleteConflictClauses(bool delete_all) {
  statistics_.times_conflict_clauses_cleaned_++;
  std::vector<ClauseOfs> tmp_conflict_clauses = conflict_clauses_;
  conflict_clauses_.clear();
  std::vector<double> tmp_ratios;
  for (auto clause_ofs : tmp_conflict_clauses) {
    double score = getHeaderOf(clause_ofs).score();
//    lifetime = statistics_.num_conflicts_ - getHeaderOf(clause_ofs).creation_time();
//    tmp_ratios.push_back(score/lifetime);
    tmp_ratios.push_back(score);
  }
  std::vector<double> tmp_ratiosB = tmp_ratios;

  sort(tmp_ratiosB.begin(), tmp_ratiosB.end());

  double cutoff = -1;
  if (!tmp_ratiosB.empty())
    cutoff = tmp_ratiosB[tmp_ratiosB.size()/2];

  for (unsigned i = 0; i < tmp_conflict_clauses.size(); i++) {
    if (delete_all) {
      markClauseDeleted(tmp_conflict_clauses[i]);
      continue;
    }
    if (tmp_ratios[i] < cutoff) {
      if (!markClauseDeleted(tmp_conflict_clauses[i]))
        conflict_clauses_.push_back(tmp_conflict_clauses[i]);
    } else {
      conflict_clauses_.push_back(tmp_conflict_clauses[i]);
    }
  }
  return true;
}


bool Instance::markClauseDeleted(ClauseOfs cl_ofs) {
  // only first literal may possibly have cl_ofs as antecedent
  if (isAntecedentOf(cl_ofs, *beginOf(cl_ofs)))
    return false;

  literal(*beginOf(cl_ofs)).removeWatchLinkTo(cl_ofs);
  literal(*(beginOf(cl_ofs)+1)).removeWatchLinkTo(cl_ofs);
  return true;
}


bool Instance::createfromFile(const std::string &file_name) {
  if (!file_name.empty())
```

(continues on next page)

(continued from previous page)

```
    statistics_.input_file_ = file_name;
  unsigned max_ignore = 1000000;  // Max number of characters on line to ignore.

  // Initialize the literal pool
  literal_pool_.clear();
  literal_pool_.push_back(SENTINEL_LIT);

  // Builds the list
  variables_.clear();
  variables_.emplace_back();  //initializing the Sentinel
  literal_values_.clear();
  unit_clauses_.clear();
  unused_vars_.clear();

  // BEGIN File input
  std::ifstream input_file(statistics_.input_file_);
  if (!input_file)
    ExitWithError("Cannot open file: " + statistics_.input_file_, EX_IOERR);

  char c;
  while ((input_file >> c) && c != 'p')
    ParseCnfCommentForSupport(input_file);
  std::string idstring;
  long nVars = -1, nCls = -1;
  if (!((input_file >> idstring) && idstring == "cnf" && (input_file >> nVars)
      && (input_file >> nCls)))
    ExitWithError("Invalid CNF file", EX_PROTOCOL);
  else if (nVars <= 0)
    ExitWithError("Invalid variable count.  At least one variable is required.", EX_
↪PROTOCOL);
  else if (nCls < 0)
    ExitWithError("Invalid clause count.  A negative clause count is invalid.", EX_
↪PROTOCOL);

  variables_.resize(nVars + FIRST_VAR);
  literal_values_.resize(nVars + FIRST_VAR, X_TRI);
  struct stat file_status;
  stat(statistics_.input_file_ .c_str(), &file_status);
  literal_pool_.reserve(file_status.st_size);
  conflict_clauses_.reserve(2 * nCls);
  occurrence_lists_.clear();
  occurrence_lists_.resize(nVars + FIRST_VAR);

  std::vector<LiteralID> literals;
  literals_.clear();
  literals_.resize(nVars + FIRST_VAR);
  std::vector<bool> var_used(nVars + FIRST_VAR, false);

  // Analyze each clause and add literals/variables as appropriate.
  unsigned clauses_added = 0;
  while ((input_file >> c) && clauses_added < nCls) {
    input_file.unget();  // extracted a nonspace character to determine if
                         // we have a clause, so put it back

    if ((c == '-') || isdigit(c)) {
      literals.clear();  // Empty the literal set in the clause
      bool skip_clause = false;
```

(continues on next page)

(continued from previous page)

```
      long lit;
      while ((input_file >> lit) && lit != 0) {
        var_used[abs(lit)] = true;  // Mark the variable as used.
        bool duplicate_literal = false;
        for (auto i : literals) {
          // Checks if same literal already in the clause
          if (i.toInt() == lit) {
            duplicate_literal = true;
            break;
          }
          // Checks if a clause has a literal and its complement in the same clause.
          if (i.toInt() == -lit) {
            skip_clause = true;
            break;
          }
        }
        if (!duplicate_literal)
          literals.emplace_back(lit);
      }
      if (!skip_clause) {
        assert(!literals.empty());  // May report a fail in an UNSAT file.
        clauses_added++;
        statistics_.incorporateClauseData(literals);
        long cl_ofs = addClause(literals);
        if (cl_ofs == CLAUSE_ADDING_ERROR)
          return false;
        // If the clause is non-binary and non-unit, then
        // Add to the occurrence list for the literal.
        if (literals.size() >= 3)
          for (auto l : literals)
            occurrence_lists_[l].push_back(cl_ofs);
      }
    } else if (c == 'c') {
      input_file >> c;
      ParseCnfCommentForSupport(input_file);
      continue;
    }
    input_file.ignore(max_ignore, '\n');
  }
  // END NEW
  input_file.close();
  //  /// END FILE input
  statistics_.num_variables_ = statistics_.num_original_variables_ =␣
↪(VariableIndex)nVars;
  statistics_.num_used_variables_ = num_variables();
  statistics_.num_indep_support_variables_ = independent_support.size();
//  statistics_.num_free_variables_ = nVars - num_variables();  // This line seems␣
↪not to work right
  if (!independent_support.empty()) {
    has_independent_support_ = true;
    std::sort(independent_support.begin(), independent_support.end());
  }

  // Get a list of unused variables.
  statistics_.num_free_variables_ = 0;
  for (VariableIndex i = 1; i < var_used.size(); i++) {
    if (!var_used[i]) {
```

(continues on next page)

(continued from previous page)

```
      statistics_.num_free_variables_++;
//      unused_vars_.push_back(i);  // Done instead at the top of var stack
    }
  }
  statistics_.num_original_clauses_ = static_cast<ClauseIndex>(nCls);

  statistics_.num_original_binary_clauses_ = statistics_.num_binary_clauses_;
  statistics_.num_original_unit_clauses_ = statistics_.num_unit_clauses_ = unit_
↪clauses_.size();

  original_lit_pool_size_ = literal_pool_.size();
  return true;
}


void Instance::ParseCnfCommentForSupport(std::ifstream &input_file) {
  std::string ind_str, line;
  std::istringstream iss;
  // Check if the comment line contains an independent support
  std::getline(input_file, line);
  iss.str(line);
  iss.clear();
  iss >> ind_str;
  if (ind_str != "ind")
    return;
  VariableIndex support_var;
  while ((iss >> support_var) && support_var != 0)
    independent_support.emplace_back(support_var);
}
```

Includes

- algorithm
- fstream
- instance.h (*File instance.h*)
- model_sampler.h (*File model_sampler.h*)
- string
- sys/stat.h
- vector

File instance.h

Parent directory (src)

Contents

- *Definition (src/instance.h)*
- *Includes*

- *Included By*
- *Classes*
- *Defines*

Definition (`src/instance.h`)

Program Listing for File instance.h

Return to documentation for file (`src/instance.h`)

```
#ifndef INSTANCE_H_
#define INSTANCE_H_

#include <sysexits.h>
#include <limits.h>

#include <cassert>
#include <string>
#include <vector>

#include "statistics.h"
#include "structures.h"
#include "containers.h"


#define CLAUSE_ADDING_ERROR INT_MIN

class Instance {
 protected:
  void unSet(LiteralID lit) {
    var(lit).ante = Antecedent(NOT_A_CLAUSE);
    var(lit).decision_level = INVALID_DL;
    literal_values_[lit] = X_TRI;
    literal_values_[lit.neg()] = X_TRI;
  }

  Antecedent & getAntecedent(LiteralID lit) {
    return variables_[lit.var()].ante;
  }
  const Antecedent & antecedent(LiteralID lit) const {
    return variables_[lit.var()].ante;
  }

  bool hasAntecedent(LiteralID lit) const {
    return variables_[lit.var()].ante.isAnt();
  }

  bool isAntecedentOf(ClauseOfs ante_cl, LiteralID lit) {
    return var(lit).ante.isAClause() && (var(lit).ante.asCl() == ante_cl);
  }

  bool isolated(VariableIndex v) {
    LiteralID lit(v, false);
    return (literal(lit).binary_links_.size() <= 1)
```

(continues on next page)

(continued from previous page)

```
        & occurrence_lists_[lit].empty()
        & (literal(lit.neg()).binary_links_.size() <= 1)
        & occurrence_lists_[lit.neg()].empty();
  }

  bool free(VariableIndex v) {
    return isolated(v) & isActive(v);
  }

  bool deleteConflictClauses(bool delete_all = false);
  bool markClauseDeleted(ClauseOfs cl_ofs);

  // Compact the literal pool erasing all the clause
  // information from deleted clauses
  void compactConflictLiteralPool();

  // we assert that the formula is consistent
  // and has not been found UNSAT yet
  // hard wires all assertions in the literal stack into the formula
  // removes all set variables and essentially reinitiallizes all
  // further data
  void compactClauses();
  void compactVariables();
  void cleanClause(ClauseOfs cl_ofs);

  // END access to variables and literals


  unsigned long num_conflict_clauses() const {
    return conflict_clauses_.size();
  }
  const VariableIndex num_variables() const {
    return (VariableIndex)variables_.size() - 1;
  }
  bool createfromFile(const std::string &file_name);

  DataAndStatistics statistics_;

  std::vector<LiteralID> literal_pool_;

  unsigned long original_lit_pool_size_;

  LiteralIndexedVector<Literal> literals_;

  LiteralIndexedVector<std::vector<ClauseOfs>> occurrence_lists_;

  std::vector<ClauseOfs> conflict_clauses_;

  std::vector<LiteralID> unit_clauses_;

  std::vector<Variable> variables_;
  LiteralIndexedVector<TriValue> literal_values_;


  /*------------------------------------------
  --          Begin Sampler Objects          --
  ------------------------------------------*/
```

(continues on next page)

(continued from previous page)

```
  std::vector<VariableIndex> unused_vars_;
  /*-----------------------------------------
  --         End Sampler Objects           --
  -----------------------------------------*/

  void decayActivities() {
    for (auto l_it = literals_.begin(); l_it != literals_.end(); l_it++)
      l_it->activity_score_ *= 0.5;

    for (auto clause_ofs : conflict_clauses_)
        getHeaderOf(clause_ofs).decayScore();
  }

  void updateActivities(ClauseOfs clause_ofs) {
    getHeaderOf(clause_ofs).increaseScore();
    for (auto it = beginOf(clause_ofs); *it != SENTINEL_LIT; it++) {
      literal(*it).increaseActivity();
    }
  }

  bool isUnitClause(const LiteralID lit) {
    for (auto l : unit_clauses_)
      if (l == lit)
        return true;
    return false;
  }
  bool existsUnitClauseOf(VariableIndex v) {
    for (auto l : unit_clauses_)
      // Extract the unit clause's variable number
      if (l.var() == v)
        return true;
    return false;
  }
  void ParseCnfCommentForSupport(std::ifstream &input_file);

  // addUnitClause checks whether lit or lit.neg() is already a
  // unit clause
  // a negative return value implied that the Instance is UNSAT
  bool addUnitClause(const LiteralID lit) {
    for (auto l : unit_clauses_) {
      if (l == lit)
        return true;
      if (l == lit.neg())
        return false;
    }
    unit_clauses_.push_back(lit);
    return true;
  }
  inline long addClause(std::vector<LiteralID> &literals);

  // adds a UIP Conflict Clause
  // and returns it as an Antecedent to the first
  // literal stored in literals
  inline Antecedent addUIPConflictClause(std::vector<LiteralID> &literals);

  inline bool addBinaryClause(LiteralID litA, LiteralID litB);
```

(continues on next page)

(continued from previous page)

```
  // BEGIN access to variables, literals, clauses

  inline Variable& var(const LiteralID lit) {
    //return variables_[lit.var()];
    return const_cast<Variable&>(var_const(lit));
  }
  inline const Variable& var_const(const LiteralID lit) const {
    return variables_[lit.var()];
  }
  Literal & literal(LiteralID lit) {
    return literals_[lit];
  }

  inline bool isSatisfied(const LiteralID &lit) const {
    return literal_values_[lit] == T_TRI;
  }

  bool isResolved(LiteralID lit) {
    return literal_values_[lit] == F_TRI;
  }
  bool isActive(LiteralID lit) const {
    return literal_values_[lit] == X_TRI;
  }

  std::vector<LiteralID>::const_iterator beginOf(ClauseOfs cl_ofs) const {
    return literal_pool_.begin() + cl_ofs;
  }
  std::vector<LiteralID>::iterator beginOf(ClauseOfs cl_ofs) {
    return literal_pool_.begin() + cl_ofs;
  }

  decltype(literal_pool_.begin()) conflict_clauses_begin() {
    return literal_pool_.begin() + original_lit_pool_size_;
  }

  ClauseHeader &getHeaderOf(ClauseOfs cl_ofs) {
    return *reinterpret_cast<ClauseHeader *>(&literal_pool_[cl_ofs
                                                            -␣
↪ClauseHeader::overheadInLits()]);
  }

  bool isSatisfied(ClauseOfs cl_ofs) {
    for (auto lt = beginOf(cl_ofs); *lt != SENTINEL_LIT; lt++)
      if (isSatisfied(*lt))
        return true;
    return false;
  }
  bool has_independent_support_ = false;
  std::vector<VariableIndex> independent_support;
};

long Instance::addClause(std::vector<LiteralID> &literals) {
  if (literals.size() == 1) {
    // MT_TODO Deal properly with the situation that opposing unit clauses are learned
//    assert(!isUnitClause(literals[0].neg()));
    if (isUnitClause(literals[0].neg()))
      return CLAUSE_ADDING_ERROR;
```

(continues on next page)

(continued from previous page)

```
    unit_clauses_.push_back(literals[0]);
    return 0;
  }
  if (literals.size() == 2) {
    addBinaryClause(literals[0], literals[1]);
    return 0;
  }
  for (unsigned i = 0; i < ClauseHeader::overheadInLits(); i++)
    literal_pool_.emplace_back(0);
  ClauseOfs cl_ofs = literal_pool_.size();

  for (auto l : literals) {
    literal_pool_.push_back(l);
    literal(l).increaseActivity(1);
  }
  // make an end: SENTINEL_LIT
  literal_pool_.push_back(SENTINEL_LIT);
  literal(literals[0]).addWatchLinkTo(cl_ofs);
  literal(literals[1]).addWatchLinkTo(cl_ofs);
  getHeaderOf(cl_ofs).set_creation_time(statistics_.num_conflicts_);
  return cl_ofs;
}


Antecedent Instance::addUIPConflictClause(std::vector<LiteralID> &literals) {
    Antecedent ante(NOT_A_CLAUSE);
    statistics_.num_clauses_learned_++;
    ClauseOfs cl_ofs = addClause(literals);
    if (cl_ofs != 0) {
      conflict_clauses_.push_back(cl_ofs);
      getHeaderOf(cl_ofs).set_length(literals.size());
      ante = Antecedent(cl_ofs);
    } else if (literals.size() == 2) {
      ante = Antecedent(literals.back());
      statistics_.num_binary_conflict_clauses_++;
    } else if (literals.size() == 1) {
      statistics_.num_unit_clauses_++;
    }
    return ante;
  }

bool Instance::addBinaryClause(LiteralID litA, LiteralID litB) {
  if (literal(litA).hasBinaryLinkTo(litB))
    return false;
  literal(litA).addBinLinkTo(litB);
  literal(litB).addBinLinkTo(litA);
  literal(litA).increaseActivity();
  literal(litB).increaseActivity();
  return true;
}


#endif /* INSTANCE_H_ */
```

Includes

- `cassert`
- `containers.h` (*File containers.h*)
- `limits.h`
- `statistics.h` (*File statistics.h*)
- `string`
- `structures.h` (*File structures.h*)
- `sysexits.h`
- `vector`

Included By

- *File instance.cpp*
- *File solver.h*

Classes

- *Class Instance*

Defines

- *Define CLAUSE_ADDING_ERROR*

File main.cpp

Parent directory (`src`)

Contents

- *Definition (`src/main.cpp`)*
- *Includes*
- *Functions*

Definition (`src/main.cpp`)

Program Listing for File main.cpp

Return to documentation for file (`src/main.cpp`)

```
#include "sysexits.h"

#include <vector>
#include <iostream>
#include "solver.h"

int printInputArgumentDescription();
int main(int argc, char *argv[]) {
  if (argc == 1 || (argc == 2 && strcmp(argv[0], "-h") == 0))
    printInputArgumentDescription();

  Solver theSolver(argc, argv);

  if (theSolver.config().perform_random_sampling_)
    theSolver.sample_models();
  else
    theSolver.solve();

  return EXIT_SUCCESS;
}
int printInputArgumentDescription() {
  std::cout << "Usage: spur [settings]\n"
            << "\t -cnf [cnf_file] Path to the CNF file\n"
            << "\t -s [s] \t Number of models \"s\" to sample uniformly at random\n"
            << "\t -tp    \t Forces two-pass sampling. (Only applicable when s=1)\n"
            << "\t -out [out_file] Path to write the specified samples\n"
            << "\t -no-sample-write Disable writing the final samples to a file.\n"
            << "\t -count-only\t Perform only model counting.  Disable sampling.\n"
            << "\t -q     \t Quiet mode\n"
            << "\t -v     \t Verbose and trace mode\n"
            << "\t -d     \t Debug mode\n"
            << "\t -t [s] \t Set time bound to s seconds\n"
            << "\t -cs [n]\t Set max cache size to n MB\n"
            << std::flush;
  exit(EX_USAGE);
}
```

Includes

- iostream
- solver.h (*File solver.h*)
- sysexits.h
- vector

Functions

- *Function main*
- *Function printInputArgumentDescription*

File model_sampler.cpp

Parent directory (`src`)

Contents

- *Definition (`src/model_sampler.cpp`)*
- *Includes*

Definition (`src/model_sampler.cpp`)

Program Listing for File model_sampler.cpp

Return to documentation for file (`src/model_sampler.cpp`)

```
#include <gmpxx.h>
#include <fstream>

// Used for precision string printing
#include <iomanip>
#include <sstream>
#include <algorithm>

#include "stack.h"
#include "model_sampler.h"
//#include "top_tree_sampler.h"


// Satisfy the linker by initializing the static variables
VariableIndex SampleAssignment::num_var_ = 0;
VariableIndex SampleAssignment::var_vec_len_ = 0;

void SamplesManager::exportFinal(std::ostream &out,
                                 const DataAndStatistics& statistics,
                                 const SolverConfiguration& config) {
  time_t ltime = time(&ltime);
  tm cur_time;
  localtime_r(&ltime, &cur_time);

  // Create the output file header
  char buf[50];
  out << "#START_HEADER" << "\n"
      << "start_time," << asctime_r(&cur_time, buf)  // Implicit new line.
      << "formula_file," << statistics.input_file_ << "\n"
      << "num_vars," << statistics.num_variables_ << "\n"
      << "independent_support_size," << statistics.num_indep_support_variables_ << "\n
↪"
      << "num_clauses," << statistics.num_clauses() << "\n"
      << "tot_num_models," << statistics.final_solution_count().get_str() << "\n"
      << "max_component_split_depth," << statistics.max_component_split_depth_ << "\n"
      << "max_branch_var_depth," << statistics.max_branch_var_depth_ << "\n"
      << "num_samples," << config.num_samples_ << "\n"
      << "num_second_pass_groups," << statistics.numb_second_pass_vars_.size() << "\n"
```

(continues on next page)

(continued from previous page)

```
      << "num_second_pass_vars";

  VariableIndex average_rem_var_cnt = 0;
  for (auto rem_var_cnt : statistics.numb_second_pass_vars_) {
    out << "," << rem_var_cnt;
    average_rem_var_cnt += rem_var_cnt;
  }
  std::stringstream stream;
  stream << std::fixed << std::setprecision(2)
         << static_cast<double>(average_rem_var_cnt)/statistics.numb_second_pass_vars_
↪.size();
  out << "\n";
//  TopTreeSampler::printTreeSolverStatistics(out);
  out << "avg_second_pass_var," << stream.str() << "\n"
      << "execution_time," << statistics.sampler_time_elapsed_ << "\n"
      << "pass1_time," << statistics.sampler_pass_1_time_ << "\n"
      << "pass2_time," << statistics.sampler_pass_2_time_ << "\n"
      << "#END_HEADER\n";

  // Do not write the solutions to the console.
  if (&out == &std::cout || config.disable_samples_write_)
    return;

  // Write the samples (if any
  out << "\n\n#START_SAMPLES\n";
  if (statistics.final_solution_count_ > 0) {
    for (const auto &sample : samples_)
      out << sample.sample_count() << "," << sample.ToString() << "\n";
  } else {
    out << "UNSAT\n";
  }
  out << "#END_SAMPLES" << std::endl;
}


void SamplesManager::reservoirSample(const Component * active_comp,
                                     const std::vector<LiteralID> &literal_stack,
                                     const mpz_class &solution_weight,
                                     const mpz_class &weight_multiplier,
                                     const AltComponentAnalyzer &ana,
                                     const VariableIndex literal_stack_ofs,
                                     const std::vector<VariableIndex> &freed_vars,
                                     const std::vector<CacheEntryID> &cached_comp_ids,
                                     const CachedAssignment &cached_assn,
                                     SampleAssignment& cached_sample) {
  assert(num_samples() > 0);
  if (solution_weight == 0)
    return;  // UNSAT so move along.

  mpz_class sample_weight = solution_weight;
  if (weight_multiplier != 1)
    sample_weight *= weight_multiplier;
  solution_count_ += sample_weight;

  SampleSize num_samples_to_replace;
  if (solution_count_ == sample_weight) {
    // First solution always gets weight equivalent to the number of samples.
```

(continues on next page)

(continued from previous page)

```
    assert(samples_.empty());

    num_samples_to_replace = num_samples();
    if (config_->verbose) {
      std::stringstream ss;
      ss << "\t\tInitial sample auto-selected.";
      PrintInColor(ss, COLOR_GREEN);
    }
  } else {
    // Use a binomial random variable to determine how many samples to replace.
    std::vector<SampleSize> samples_to_replace;
    GenerateSamplesToReplace(sample_weight, samples_to_replace);
    num_samples_to_replace = static_cast<SampleSize>(samples_to_replace.size());
    // Nothing to replace in this case.
    if (num_samples_to_replace == 0) {
      if (config_->perform_sample_caching()) {
        BuildSample(cached_sample, active_comp, literal_stack,
                    literal_stack_ofs, ana, freed_vars, cached_comp_ids);
        cached_sample.IncorporateCachedAssignment(cached_assn);
        assert(cached_sample.VerifyEmancipatedVars());
      }
      return;
    }

    RemoveSamples(samples_to_replace);
    if (config_->verbose) {
      std::stringstream ss;
      ss << "\t\tReservoir sampling accepted " << samples_to_replace.size() << "␣
↪samples.";
      PrintInColor(ss, COLOR_GREEN);
    }
  }

  samples_.emplace_back(SampleAssignment(num_samples_to_replace));
  BuildSample(samples_.back(), active_comp, literal_stack,
              literal_stack_ofs, ana, freed_vars, cached_comp_ids);

  // Handle the special requirements of sample caching.
  if (config_->perform_sample_caching()) {
    samples_.back().IncorporateCachedAssignment(cached_assn);
    cached_sample = SampleAssignment(config_->num_samples_to_cache_, samples_.back());
    assert(cached_sample.VerifyEmancipatedVars());
    assert(num_samples() == cached_sample.sample_count());
  }
}

void SamplesManager::GenerateSamplesToReplace(const mpz_class &new_sample_weight,
                                              std::vector<SampleSize> &samples_to_
↪replace) const {
  assert(new_sample_weight <= solution_count_);
  SampleSize num_samples_to_replace = Random::Mpz::binom(num_samples(), solution_
↪count_,
                                                         new_sample_weight);
  samples_to_replace.clear();
  if (num_samples_to_replace == 0)
    return;
```

(continues on next page)

(continued from previous page)

```
  Random::SelectRangeInts(num_samples(), num_samples_to_replace, samples_to_replace);
  // This sorts in DESCENDING ORDER.  Hence, it would be <5, 4, 3, 2, 1>.  This is for
  // Simplified removal later.
  std::sort(samples_to_replace.rbegin(), samples_to_replace.rend());
}



bool SamplesManager::VerifySolutions(const std::string &input_file_path,
                                     const bool skip_unassigned) const {
  std::vector<std::vector<signed long>> clauses;
  buildCnfClauseLiterals(input_file_path, clauses);
  bool cls_sat;
  for (const auto &sample : samples_) {
    for (SampleSize i = 0; i < sample.sample_count(); i++) {
      PartialAssignment assn;
      sample.BuildRandomizedPartialAssignment(assn);
      for (auto cls : clauses) {
        cls_sat = false;
        for (auto lit : cls) {
          // Any unassigned literals automatically fail.
          if (assn[abs(lit)] == ASSN_U) {
            if (skip_unassigned) {// If true, ignore all clauses with unassigned␣
↪variables
              cls_sat = true;
              break;
            }
            std::cerr << "Error: Variable #" << abs(lit) << " Unassigned bit in final␣
↪sample\n";
            return false;
          }
          // One true literal satisfies the clause so we can break
          if ((lit < 0 && assn[abs(lit)] == ASSN_F)
              || (lit > 0 && assn[abs(lit)] == ASSN_T)) {
            cls_sat = true;
            break;
          }
        }
        if (cls_sat)
          continue;
        std::stringstream ss;
        ss << "Assignment is invalid.  Clause is ";
        for (auto lit : cls)
          ss << lit << ", ";
        PrintError(ss);
        return false;
      }
    }
  }
  return true;
}

void SamplesManager::BuildSample(SampleAssignment &new_sample,
                                 const Component *active_comp,
                                 const std::vector<LiteralID> &literal_stack,
                                 const VariableIndex last_branch_lit,
                                 const AltComponentAnalyzer &ana,
```

(continues on next page)

(continued from previous page)

```
                                  const std::vector<VariableIndex> &freed_vars,
                                  const std::vector<CacheEntryID> &cached_comp_ids) {
  assert(last_branch_lit < literal_stack.size() || literal_stack.empty());
  assert(new_sample.num_set_vars() == 0);

  // Literal stack values are constrained so explicitly set them
  for (auto lit : literal_stack)
    new_sample.setVarAssignment(lit.var(), lit.sign() ? ASSN_T : ASSN_F);
  new_sample.addEmancipatedVars(freed_vars);
  new_sample.addCachedCompIds(cached_comp_ids);

  // Get all the free/unassigned variables
  std::vector<VariableIndex> new_free_vars;
  VariableIndex i = 0;
  const auto &component_data = active_comp->getData();
  // Run to the end of the variables
  while (component_data[i] != varsSENTINEL) {
    VariableIndex var_num = component_data[i];
    i++;
    // Ignore all non-free variables.
    // Sometimes variables set in the last or in BCP may appear free but are not
    if (ana.scoreOf(var_num) > 0)
      continue;
    if (SamplesManager::isVarInLitStack(var_num, literal_stack, last_branch_lit))
      continue;
    new_free_vars.push_back(var_num);
    // DebugZH - Make sure literal was not missed in the literal stack.
    assert(!SamplesManager::isVarInLitStack(var_num, literal_stack));
  }
  new_sample.addEmancipatedVars(new_free_vars);
}

void SamplesManager::buildCnfClauseLiterals(const std::string &input_file_path,
                                            std::vector<std::vector<signed long>> &
↪clauses) {
  // BEGIN File input
  std::ifstream input_file(input_file_path);
  if (!input_file)
    ExitWithError("Cannot open file " + input_file_path + " Solution validation␣
↪failed.", EX_IOERR);

  // Remove the comments header
  VariableIndex nVars;
  ClauseIndex nCls;
  const uint64_t max_ignore = UINT64_MAX;  // Max number of characters on line to␣
↪ignore.
  std::string idstring;
  char c;
  while (input_file >> c && c != 'p')
    input_file.ignore(max_ignore, '\n');
  if (!(input_file >> idstring && idstring == "cnf" && input_file >> nVars && input_
↪file >> nCls))
    ExitWithError("Invalid CNF file\n" "Solution validation failed.", EX_DATAERR);

  // Analyze each clause and add literals/variables as appropriate.
  long lit;
  clauses.clear();
```

(continues on next page)

(continued from previous page)

```
  while ((input_file >> c) && clauses.size() < nCls) {
    // Ignore comment inline
    if (c == 'c') {
      input_file.ignore(max_ignore, '\n');
      continue;
    }
    clauses.emplace_back();  // Create a new clause
    input_file.unget();  // Extracted a non-space character to determine if a clause,␣
↪so put it back
    if ((c == '-') || isdigit(c)) {
      while ((input_file >> lit) && lit != 0) {
        clauses.back().push_back(lit);
      }
    }
    input_file.ignore(max_ignore, '\n');
    if (clauses.back().empty())
      clauses.pop_back();
  }
  input_file.close();
  assert(clauses.size() == nCls);
}

void SamplesManager::merge(SamplesManager &other, const mpz_class &other_multiplier,
                           const std::vector<VariableIndex> &freed_vars,
                           const std::vector<CacheEntryID> &cached_comp_ids,
                           const CachedAssignment & cached_assn,
                           SampleAssignment& cached_sample) {
  // No solutions in the component split so move on.
  if (other.solution_count_ == 0 || other_multiplier == 0)
    return;

  // Determine which samples to take from the other component split.
  mpz_class other_weighted = other.solution_count_;
  if (other_multiplier > 1)
    other_weighted *= other_multiplier;
  solution_count_ += other_weighted;

  other.AddEmancipatedVars(freed_vars);
  other.AddCachedCompIds(cached_comp_ids);
  // Handle the case of sample caching.
  if (config_->perform_sample_caching()) {
    assert(config_->num_samples_to_cache_ == 1);
    other.samples_.front().IncorporateCachedAssignment(cached_assn);
    cached_sample = other.samples_.front();
  }

  std::vector<SampleSize> samples_to_replace;
  GenerateSamplesToReplace(other_weighted, samples_to_replace);
  if (samples_to_replace.empty())
    return;
  RemoveSamples(samples_to_replace);

  // What is removed from the existing set is what is kept from the other set.
  other.KeepSamples(samples_to_replace);

  if (config_->verbose) {
    std::stringstream ss;
```

(continues on next page)

(continued from previous page)

```
    ss << "\t\tMerged " << samples_to_replace.size() << " samples from the component␣
↪branches.";
    PrintInColor(ss, COLOR_GREEN);
  }
  append(other);
  assert(GetActualSampleCount() == num_samples());
}


//bool IsVarInLiteralStack(const std::vector<LiteralID> &literal_stack, VariableIndex␣
↪var) {
//  for (unsigned i = 0; i < literal_stack.size(); i++)
//    if (literal_stack[i].var() == var) {
//      std::stringstream ss;
//      ss << "Var #" << var << " is in the literal stack at level " << i << " with␣
↪val "
//         << (literal_stack[i].sign());
//      PrintInColor(std::cerr, ss.str(), COLOR_RED);
//      return true;
//    }
//  return false;
//}
```

Includes

- algorithm
- fstream
- gmpxx.h
- iomanip
- model_sampler.h (*File model_sampler.h*)
- sstream
- stack.h (*File stack.h*)

File model_sampler.h

Parent directory (src)

Contents

- *Definition (src/model_sampler.h)*
- *Includes*
- *Included By*
- *Classes*
- *Typedefs*

Definition (`src/model_sampler.h`)

Program Listing for File model_sampler.h

Return to documentation for file (`src/model_sampler.h`)

```
#ifndef SHARPSAT_SOLUTION_RECIPE_H
#define SHARPSAT_SOLUTION_RECIPE_H

#include <gmpxx.h>

#include <random>
#include <fstream>
#include <string>
#include <vector>
#include <list>
#include <algorithm>

#include "structures.h"
#include "component_types/component.h"
#include "statistics.h"
#include "alt_component_analyzer.h"
#include "solver_config.h"
#include "cached_assignment.h"
#include "rand_distributions.h"

typedef std::vector<uint32_t> AssignmentContainer;

class SampleAssignment {
  friend class SamplesManager;
 private:
  explicit SampleAssignment(SampleSize sample_count, const SampleAssignment &other) :
      sample_count_(sample_count), assn_(other.assn_), num_vars_set_(other.num_vars_
↪set_),
      emancipated_vars_(other.emancipated_vars_), cache_comp_ids_(other.cache_comp_
↪ids_) { }
  SampleSize sample_count_;
  AssignmentContainer assn_;
  VariableIndex num_vars_set_ = 0;
  std::vector<VariableIndex> remaining_vars_;
  std::vector<VariableIndex> emancipated_vars_;
  std::vector<CacheEntryID> cache_comp_ids_;
  std::string comp_cache_key_;
  static VariableIndex num_var_;
  static VariableIndex var_vec_len_;
  static const unsigned int size_of_word_ = sizeof(typename␣
↪AssignmentContainer::value_type)
                                            * BITS_PER_BYTE;
  static const VariableIndex bits_per_var_ = 2;
  static const AssignmentContainer::value_type var_mask_ = 0x3;
  bool VerifyStitchingCompatibility(const SampleAssignment &other) const {
    assert(assn_.size() == other.assn_.size());
    assert(sample_count_ <= other.sample_count_);  // Less than or equal for␣
↪simplified code
                                                   // when stitching the␣
↪SamplesManager objects
```

(continues on next page)

(continued from previous page)

```
    AssignmentEncoding this_val, other_val;
    for (VariableIndex i = FIRST_VAR; i <= num_var_; i++) {
      this_val = this->var_assignment(i);
      other_val = other.var_assignment(i);
      if (this_val != other_val && this_val != ASSN_U && other_val != ASSN_U) {
        std::cerr << "Stitching error on variable #" << i << std::endl
                  << "Implicit Value = " << this_val << ". Other Value = "
                  << other_val << std::endl;
        return false;
      }
    }
    return true;
  }
  inline static void calculateWordAndBitNumbers(const VariableIndex &var_num,
                                                 VariableIndex &word_num,␣
↪VariableIndex &bit_num) {
    assert(var_num >= FIRST_VAR && var_num <= num_var_);
    word_num = (var_num * bits_per_var_) / size_of_word_;
    assert(word_num >= 0 && word_num < var_vec_len_);
    bit_num = (var_num * bits_per_var_) % size_of_word_;
  }
  const bool VerifyEmancipatedVars() const {
    if (emancipated_vars_.empty())
      return true;

    std::vector<VariableIndex> all_vars = emancipated_vars_;
    std::sort(all_vars.begin(), all_vars.end());
    if (var_assignment(all_vars[0]) != ASSN_U)
      return false;
    for (VariableIndex i = 1; i < all_vars.size(); i++) {
      if (all_vars[i] == all_vars[i-1] || var_assignment(all_vars[i]) != ASSN_U)
        return false;
    }
    return true;
  }
  template <typename T>
  static const bool VerifyNoDuplicates(const std::vector<T> &vec) {
    if (vec.empty())
      return true;

    std::vector<T> vec_copy = vec;
    std::sort(vec_copy.begin(), vec_copy.end());
    for (VariableIndex i = 1; i < vec_copy.size(); i++) {
      if (vec_copy[i] == vec_copy[i-1])
        return false;
    }
    return true;
  }
  const std::vector<VariableIndex>& emancipated_vars() const { return emancipated_
↪vars_; }
  void DeleteEmancipatedVars(const std::vector<VariableIndex> &vars_to_delete) {
    for (auto &var_id : vars_to_delete) {
      auto itr = std::find(emancipated_vars_.begin(), emancipated_vars_.end(), var_
↪id);
      emancipated_vars_.erase(itr);
    }
  }
```

(continues on next page)

(continued from previous page)

```
  void addEmancipatedVars(const std::vector<VariableIndex> &new_emancipated_vars) {
    if (new_emancipated_vars.empty())
      return;

    emancipated_vars_.insert(emancipated_vars_.end(), new_emancipated_vars.begin(),
                             new_emancipated_vars.end());
//    for (auto var_id : new_emancipated_vars) {
//      if (!IsVarEmancipated(var_id))
//        emancipated_vars_.push_back(var_id);
//    }
    assert(VerifyEmancipatedVars());
  }
  void addCachedCompIds(const std::vector<CacheEntryID> &cache_comp_ids) {
    comp_cache_key_.clear();
    cache_comp_ids_.insert(cache_comp_ids_.end(), cache_comp_ids.begin(), cache_comp_
↪ids.end());
    assert(VerifyNoDuplicates<CacheEntryID>(cache_comp_ids_));
  }
  void DecreaseSampleCount(SampleSize dec_sample_count) {
    assert(dec_sample_count > 0 && dec_sample_count < sample_count_);
    sample_count_ -= dec_sample_count;
  }
  void zeroSampleCount() { sample_count_ = 0; }
  inline const VariableIndex num_var() const { return num_var_; }
  void IncorporateCachedAssignment(const CachedAssignment & cached_assn) {
    for (auto lit : cached_assn.literals()) {
      assert(var_assignment(lit.var()) == ASSN_U);  // Only assign to unassigned␣
↪variables
      setVarAssignment(lit.var(), lit.sign() ? ASSN_T : ASSN_F);
    }
    if (!cached_assn.emancipated_vars().empty())
      emancipated_vars_.insert(emancipated_vars_.end(), cached_assn.emancipated_
↪vars().begin(),
                               cached_assn.emancipated_vars().end());
    assert(VerifyEmancipatedVars());
  }
  inline void stitch(const SampleAssignment &other) {
    assert(VerifyStitchingCompatibility(other));
    for (VariableIndex i = 0; i < assn_.size(); i++)
      assn_[i] &= other.assn_[i];

    addEmancipatedVars(other.emancipated_vars_);
    addCachedCompIds(other.cache_comp_ids_);
    num_vars_set_ += other.num_vars_set_;

    remaining_vars_.clear();
    // Make sure no variables miraculously materialized.
    assert(num_set_vars_const() + emancipated_vars_.size() <= num_var());
  }
  inline SampleAssignment split(SampleSize new_assn_sample_count) {
    DecreaseSampleCount(new_assn_sample_count);
    return SampleAssignment(new_assn_sample_count, *this);
  }
  inline void set_sample_count(SampleSize new_sample_count) { sample_count_ = new_
↪sample_count; }
  inline void BuildRandomizedPartialAssignment(PartialAssignment &all_vars) const {
    GetPartialAssignment(all_vars);
```

(continues on next page)

(continued from previous page)

```
    for (auto var : emancipated_vars_)
      all_vars[var] = (Random::uniform(0, 1)) ? ASSN_F : ASSN_T;
  }
  static SampleAssignment buildUnsetterAssignment(std::vector<VariableIndex> vars_to_
↪keep) {
    SampleAssignment new_assn;
    for (auto &var : vars_to_keep)
      new_assn.setVarAssignment(var, ASSN_F);
    return new_assn;
  }
  void unsetVariableAssignments(SampleAssignment &unsetter) {
    assert(assn_.size() == unsetter.assn_.size());
    for (VariableIndex i = 0; i < assn_.size(); i++)
      assn_[i] |= unsetter.assn_[i];
  }
  std::vector<VariableIndex> GetRemainingVariables() {
    if (remaining_vars_.empty())
      num_set_vars();  // Builds the unset variables list.
    return remaining_vars_;
  }

 public:
  explicit SampleAssignment(SampleSize sample_count) : sample_count_(sample_count) {
    assn_.resize(var_vec_len_, static_cast<typename AssignmentContainer::value_type>(-
↪1));
  }
  SampleAssignment() : SampleAssignment(0) {}
  inline std::string ToString() const {
    std::stringstream ss;
    for (VariableIndex i = FIRST_VAR; i <= num_var(); i++) {
      AssignmentEncoding var_val = var_assignment(i);
      switch (var_val) {
        case ASSN_F: ss << '0'; break;
        case ASSN_T: ss << '1'; break;
        case ASSN_U: ss << '*'; break;
      }
    }
    return ss.str();
  }
  inline const AssignmentEncoding var_assignment(const VariableIndex &var) const {
    VariableIndex word_num, bit_num;
    calculateWordAndBitNumbers(var, word_num, bit_num);
    return (AssignmentEncoding) ((assn_[word_num] >> bit_num) & var_mask_);
  }
  inline void GetPartialAssignment(PartialAssignment &all_vars) const {
    all_vars.clear();
    all_vars.resize(num_var_ + FIRST_VAR);
    for (VariableIndex i = FIRST_VAR; i <= num_var_; i++)
      all_vars[i] = var_assignment(i);
  }
  inline void setVarAssignment(const VariableIndex var, const AssignmentEncoding &
↪val) {
    assert((val == ASSN_F || val == ASSN_T) && var_assignment(var) == ASSN_U);

    VariableIndex word_num, bit_num;
    calculateWordAndBitNumbers(var, word_num, bit_num);
```

(continues on next page)

(continued from previous page)

```
    // Updates the bits of interest only by masking then setting them.
    assn_[word_num] = (assn_[word_num] & ~(var_mask_ << bit_num)) | (val << bit_num);
    num_vars_set_++;
    remaining_vars_.clear();
  }
  static void set_num_var(const VariableIndex num_var) {
    assert(num_var_ == 0);  // This function should onyl be called once.
    num_var_ = num_var;
    var_vec_len_ = ((num_var + 1) * bits_per_var_ / size_of_word_)+ 1;
  }
  const bool IsComplete() const {
//    return cache_comp_ids().empty();
    return num_set_vars_const() + emancipated_vars_.size() == num_var_;
  }
//  /**
//   * Complete Assignment Checker
//   *
//   * Determines whether the sample model is partial or complete.
//   *
//   * @return true if the sample model is a complete assignment.
//   */
//  bool IsComplete() const {
//    return num_set_vars_const() + emancipated_vars_.size() == num_var_;
//  }
//  /**
//   * Accessor for the number of cached component IDs in this object.
//   * @return Number of cached component IDs in this object
//   */
//  inline const uint64_t cached_comp_count() const { return cache_comp_ids_.size(); }
  const SampleSize sample_count() const { return sample_count_; }
  const VariableIndex num_set_vars() {
    if (!remaining_vars_.empty())
      return num_vars_set_;

    // ToDo once variable setting is efficient, just return the set variable count
    num_vars_set_ = 0;
    for (VariableIndex i = FIRST_VAR; i <= num_var(); i++) {
      if (var_assignment(i) != ASSN_U) {
        num_vars_set_++;
      } else if (!IsVarEmancipated(i)) {
        remaining_vars_.emplace_back(i);
      }
    }
    return num_vars_set_;
  }
  const VariableIndex num_set_vars_const() const {
    // ToDo once variable setting is efficient, just return the set variable count
    SampleSize num_vars_set = 0;
    for (VariableIndex i = FIRST_VAR; i <= num_var(); i++)
      if (var_assignment(i) != ASSN_U)
        num_vars_set++;
    return num_vars_set;
  }
//  /**
//   * Updates the assignment of the implicit assignment with that of the specified␣
↪one.  It does
//   * NOT update the sample count
```

(continues on next page)

(continued from previous page)

```
//   *
//   * @param other Another SampleAssignment.
//   */
//  void updateAssignmentOnly(const SampleAssignment &other) {
//    this->assn_ = other.assn_;
//    this->emancipated_vars_ = other.emancipated_vars_;
//  }
  const VariableIndex num_unset_vars() {
    return num_var_ - num_set_vars() - emancipated_vars_.size();
  }
//  /**
//   * Builds and returns the set of unconstrained variables in this sample.
//   *
//   * @return Identification number of the unset variables.
//   */
//  const std::vector<VariableIndex> GetUnsetConstrainedVars() const {
//    std::vector<VariableIndex> unset_vars;
//    for (VariableIndex i = FIRST_VAR; i <= num_var(); i++)
//      if (var_assignment(i) == ASSN_U && !IsVarEmancipated(i))
//        unset_vars.push_back(i);
//    return unset_vars;
//  }
  const bool IsVarEmancipated(VariableIndex var) const {
    return std::find(emancipated_vars_.begin(), emancipated_vars_.end(), var)
           != emancipated_vars_.end();
  }
  const std::vector<CacheEntryID>& cache_comp_ids() const {
    return cache_comp_ids_;
  }
  void clear_cache_comp_ids() {
    cache_comp_ids_.clear();
  }
//  /**
//   * Generate a unique key for the cached components in the sample assignment.
//   *
//   * @return Key string for the cached components.
//   */
//  std::string GetCachedCompKey() {
//    if (!comp_cache_key_.empty())
//      return comp_cache_key_;
//    std::sort(cache_comp_ids_.begin(), cache_comp_ids_.end());
//    std::stringstream ss;
//    for (auto cached_comp : cache_comp_ids_) {
//      if (cached_comp != cache_comp_ids_[0])
//        ss << ",";
//      ss << cached_comp;
//    }
//    comp_cache_key_ = ss.str();
//    return comp_cache_key_;
//  }
};


typedef std::list<SampleAssignment> ListOfSamples;

class SamplesManager {
 private:
```

(continues on next page)

(continued from previous page)

```
  mpz_class solution_count_ = 0;
  ListOfSamples samples_;
//  /**
//   * Stores the final expected number of samples to be returned at the
//   * end of sampling.
//   */
//  static SampleSize final_num_samples_;
//  /**
//   * Stores the current number of samples being built by the sampler.
//   */
//  static SampleSize samples_manager_vector_size_;
  SampleSize tot_num_samples_;
  SolverConfiguration *config_;
//  /**
//   * Used in the random number generator.  Stores the random bits
//   * to be extracted.
//   */
//  static int random_bits_;
//  /**
//   * Next random bit to be used. It is between [0,NUM_INT_BITS-1).
//   */
//  static int next_rand_bit_;
//  static const int NUM_INT_BITS_;
  static void BuildSample(SampleAssignment &new_sample,
                          const Component *active_comp,
                          const std::vector<LiteralID> &literal_stack,
                          VariableIndex last_branch_lit,
                          const AltComponentAnalyzer &ana,
                          const std::vector<VariableIndex> &freed_vars,
                          const std::vector<CacheEntryID> &cached_comp_ids);
  static bool isVarInLitStack(const VariableIndex var_num,
                              const std::vector<LiteralID> &literal_stack,
                              const VariableIndex start_ofs = 0) {
    for (auto itr = literal_stack.begin() + start_ofs; itr != literal_stack.end();␣
↪itr++)
      if ((*itr).var() == var_num)
        return true;
    return false;
  }
  static void buildCnfClauseLiterals(const std::string &input_file_path,
                                     std::vector<std::vector<signed long>> &clauses);
  void splitSampleAndInsert(std::list<SampleAssignment>::iterator &itr,
                            const SampleSize &new_assn_sample_count) {
    assert(new_assn_sample_count > 0 && new_assn_sample_count < itr->sample_count());
    // Need to increment then decrement since this method inserts the new element␣
↪before itr
    SampleAssignment new_node = itr->split(new_assn_sample_count);
    samples_.insert(itr, new_node);
    --itr;
  }

 public:
  SamplesManager(SampleSize num_samples, SolverConfiguration &config) : config_(&
↪config) {
    tot_num_samples_ = num_samples;
  }
//  /**
```

(continues on next page)

(continued from previous page)

```
//   * Copy constructor.
//   */
//  SamplesManager(const SamplesManager &other)
//      : SamplesManager(other.num_samples(), *other.config_) {}
//  /**
//   * Equality operator.
//   *
//   * @param other Object to which the implicit object will be set.
//   * @return Reference to the new SamplesManager created.  This allows for chaining␣
↪equality
//   * operators.
//   */
//  SamplesManager& operator=(const SamplesManager &other) {
//    this->tot_num_samples_ = other.tot_num_samples_;
//    this->samples_ = other.samples_;
//    this->config_ = other.config_;
//    return *this;
//  }
  void exportFinal(std::ostream &out, const DataAndStatistics &statistics,
                   const SolverConfiguration& config);
  void reservoirSample(const Component * active_comp,
                       const std::vector<LiteralID> & literal_stack,
                       const mpz_class &solution_weight,
                       const mpz_class &weight_multiplier,
                       const AltComponentAnalyzer &ana,
                       VariableIndex literal_stack_ofs,
                       const std::vector<VariableIndex>& freed_vars,
                       const std::vector<CacheEntryID> &cached_comp_ids,
                       const CachedAssignment& cached_assn,
                       SampleAssignment& cached_sample);
  inline void GenerateSamplesToReplace(const mpz_class &new_sample_weight,
                                       std::vector<SampleSize> &samples_to_replace)␣
↪const;
//  /**
//   * Sample Variable Assignment Accessor
//   *
//   * Gets the value of the variable assignment for a specific sample in the list of␣
↪samples
//   *
//   * Debug function.
//   *
//   * @param sample_num Sample number between 0 (inclusive) and num_samples␣
↪(exclusive)
//   * @param var Variable number
//   *
//   * @return Variable's assigned value.
//   */
//  AssignmentEncoding sample_var_val(const SampleSize sample_num, const␣
↪VariableIndex var) const {
//    assert(var >= FIRST_VAR && var <= samples_[sample_num].num_var());
//    return samples_[sample_num].var_assignment(var);
//  }
  inline void stitch(SamplesManager &other) {
    assert(this->tot_num_samples_ == other.tot_num_samples_);
    if (solution_count_ == 0) {
      this->samples_ = other.samples_;
      solution_count_ = other.solution_count_;
```

(continues on next page)

(continued from previous page)

```
      return;
    } else {
      solution_count_ *= other.solution_count_;
    }
    // Handle the UNSAT case
    if (other.solution_count_ == 0) {
      this->samples_.clear();
      return;
    }

    // Depending on the number of sample objects, different stitching approaches are␣
↪faster
    StitchShuffledArray(other);

    // After the size normalization, perform the stitching sample by sample.
    assert(verifyPostStitchingCorrectness(other));
  }
  inline void StitchShuffledArray(SamplesManager &other) {
    assert(this->GetActualSampleCount() == other.GetActualSampleCount());

    std::vector<ListOfSamples::iterator> other_samples_itrs;
    other_samples_itrs.reserve(other.samples_.size());
    std::vector<SampleSize> other_sample_order;
    other_sample_order.reserve(other.tot_num_samples_);

    // Store a reference to each element in other's samples
    // These will be used for simplifying the stitching look-up.
    SampleSize i = 0;
    for (auto other_itr = other.samples_.begin(); other_itr != other.samples_.end();␣
↪++other_itr) {
      other_samples_itrs.emplace_back(other_itr);
      for (SampleSize j=0; j < other_itr->sample_count(); j++)
        other_sample_order.emplace_back(i);
      ++i;
    }
    Random::shuffle<SampleSize>(other_sample_order.begin(), other_sample_order.end());

    // Split and merge to create the permutations
    SampleSize sample_offset = 0;
    for (auto this_itr = samples_.begin(); this_itr != samples_.end(); ) {
      SampleSize sample_end = sample_offset + this_itr->sample_count();
      std::vector<SampleSize> samples_per_element(other_sample_order.size(), 0);
      for (SampleSize sample_cnt = sample_offset; sample_cnt < sample_end; ++sample_
↪cnt)
        samples_per_element[other_sample_order[sample_cnt]]++;

      // Indices from the same "other" sample are grouped together so split and␣
↪stitch.
      for (SampleSize other_cnt = 0; other_cnt < samples_per_element.size(); ++other_
↪cnt) {
        if (samples_per_element[other_cnt] == 0)
          continue;
        SplitAndStitch(this_itr, other_samples_itrs[other_cnt], samples_per_
↪element[other_cnt]);
      }

      // Update all pointer
```

(continues on next page)

(continued from previous page)

```
      sample_offset = sample_end;
    }
  }
  void SplitAndStitch(ListOfSamples::iterator &this_itr, ListOfSamples::iterator &
↪other_itr,
                      SampleSize &num_new_samples) {
    assert(num_new_samples > 0 && num_new_samples <= this_itr->sample_count()
           && num_new_samples <= other_itr->sample_count());

    // If applicable add a new node
    if (num_new_samples != this_itr->sample_count())
      splitSampleAndInsert(this_itr, num_new_samples);

    this_itr->stitch(*other_itr);
    ++this_itr;
    if (other_itr->sample_count() == num_new_samples)
      other_itr->zeroSampleCount();
    else
      other_itr->DecreaseSampleCount(num_new_samples);
  }
  const bool verifyPostStitchingCorrectness(SamplesManager &other) const {
    for (auto &sample : other.samples_) {
      if (sample.sample_count() != 0) {
        PrintInColor("An other_sample had non-zero size.", PrintColor::COLOR_RED);
        return false;
      }
    }

    if (!this->verifySampleCount()) {
      PrintInColor("The sample count of the stitched object is incorrect.",␣
↪PrintColor::COLOR_RED);
      return false;
    }

    return true;
  }
  void merge(SamplesManager &other, const mpz_class &other_multiplier,
             const std::vector<VariableIndex> &freed_vars,
             const std::vector<CacheEntryID> &cached_comp_ids,
             const CachedAssignment & cached_assn,
             SampleAssignment& cached_sample);
  bool VerifySolutions(const std::string &input_file_path,
                       bool skip_unassigned = false) const;
  const mpz_class &model_count() const { return solution_count_; }
  void AddEmancipatedVars(const std::vector<VariableIndex> &emancipated_vars) {
    for (auto &sample : samples_)
      sample.addEmancipatedVars(emancipated_vars);

    // Multiply by 2^num_unused_vars since those represent a parallel cylinder
    mpz_mul_2exp(solution_count_.get_mpz_t(), solution_count_.get_mpz_t(),
                 emancipated_vars.size());
  }
  void AddCachedCompIds(const std::vector<CacheEntryID> &cached_comp_ids) {
    for (auto &sample : samples_)
      sample.addCachedCompIds(cached_comp_ids);
  }
  void TransferVariableAssignments(ListOfSamples &others) {
```

(continues on next page)

(continued from previous page)

```
    std::vector<VariableIndex> unset_vars = others.front().GetRemainingVariables();

    assert(!unset_vars.empty());
    auto unsetter = SampleAssignment::buildUnsetterAssignment(unset_vars);
    // Delete redundant emancipated variables.
    for (auto &sample : samples_) {
      sample.unsetVariableAssignments(unsetter);
      sample.DeleteEmancipatedVars(others.front().emancipated_vars());
    }

    SampleSize sample_count = 0;
    for (auto &other : others) {
      sample_count += other.sample_count();
      other.clear_cache_comp_ids(); // Out of data cache ids from previous run.
    }
    assert(sample_count == this->num_samples());
    SamplesManager others_manager(sample_count, *config_);
    others_manager.solution_count_ = 1;
    others_manager.samples_ = others;

    this->stitch(others_manager);
  }
//  /**
//   * Solver Configuration Storer
//   *
//   * This function is used to store a reference to the solver's configuration.
//   * This is useful in case any of the run parameters are used.
//   * @param config Solver's configuration.
//   */
//  static void set_solver_config(SolverConfiguration &config) { config_ = &config; }
  ListOfSamples &samples() { return samples_; }
//  /**
//   * Sample Setter
//   *
//   * This function replaces the sample (based off the specified number)
//   * with the new model passed to the function.
//   *
//   * @param sample_num Sample number to be set
//   * @param new_model New sample model to be stored
//   */
//  inline void set_sample(SampleSize sample_num,
//                         const SampleAssignment &new_model) {
//    assert(sample_num >= 0 && sample_num < samples_.size());
//    samples_[sample_num] = new_model;
//  }
//  /**
//   * Sample Accessor
//   *
//   +   * Extracts a reference to the specified sample from the manager.
//   *
//   +   * @param sample_num Number of the sample to access - base 0
//   *
//   +   * @return Sample at the specified number
//   */
//  inline const SampleAssignment &sample(SampleSize sample_num) const {
//    assert(sample_num >= 0 && sample_num < samples_.size());
//    return samples_[sample_num];
```

(continues on next page)

(continued from previous page)

```
//  }
  inline bool IsComplete() const {
    for (const auto &sample : samples_)
      if (!sample.IsComplete())
        return false;
    return true;
  }
  inline void append(SamplesManager &other) {
    append(other.samples_);
  }
  inline void append(ListOfSamples &other) {
    samples_.splice(samples_.end(), other);
    assert(GetActualSampleCount() <= tot_num_samples_);
  }
  inline const SampleSize num_samples() const {
//    assert(verifySampleCount());
    return tot_num_samples_;
  }
  void RemoveSamples(std::vector<SampleSize> &samples_to_remove) {
    if (samples_to_remove.empty())
      return;
    if (num_samples() == samples_to_remove.size()) {
      samples_.clear();
      return;
    }

    // Delete from back to front to prevent deletion affecting counts.
    assert(!samples_.empty());
    auto sample_itr = --(samples_.end());
    SampleSize cur_sample_start_count = num_samples() - sample_itr->sample_count();
    SampleSize num_to_remove = 0;
    for (auto &sample_to_remove : samples_to_remove) {
      assert(sample_to_remove >= 0 && sample_to_remove < num_samples());
      // Skip to the next sample to remove.
      while (sample_to_remove < cur_sample_start_count) {
        if (num_to_remove > 0) {
          if (sample_itr->sample_count() == num_to_remove)
            // If no samples are left then remove the object
            sample_itr = samples_.erase(sample_itr);
          else
            sample_itr->DecreaseSampleCount(num_to_remove);
          num_to_remove = 0;
        }
        --sample_itr;
        cur_sample_start_count -= sample_itr->sample_count();
      }
      num_to_remove++;
    }

    // Handle the last element that requires removal
    if (sample_itr->sample_count() == num_to_remove)
      sample_itr = samples_.erase(sample_itr);
    else
      sample_itr->DecreaseSampleCount(num_to_remove);
    assert(GetActualSampleCount() == tot_num_samples_ - samples_to_remove.size());
  }
  void KeepSamples(std::vector<SampleSize> &samples_to_keep) {
```

(continues on next page)

(continued from previous page)

```
    if (samples_to_keep.empty()) {
      samples_.clear();
      return;
    }
    if (num_samples() == samples_to_keep.size())
      return;

    // Delete from back to front to prevent deletion affecting counts.
    assert(!samples_.empty());
    auto sample_itr = --(samples_.end());
    SampleSize cur_sample_start_count = num_samples() - sample_itr->sample_count();
    SampleSize num_to_keep = 0;
    for (auto &sample_to_keep : samples_to_keep) {
      assert(sample_to_keep >= 0 && sample_to_keep < num_samples());
      // Skip to the next sample to check
      while (sample_to_keep < cur_sample_start_count) {
        if (num_to_keep > 0) {
          sample_itr->set_sample_count(num_to_keep);
          num_to_keep = 0;
        } else {
          sample_itr = samples_.erase(sample_itr);
        }
        --sample_itr;
        cur_sample_start_count -= sample_itr->sample_count();
      }
      num_to_keep++;
    }

    // Handle the last node to keep
    sample_itr->set_sample_count(num_to_keep);
    // Any nodes never encountered have to be removed.
    if (sample_itr != samples_.begin())
      samples_.erase(samples_.begin(), sample_itr);
    assert(GetActualSampleCount() == samples_to_keep.size());
  }
  const bool verifySampleCount() const {
    return GetActualSampleCount() == tot_num_samples_;
  }
  const SampleSize GetActualSampleCount() const {
    SampleSize actual_sample_count = 0;
    for (auto &sample : samples_) {
      assert(sample.sample_count() > 0);  // Make sure no dead samples
      actual_sample_count += sample.sample_count();
    }
    return actual_sample_count;
  }
  void clear() { samples_.clear(); }
};


// * Recipe Structure Initializer
// *
// * Initializes recipe static structures.  This requires a dedicated function
// * because of the scope requirements of static objects.
// *
// * @param num_var Number of variables in the Boolean formula.
// */
```

(continues on next page)

(continued from previous page)

```
//void InitializeSamplerStructures(VariableIndex num_var);
// * Sample Size Initializer
// *
// * Initializes the number of samples to by collected by the algorithm.
// *
// * @param sample_count Number of samples
// */
//void InitializeSampleCount(SampleSize sample_count);
//
//
//bool IsVarInLiteralStack(const std::vector<LiteralID> &literal_stack, VariableIndex
↪var);

#endif //SHARPSAT_SOLUTION_RECIPE_H
```

Includes

- algorithm
- alt_component_analyzer.h (*File alt_component_analyzer.h*)
- cached_assignment.h (*File cached_assignment.h*)
- component_types/component.h (*File component.h*)
- fstream
- gmpxx.h
- list
- rand_distributions.h (*File rand_distributions.h*)
- random
- solver_config.h (*File solver_config.h*)
- statistics.h (*File statistics.h*)
- string
- structures.h (*File structures.h*)
- vector

Included By

- *File stack.h*
- *File component_management.h*
- *File instance.cpp*
- *File solver.h*
- *File model_sampler.cpp*

Classes

- *Class SampleAssignment*
- *Class SamplesManager*

Typedefs

- *Typedef AssignmentContainer*
- *Typedef ListOfSamples*

File primitive_types.h

Parent directory (`src`)

Contents

- *Definition (`src/primitive_types.h`)*
- *Includes*
- *Included By*
- *Enums*
- *Functions*
- *Defines*
- *Typedefs*

Definition (`src/primitive_types.h`)

Program Listing for File primitive_types.h

Return to documentation for file (`src/primitive_types.h`)

```
#include <vector>

#ifndef PRIMITIVE_TYPES_H_
#define PRIMITIVE_TYPES_H_

#define varsSENTINEL  0
#define clsSENTINEL   NOT_A_CLAUSE

#define EXIT_TIMEOUT 128

typedef uint64_t VariableIndex;
typedef VariableIndex ClauseIndex;
typedef VariableIndex ClauseOfs;
typedef VariableIndex ComponentVarAndCls;
typedef unsigned CacheEntryID;
```

(continues on next page)

(continued from previous page)

```
typedef int64_t DecisionLevel;

static const ClauseIndex NOT_A_CLAUSE(0);
#define SENTINEL_CL NOT_A_CLAUSE

#define BITS_PER_BYTE 8
#define FIRST_VAR 1

enum class SolverExitState {
  NO_STATE, SUCCESS//, TIMEOUT, ABORTED
};
enum class SolverNextAction {
  EXIT, RESOLVED, PROCESS_COMPONENT, BACKTRACK
};


#ifdef DEBUG
  #define toDEBUGOUT(X) std::cout << X;
#else
  #define toDEBUGOUT(X)
#endif

enum AssignmentEncoding{
  ASSN_F = 0x0, ASSN_T = 0x1, ASSN_U = 0x3,
};
typedef std::vector<AssignmentEncoding> PartialAssignment;
typedef uint32_t SampleSize;


typedef int32_t TopTreeLiteral;
typedef uint64_t TreeNodeIndex;
enum class TopTreeNodeType {
  MAX_DEPTH,
  CYLINDER,
  COMPONENT_SPLIT,
  NUM_TREE_NODE_TYPES // Always be last
};

#endif /* PRIMITIVE_TYPES_H_ */
```

Includes

- vector

Included By

- *File structures.h*
- *File component.h*
- *File cacheable_component.h*
- *File statistics.h*
- *File solver_config.h*

- *File component_archetype.h*
- *File rand_distributions.h*
- *File solver.cpp*

Enums

- *Enum AssignmentEncoding*
- *Enum SolverExitState*
- *Enum SolverNextAction*
- *Enum TopTreeNodeType*

Functions

- *Function NOT_A_CLAUSE*

Defines

- *Define BITS_PER_BYTE*
- *Define clsSENTINEL*
- *Define EXIT_TIMEOUT*
- *Define FIRST_VAR*
- *Define SENTINEL_CL*
- *Define toDEBUGOUT*
- *Define varsSENTINEL*

Typedefs

- *Typedef CacheEntryID*
- *Typedef ClauseIndex*
- *Typedef ClauseOfs*
- *Typedef ComponentVarAndCls*
- *Typedef DecisionLevel*
- *Typedef PartialAssignment*
- *Typedef SampleSize*
- *Typedef TopTreeLiteral*
- *Typedef TreeNodeIndex*
- *Typedef VariableIndex*

File rand_distributions.cpp

Parent directory (`src`)

Contents

- *Definition (`src/rand_distributions.cpp`)*
- *Includes*

Definition (`src/rand_distributions.cpp`)

Program Listing for File rand_distributions.cpp

Return to documentation for file (`src/rand_distributions.cpp`)

```
#include <cstdlib>

#include "rand_distributions.h"

// Satisfy the linker by initializing the static variables
SolverConfiguration * Random::master_config_;
std::random_device Random::rd_;      // only used once to initialise (seed) engine
std::mt19937 Random::rng_;    // random-number engine used (Mersenne-Twister in this␣
↪case)
std::uniform_int_distribution<int> Random::uni_(INT_MIN, INT_MAX);

bool Random::Mpz::use_approx_binom_ = true;
gmp_randstate_t Random::Mpz::rand_state_;
```

Includes

- `cstdlib`
- `rand_distributions.h` (*File rand_distributions.h*)

File rand_distributions.h

Parent directory (`src`)

Contents

- *Definition (`src/rand_distributions.h`)*
- *Includes*
- *Included By*
- *Classes*

Definition (`src/rand_distributions.h`)

Program Listing for File rand_distributions.h

Return to documentation for file (`src/rand_distributions.h`)

```
#ifndef PROJECT_RANDOM_H
#define PROJECT_RANDOM_H

#include <gmpxx.h>

#include <chrono> // NOLINT (build/c++11)
#include <cstdlib>
#include <iostream>
#include <random>
#include <vector>

#include "solver_config.h"
#include "primitive_types.h"
#include "sampler_tools.h"


class Random{
 public:
  static void init(SolverConfiguration * config) {
    master_config_ = config;
    gmp_randinit_mt(Random::Mpz::rand_state_);

    // create the generator seed for the random engine to reference
    long long int seed;
    if (master_config_->debug_mode) {
      if (!master_config_->quiet)
        std::cout << "WARNING: Debug mode has a fixed seed for the MPZ class.\n";
      seed = 0;
    } else {
      seed = std::chrono::high_resolution_clock::now().time_since_epoch().count();
    }
    mpz_class rand_seed;
    mpz_import(rand_seed.get_mpz_t(), 1, -1, sizeof(seed), 0, 0, &seed);
    gmp_randseed(Random::Mpz::rand_state_, rand_seed.get_mpz_t());

    if (master_config_->debug_mode) {
      if (!master_config_->quiet)
        std::cout << "WARNING: Debug mode uses a fixed random number seed." <<␣
↪std::endl;
    } else {
      Random::rng_ = std::mt19937(Random::rd_());
    }
  }
  template <typename T> inline static T uniform(T min, T max) {
    std::uniform_int_distribution<T> distribution(min, max);
    return distribution(rng_);
  }
  inline static int uniform(int min = INT_MIN, int max = INT_MAX) {
    if (min == INT_MIN && max == INT_MAX) {
      return uni_(rng_);
    } else {
```

(continues on next page)

(continued from previous page)

```
      std::uniform_int_distribution<int> distribution(min, max);
      return distribution(rng_);
    }
  }

  static SampleSize binom(SampleSize n, double p) {
    std::binomial_distribution<> d(n, p);
    return static_cast<SampleSize>(d(rng_));
  }
  template<typename ListType, typename CountType>
  static void DownsampleList(CountType target_size, std::vector<ListType> &
↪oversampled_vec,
                             bool resize = true) {
    assert(oversampled_vec.size() >= target_size);

    CountType end_point = oversampled_vec.size() - 1;
    while (end_point > target_size) {
      auto id_loc = Random::uniform<CountType>(0, end_point);
      oversampled_vec[id_loc] = oversampled_vec[end_point];  // Copy back & overwrite␣
↪the used value
      end_point--;
    }
    if (resize)
      oversampled_vec.resize(target_size);
  }
  template<typename T>
  static void shuffle(std::vector<T> &vec) {
    std::shuffle(vec.begin(), vec.end(), rng_);
  }

  class Mpz {
    // Allow the Random class to access this class' private methods/fields
    friend class Random;
   public:
    inline static void uniform(mpz_class max_z, mpz_class &rand_val) {
      mpz_urandomm(rand_val.get_mpz_t(), Random::Mpz::rand_state_, max_z.get_mpz_t());
    }
    inline static SampleSize binom(SampleSize n, const mpz_class &t,
                                   const mpz_class &a) {
      assert(t > 0 && t >= a);
      // Handle the edge cases
      if (a == 0)
        return 0;
      else if (a == t)
        return n;

      if (use_approx_binom_)
        return Random::Mpz::binom_approx(n, t, a);
      else
        return Random::Mpz::binom_exact(n, t, a);
    }

   private:
    static bool use_approx_binom_;
    static gmp_randstate_t rand_state_;
    inline static SampleSize binom_approx(SampleSize n, const mpz_class &t,
                                          const mpz_class &a) {
```

(continues on next page)

(continued from previous page)

```
      mpf_class a_mpf = a, t_mpf = t;
      mpf_class p_mpf = a_mpf / t_mpf;
      return Random::binom(n, p_mpf.get_d());
//      double p = mpz_get_d(a.get_mpz_t()) / mpz_get_d(t.get_mpz_t());
//      return Random::binom(n,p);
    }
    inline static SampleSize binom_exact(SampleSize n, const mpz_class &t,
                                         const mpz_class &a) {
      mpz_class rand_mpz = 0;
      SampleSize num_success = 0;
      for (SampleSize i = 0; i < n; i++) {
        Random::Mpz::uniform(t, rand_mpz);
        if (rand_mpz < a)
          num_success++;
      }
      return num_success;
    }
  };
  static void SelectRangeInts(SampleSize max_id, SampleSize num_elements,
                              std::vector<SampleSize> &samples_to_replace) {
    assert(max_id >= num_elements);
    samples_to_replace.clear();
    samples_to_replace.reserve(max_id);
    for (SampleSize i = 0; i < max_id; i++)
      samples_to_replace.emplace_back(i);
    //ToDo Modify this function to only ever do a maximum of max_id/2 random numbers
    Random::DownsampleList<SampleSize, SampleSize>(num_elements, samples_to_replace);
  }
  template<typename T>
  static void shuffle(typename std::vector<T>::iterator begin, // Dependent types so␣
↪add the
                      typename std::vector<T>::iterator end) {  // "typename" keyword.
    std::shuffle(begin, end, rng_);
  }

 private:
  Random() = default;
  static std::uniform_int_distribution<int> uni_;
  static SolverConfiguration * master_config_;
  static std::random_device rd_;
  static std::mt19937 rng_;
};

#endif //PROJECT_RANDOM_H
```

Includes

- chrono
- cstdlib
- gmpxx.h
- iostream
- primitive_types.h (*File primitive_types.h*)

- random
- sampler_tools.h (*File sampler_tools.h*)
- solver_config.h (*File solver_config.h*)
- vector

Included By

- *File model_sampler.h*
- *File rand_distributions.cpp*

Classes

- *Class Random*
- *Class Random::Mpz*

File sampler_tools.h

Parent directory (src)

Contents

- *Definition (src/sampler_tools.h)*
- *Includes*
- *Included By*
- *Enums*
- *Functions*
- *Defines*

Definition (src/sampler_tools.h)

Program Listing for File sampler_tools.h

Return to documentation for file (src/sampler_tools.h)

```
#ifndef SAMPLER_TOOLS_
#define SAMPLER_TOOLS_

#define ESCAPE_CHAR "\033["
#define PRINT_BOLD "1"
#define ITEM_SEP ";"
#define END_ESCAPE "m"
#define RESET_CONSOLE_COLOR (ESCAPE_CHAR "0" END_ESCAPE)

#include <sysexits.h>
```

(continues on next page)

(continued from previous page)

```
#include <sys/stat.h>

#include <iostream>
#include <string>
#include <sstream>

#include "solver_config.h"
#include "statistics.h"

#define STR_DECIMAL_BASE 10

#if defined(WIN32) || defined(_WIN32)
  #define FILE_PATH_SEPARATOR "\\"
#else
  #define FILE_PATH_SEPARATOR "/"
#endif

enum PrintColor {
//  INVERT_COLORS  =  7,
  COLOR_BLACK    = 30,
  COLOR_RED      = 31,
  COLOR_GREEN    = 32,
  COLOR_YELLOW   = 33,
  COLOR_BLUE     = 34,
  COLOR_MAGENTA  = 35,
  COLOR_CYAN     = 36,
//  COLOR_WHITE    = 37
};

inline void PrintInColor(std::ostream &out, const std::string &msg,
                         const PrintColor color = COLOR_BLACK, bool bold = true)  {
  out << ESCAPE_CHAR;
  if (bold)
    out << PRINT_BOLD ITEM_SEP;
  out << color;
  out << END_ESCAPE << msg << RESET_CONSOLE_COLOR << std::endl;
}
inline void PrintInColor(const std::string &msg,
                         PrintColor color = COLOR_BLACK, bool bold = true)  {
  PrintInColor(std::cout, msg, color, bold);
}
inline void PrintInColor(std::ostream &out, const std::stringstream &msg,
                         const PrintColor color = COLOR_BLACK, bool bold = true) {
  PrintInColor(out, msg.str(), color, bold);
}
inline void PrintInColor(const std::stringstream &msg,
                         PrintColor color = COLOR_BLACK, bool bold = true)  {
  PrintInColor(std::cout, msg.str(), color, bold);
}

inline void PrintWarning(const std::string &msg) {
  PrintInColor(std::cout, "WARNING: " + msg, COLOR_YELLOW);
}
inline void PrintError(const std::string &msg) {
  PrintInColor(std::cerr, "ERROR: " + msg, COLOR_RED);
}
inline void PrintError(const std::stringstream &msg) { PrintError(msg.str()); }
```

(continues on next page)

(continued from previous page)

```
inline void ExitWithError(const std::string &msg, const int err_code=EXIT_FAILURE) {
  PrintError(msg);
  exit(err_code);
}
inline void ExitInvalidParam(const std::string &msg) {
  ExitWithError(msg, EX_DATAERR);
}
inline const bool FileExists(const std::string &file_path) {
  struct stat buffer;
  return (stat(file_path.c_str(), &buffer) == 0);
}

#endif // SAMPLER_TOOLS_
```

Includes

- iostream
- solver_config.h (*File solver_config.h*)
- sstream
- statistics.h (*File statistics.h*)
- string
- sys/stat.h
- sysexits.h

Included By

- *File solver_config.h*
- *File rand_distributions.h*
- *File component_cache.cpp*
- *File solver.cpp*

Enums

- *Enum PrintColor*

Functions

- *Function ExitInvalidParam*
- *Function ExitWithError*
- *Function FileExists*
- *Function PrintError()*
- *Function PrintError()*

- *Function PrintInColor()*
- *Function PrintInColor()*
- *Function PrintInColor()*
- *Function PrintInColor()*
- *Function PrintWarning*

Defines

- *Define END_ESCAPE*
- *Define ESCAPE_CHAR*
- *Define FILE_PATH_SEPARATOR*
- *Define ITEM_SEP*
- *Define PRINT_BOLD*
- *Define RESET_CONSOLE_COLOR*
- *Define STR_DECIMAL_BASE*

File solver.cpp

Parent directory (`src`)

Contents

- *Definition (`src/solver.cpp`)*
- *Includes*

Definition (`src/solver.cpp`)

Program Listing for File solver.cpp

Return to documentation for file (`src/solver.cpp`)

```
#include <deque>
#include <string>
#include <utility>
#include <vector>
#include <sstream>
#include <unordered_map>

#include "primitive_types.h"
#include "solver.h"
//#include "top_tree_sampler.h"
#include "solver_config.h"
#include "sampler_tools.h"

StopWatch::StopWatch() {
```

(continues on next page)

(continued from previous page)

```
  interval_length_.tv_sec = 60;
  gettimeofday(&last_interval_start_, nullptr);
  start_time_ = stop_time_ = last_interval_start_;
}


timeval StopWatch::getElapsedTime() {
  timeval other_time = stop_time_;
  if (stop_time_.tv_sec == start_time_.tv_sec
      && stop_time_.tv_usec == start_time_.tv_usec)
    gettimeofday(&other_time, nullptr);
  long int ad = 0;
  unsigned int bd = 0;

  if (other_time.tv_usec < start_time_.tv_usec) {
    ad = 1;
    bd = 1000000;
  }
  timeval r = (struct timeval) {0};
  r.tv_sec = other_time.tv_sec - ad - start_time_.tv_sec;
  r.tv_usec = other_time.tv_usec + bd - start_time_.tv_usec;
  return r;
}


bool Solver::simplePreProcess() {
  if (!config_.perform_pre_processing)
    return true;
  assert(literal_stack_.empty());
  // BEGIN process unit clauses
  for (auto lit : unit_clauses_)
    setLiteralIfFree(lit);
  // END process unit clauses
  VariableIndex start_ofs = 0;
  bool succeeded = BCP(start_ofs);

  if (succeeded)
    succeeded &= prepFailedLiteralTest();

  // ToDo major slowdown possible without HardWireAndCompact.  Need to make a␣
↪different version.
//  if (succeeded)
//    HardWireAndCompact();
  return succeeded;
}

bool Solver::prepFailedLiteralTest() {
  unsigned long last_size;
  do {
    last_size = literal_stack_.size();
    for (unsigned v = 1; v < variables_.size(); v++) {
      if (isActive(v)) {
        unsigned long sz = literal_stack_.size();
        setLiteralIfFree(LiteralID(v, true));
        bool res = BCP(sz);
        while (literal_stack_.size() > sz) {
          unSet(literal_stack_.back());
```

(continues on next page)

(continued from previous page)

```
          literal_stack_.pop_back();
        }

        if (!res) {
          sz = literal_stack_.size();
          setLiteralIfFree(LiteralID(v, false));
          if (!BCP(sz))
            return false;
        } else {
          sz = literal_stack_.size();
          setLiteralIfFree(LiteralID(v, false));
          bool resb = BCP(sz);
          while (literal_stack_.size() > sz) {
            unSet(literal_stack_.back());
            literal_stack_.pop_back();
          }
          if (!resb) {
            sz = literal_stack_.size();
            setLiteralIfFree(LiteralID(v, true));
            if (!BCP(sz))
              return false;
          }
        }
      }
    }
  } while (literal_stack_.size() > last_size);

  return true;
}

//void Solver::HardWireAndCompact() {
//  compactClauses();
//  compactVariables();
//  literal_stack_.clear();
//
//  for (auto l = LiteralID(1, false); l != literals_.end_lit(); l.inc()) {
//    literal(l).activity_score_ = literal(l).binary_links_.size() - 1;
//    literal(l).activity_score_ += occurrence_lists_[l].size();
//  }
//
//  statistics_.num_unit_clauses_ = unit_clauses_.size();
//
//  statistics_.num_original_binary_clauses_ = statistics_.num_binary_clauses_;
//  statistics_.num_original_unit_clauses_ = statistics_.num_unit_clauses_ = unit_
↪clauses_.size();
//  initStack(num_variables());
//  original_lit_pool_size_ = literal_pool_.size();
//}

void Solver::sample_models() {
  if (!createfromFile(statistics_.input_file_)) {
    PrintFinalSamplerResults();
    return;
  }

  SampleAssignment::set_num_var(statistics_.num_variables_);
```

(continues on next page)

(continued from previous page)

```
  Solver temp_solver(config_, statistics_);
  temp_solver.createfromFile(statistics_.input_file_);
  LinkConfigAndStatistics();

  reservoir_sample_models(PartialAssignment(), temp_solver);

  ReportSharpSatResults();
  PrintFinalSamplerResults();
}

void Solver::reservoir_sample_models(const PartialAssignment &partial_assn, Solver &
↪temp_solver) {
  if (!InitializeSolverAndPreprocess(partial_assn))
    return;

  PerformInitialSampling();
  statistics_.sampler_pass_1_time_ = sampler_stopwatch_.getElapsedSeconds();

  // The UNSAT case can be reached in two different ways either by the
  // preprocessor or by running sharpSAT's main flow.
  if (statistics_.final_solution_count_ > 0 && config_.perform_two_pass_sampling_)
    FillPartialAssignments(temp_solver);

  // Print the final results after all sampling then quit.
  sampler_stopwatch_.stop();
  statistics_.sampler_time_elapsed_ = sampler_stopwatch_.getElapsedSeconds();
  statistics_.sampler_pass_2_time_ = statistics_.sampler_time_elapsed_
                                                                - statistics_.sampler_
↪pass_1_time_;
}


void Solver::PrintFinalSamplerResults() {
  if (!config_.quiet)
    std::cout << "\nTotal Sampler Execution Time: " << statistics_.sampler_time_
↪elapsed_ << "s\n\n";
  // Print any helper messages
  if (!config_.quiet && statistics_.final_solution_count_ < config_.num_samples_) {
      std::stringstream ss;
      ss << "Only " << statistics_.final_solution_count_ << " models exist but "
         << config_.num_samples_ << " samples were requested." << std::endl;
      PrintWarning(ss.str());
  }

  // Export to a file and the console.
  if (!config_.quiet)
    final_samples_.exportFinal(std::cout, statistics_, config_);
  if (!config_.samples_output_file.empty()) {
    std::ofstream samples_file(config_.samples_output_file);
    final_samples_.exportFinal(samples_file, statistics_, config_);
    samples_file.close();
  }
//  // ToDo Remove Debug Verification
//  bool valid_samples = final_samples_.VerifySolutions(statistics_.input_file_,
//                                                      config_.skip_partial_
↪assignment_fill);
//  if (!valid_samples)
```

(continues on next page)

(continued from previous page)

```
//    ExitWithError("INVALID SAMPLES", EX_SOFTWARE);
//  if (!final_samples_.IsComplete())
//    ExitWithError("INCOMPLETE SAMPLES", EX_SOFTWARE);
}


void Solver::PerformInitialSampling() {
  if (!config_.perform_two_pass_sampling_)
    config_.EnableSampleCaching();

  if (!config_.quiet) {
    if (config_.perform_two_pass_sampling_)
      std::cout << "STAGE #1: ";
    std::cout << "Build the initial partial assignments" << std::endl;
  }
  // If the solver has not been shown to be UNSAT by the preprocess,
  // run the basic sampler and model count
  statistics_.exit_state_ = countSAT();

  unused_vars_ = stack_.top().emancipated_vars();
  statistics_.set_final_solution_count(stack_.top().getTotalModelCount());
  statistics_.num_long_conflict_clauses_ = num_conflict_clauses();

  // Clean up the component so it is not an issue during pass two
  if (stack_.back().isComponentSplit())
    stack_.back().unsetAsComponentSplit();

  // If UNSAT, report it and exit.
  if (statistics_.final_solution_count_ == 0)
    return;

  if (unused_vars_.empty()) {
    if (config_.verbose)
      std::cout << "No unused variables.  Continuing..." << std::endl;
  } else {
    if (config_.verbose)
      std::cout << "Number of unused variables: " << unused_vars_.size() << std::endl;
    samples_stack_.back().AddEmancipatedVars(unused_vars_);
  }
  final_samples_ = samples_stack_.back();
  assert(IsEndSamplesStackSizeValid());
  assert(final_samples_.model_count() == statistics_.final_solution_count_);

  if (!config_.quiet) {
    if (config_.perform_two_pass_sampling_)
      std::cout << "STAGE #1: ";
    std::cout << "COMPLETED building initial partial assignments" << std::endl;
  }
}


inline void Solver::FillPartialAssignments(Solver &temp_solver) {
  if (config_.skip_partial_assignment_fill)
    return;

  if (!config_.quiet)
    std::cout << "STAGE #2 - Filling in partial assignments..." << std::endl;
```

(continues on next page)

(continued from previous page)

```
  SamplesManager updated_samples = SamplesManager(config_.num_samples_, config_);

  // Free the memory to be used by the descendant solver's cache
//  unit_clauses_.clear();
  comp_manager_.initialize(literals_, literal_pool_, config_.quiet);
  deleteConflictClauses(true);

  // Group the samples so that same cache ids are tested together
  std::vector<std::pair<std::vector<CacheEntryID>, long>> grouped_samples;
  std::vector<ListOfSamples> list_elements;
  for (auto &sample : final_samples_.samples()) {
    auto new_key = sample.cache_comp_ids();
    SampleSize i = 0;
    for (i = 0; i < grouped_samples.size(); i++)
      if (grouped_samples[i].first == new_key)
        break;
    if (i < grouped_samples.size()) {
      grouped_samples[i].second += sample.sample_count();
      list_elements[i].emplace_back(sample);
    } else {
      grouped_samples.emplace_back(std::pair<std::vector<CacheEntryID>, long>(new_key,
                                                                              sample.
↪sample_count()));
      list_elements.emplace_back();
      list_elements.back().emplace_back(sample);
    }
  }

  SampleSize group_cnt = 0, tot_count = 0;
  ListOfSamples * samples_list;
  auto sample_itr = final_samples_.samples().begin();
  while (sample_itr != final_samples_.samples().end()) {
    auto key = sample_itr->cache_comp_ids();
    long new_sample_count = 0;
    SampleSize group_idx;
    for (group_idx = 0; group_idx < grouped_samples.size(); group_idx++) {
      if (grouped_samples[group_idx].first == key) {
        new_sample_count = grouped_samples[group_idx].second;
        samples_list = &list_elements[group_idx];
        break;
      }
    }
    if (new_sample_count < 0) {
      if (sample_itr->cache_comp_ids().empty())
        ++sample_itr;
      else
        sample_itr = final_samples_.samples().erase(sample_itr);
      continue;
    }
    grouped_samples[group_idx].second = -new_sample_count;
    tot_count += new_sample_count;
    ++group_cnt;
    if (sample_itr->IsComplete()) {
      ++sample_itr;
      statistics_.numb_second_pass_vars_.push_back(0);
      if (!config_.quiet)
```

(continues on next page)

(continued from previous page)

```
        std::cout << "Sample #" << group_cnt << " of " << grouped_samples.size()
                  << " is already a complete assignment.  Continuing..." << std::endl;
      continue;
    }

    // If the assignment is complete, then go to next sample
    VariableIndex num_unset_vars = sample_itr->num_unset_vars();
    statistics_.numb_second_pass_vars_.push_back(num_unset_vars);
    assert(num_unset_vars < statistics_.num_variables_);
    if (!config_.quiet)
      std::cout << "Completing sample #" << group_cnt << " of " << grouped_samples.
↪size()
                << " which has " << num_unset_vars << " variable"
                << ((num_unset_vars == 1) ? "" : "s") << " unset and " << new_sample_
↪count
                << " sample" << ((new_sample_count == 1) ? "" : "s") << "." <<␣
↪std::endl;

    // Build an intermediary solver
    Solver new_solver = temp_solver;
    new_solver.config_.quiet = new_solver.config_.quiet || !config_.verbose;
    PartialAssignment partial_assn;
    sample_itr->GetPartialAssignment(partial_assn);
    new_solver.config_.num_samples_ = static_cast<SampleSize>(new_sample_count);
    if (new_solver.config_.num_samples_ == 1)
      new_solver.config_.EnableSampleCaching();
    new_solver.reservoir_sample_models(partial_assn, temp_solver);

    assert(new_solver.IsEndSamplesStackSizeValid());
    assert(new_solver.stack_.top_const().getTotalModelCount()
           == new_solver.final_samples_.model_count());
    assert(new_solver.stack_.top_const().getTotalModelCount() > 0);
    assert(new_solver.final_samples_.IsComplete());

    // Take the updated sample and store it.
    if (samples_list->size() > 1)
      new_solver.final_samples_.TransferVariableAssignments(*samples_list);

    // Insert the new elements into the samples and delete the old one.
    final_samples_.samples().splice(sample_itr, new_solver.final_samples_.samples());
    sample_itr = final_samples_.samples().erase(sample_itr);
  }
  if (tot_count != config_.num_samples_)
    ExitWithError("Not all samples tested", EX_SOFTWARE);

  if (!config_.quiet)
    std::cout << "STAGE #2 - COMPLETE" << std::endl;
  // Relink the configuration and statistics objects which may have been unlinked␣
↪during the
  // dependent tasks.
  LinkConfigAndStatistics();
}



bool Solver::InitializeSolverAndPreprocess(const PartialAssignment &partial_assn) {
  statistics_.set_final_solution_count(0);  // Zero out initial model count
```

(continues on next page)

(continued from previous page)

```
  applyPartialAssignment(partial_assn);

  initStack(num_variables());

  if (!config_.quiet) {
    if (!config_.perform_random_sampling_)
      std::cout << "Performing Exact Model Counting..." << std::endl;
    else
      std::cout << "Performing Uniform Model Sampling..." << std::endl;
    std::cout << "Input File:  " << statistics_.input_file_ << std::endl;
    if (config_.perform_random_sampling_)
      std::cout << "Output File: " << config_.samples_output_file << std::endl;
  }

  if (!config_.quiet)
    std::cout << "\nPreprocessing ..." << std::flush;
  bool notfoundUNSAT = simplePreProcess();
  if (!config_.quiet)
    std::cout << " DONE" << std::endl;

  if (!notfoundUNSAT) {
    statistics_.exit_state_ = SolverExitState::SUCCESS;
    statistics_.final_solution_count_ = 0;
    if (!config_.quiet)
      std::cout << "\nFOUND UNSAT DURING PREPROCESSING " << std::endl;
    return notfoundUNSAT;
  }

  // If preprocessor did not find the formula to be unsatisfiable, then
  // get ready for model counting
  if (!config_.quiet)
    statistics_.printShortFormulaInfo();

  last_ccl_deletion_time_ = last_ccl_cleanup_time_ = statistics_.getTime();

  violated_clause.reserve(num_variables());

  comp_manager_.initialize(literals_, literal_pool_, config_.quiet);
  return notfoundUNSAT;
}


void Solver::applyPartialAssignment(const PartialAssignment &partial_assn) {
  // Optionally constrain the solution with unit clauses that match the partial␣
↪assignment
  if (partial_assn.empty())
    return;

  assert(partial_assn.size() == num_variables() + FIRST_VAR);
  std::vector<LiteralID> literals;
  literals.emplace_back();
  for (VariableIndex var_num = FIRST_VAR; var_num <= num_variables(); var_num++) {
    AssignmentEncoding var_assn = partial_assn[var_num];
    if (var_assn != ASSN_U) {
      literals[0] = LiteralID((var_assn == ASSN_F) ? (-1 * static_cast<int>(var_num))
                                                   : static_cast<int>(var_num));
```

(continues on next page)

(continued from previous page)

```
      statistics_.incorporateClauseData(literals);
      addClause(literals);
    }
  }
  statistics_.num_original_unit_clauses_ = statistics_.num_unit_clauses_ = unit_
↪clauses_.size();
}


void Solver::solve(const PartialAssignment &partial_assn) {
  if (createfromFile(this->statistics_.input_file_)) {
    bool notfoundUNSAT = InitializeSolverAndPreprocess(partial_assn);

    if (notfoundUNSAT) {
      statistics_.exit_state_ = countSAT();
      statistics_.set_final_solution_count(stack_.top().getTotalModelCount());

      statistics_.num_long_conflict_clauses_ = num_conflict_clauses();
    }
  }

  ReportSharpSatResults();
}

void Solver::ReportSharpSatResults() {
  statistics_.sampler_time_elapsed_ = sampler_stopwatch_.getElapsedSeconds();
  comp_manager_.gatherStatistics();
//  if (!results_output_file.empty())
//    statistics_.writeToFile(results_output_file);
  if (!config_.quiet)
    statistics_.printShort();
}

SolverExitState Solver::countSAT() {
  SolverNextAction state = SolverNextAction::RESOLVED;
  // Put the initial item on the top of the samples stack
  PushNewSamplesManagerOnStack();
  while (true) {
    while (comp_manager_.findNextRemainingComponentOf(stack_.top(), literal_stack_,
                                                      samples_stack_.back())) {
      decideLiteral();
      if (sampler_stopwatch_.timeBoundBroken()) {
        if (!config_.quiet)
          PrintError("TIMEOUT");
        exit(EXIT_TIMEOUT);
      }

      if (sampler_stopwatch_.interval_tick())
        printOnlineStats();

      while (!bcp()) {
        state = resolveConflict();
        if (state == SolverNextAction::BACKTRACK)
          break;
      }
      if (state == SolverNextAction::BACKTRACK)
        break;
```

(continues on next page)

(continued from previous page)

```
    }
    state = backtrack();
    if (state == SolverNextAction::EXIT)
      return SolverExitState::SUCCESS;
    while (state != SolverNextAction::PROCESS_COMPONENT && !bcp()) {
      state = resolveConflict();
      if (state == SolverNextAction::BACKTRACK) {
        state = backtrack();
        if (state == SolverNextAction::EXIT)
          return SolverExitState::SUCCESS;
      }
    }
  }
}


void Solver::decideLiteral() {
  // Store the decision level info for determining what to do with the samples
  StackLevel *prev_top = &stack_.top();

  // establish another decision stack level
  StackLevel newLevel = StackLevel(stack_.top().currentRemainingComponent(),
                                   literal_stack_.size(),
                                   comp_manager_.component_stack_size());
  if (config_.perform_random_sampling_ && !stack_.top().isComponentSplit())
    newLevel.configureNewLevel(stack_.top(), stack_.get_decision_level());

  stack_.push_back(newLevel);

  // Manage the passing of the samples
  if (config_.perform_random_sampling_) {
    // Each component splits necessitate formula merging so create blanks
    if (prev_top->isComponentSplit()) {
      if (prev_top->isFirstComponent()) {
        PushNewSamplesManagerOnStack();
        prev_top->markFirstComponentComplete();
      }
      PushNewSamplesManagerOnStack();
    }
  }

  float max_score = -1;
  VariableIndex max_score_var = 0;
  // Select the variable with the highest score as the branch variable
  for (auto it = comp_manager_.superComponentOf(stack_.top()).varsBegin();
       *it != varsSENTINEL; it++) {
    float score = scoreOf(*it);
    if (score > max_score) {
      max_score = score;
      max_score_var = *it;
    }
  }
  // this assert should always hold,
  // if not then there is a bug in the logic of countSAT();
  assert(max_score_var != 0);

  // Create the literal to assign
```

(continues on next page)

(continued from previous page)

```
  // Select either the negated or unnegated literal depending on
  // which form of the literal is most active.
  LiteralID theLit(max_score_var,
                   literal(LiteralID(max_score_var, true)).activity_score_
                   > literal(LiteralID(max_score_var, false)).activity_score_);

  setLiteralIfFree(theLit);
  statistics_.num_decisions_++;
  set_variable_depth_++;
  if (set_variable_depth_ > statistics_.max_branch_var_depth_)
    statistics_.max_branch_var_depth_ = set_variable_depth_;

  if (statistics_.num_decisions_ % 128 == 0)
//    if (statistics_.num_conflicts_ % 128 == 0)
    decayActivities();
  // decayActivitiesOf(comp_manager_.superComponentOf(stack_.top()));
  if (config_.verbose)
    std::cout << "Literal Stack Location #" << literal_stack_.size() - 1 << ":␣
↪Variable #"
              << literal_stack_.back().var() << " assigned to "
              << ((literal_stack_.back().sign()) ? "TRUE" : "FALSE") << std::endl;
  assert(stack_.top_const().remaining_components_ofs() <= comp_manager_.component_
↪stack_size());
}


SolverNextAction Solver::backtrack() {
  assert(stack_.top_const().remaining_components_ofs() <= comp_manager_.component_
↪stack_size());
  do {
    if (stack_.top().branch_found_unsat())
      comp_manager_.removeAllCachePollutionsOf(stack_.top());
    else if (stack_.top().anotherCompProcessible())
      return SolverNextAction::PROCESS_COMPONENT;

    if (!stack_.top().isSecondBranch()) {
      LiteralID aLit = TOS_decLit();
      assert(stack_.get_decision_level() > 0);
      if (stack_.top().isComponentSplit())
        ProcessSampleComponentSplitBacktrack();
      // Must close branch after processing backtracking as it will affect the flow␣
↪otherwise
      stack_.top().changeBranch();
      reactivateTOS();
      setLiteralIfFree(aLit.neg(), Antecedent(NOT_A_CLAUSE));
      if (config_.verbose)
        std::cout << "Literal Stack Location #" << literal_stack_.size() - 1 << ":␣
↪Variable #"
                  << literal_stack_.back().var() << " switched to "
                  << (literal_stack_.back().sign() ? "TRUE" : "FALSE") << std::endl;
      return SolverNextAction::RESOLVED;
    }

    if (stack_.get_decision_level() <= 0)
      break;  // Bottomed out the decision stack so program execution is complete.
    reactivateTOS();
```

(continues on next page)

(continued from previous page)

```
    assert(stack_.size() >= 2);
    // Merge the solution count up the stack.
    (stack_.end() - 2)->includeSolution(stack_.top().getTotalModelCount());
    if (config_.perform_random_sampling_)
      ProcessSampleComponentSplitBacktrack();

    // OTHERWISE:  backtrack further
    // Store the model count AND POTENTIALLY the assignment in cache.
    ProcessCacheStore();

    // Process top-tree sampling for max depth.
    // Must be after processing component splits to ensure that an immediately␣
↪dependent
    // component split is cleared.
    if (config_.perform_top_tree_sampling && set_variable_depth_ == config_.max_top_
↪tree_depth_) {
//      if (!stack_.top().HasNoDescendentModels() && HasNoUpperComponentSplit()) {
      if (!stack_.top().HasNoDescendentModels()) {
        mpz_class multiplier;
        if (stack_.on_deck().isComponentSplit())
          multiplier = 1;
        else
          multiplier = stack_.on_deck().getSamplerSolutionMultiplier();
      }
    }
    set_variable_depth_--;
    assert(set_variable_depth_ >= 0 && set_variable_depth_ <= literal_stack_.size());

    stack_.pop_back();
    // step to the next component not yet processed
    stack_.top().nextUnprocessedComponent();

    assert(stack_.top_const().remaining_components_ofs() < comp_manager_.component_
↪stack_size()+1);
  } while (stack_.get_decision_level() >= 0);
  return SolverNextAction::EXIT;
}


void Solver::ProcessCacheStore() {
  if (stack_.size() == 1 || stack_.top().getTotalModelCount() == 0
      || !config_.perform_sample_caching()) {
    // Just store the model count as normal
    comp_manager_.cacheModelCountOf(stack_.top().super_component(),
                                    stack_.top().getTotalModelCount());
  } else {
    // Bundle the model count with a partial assignment
    StackLevel top = stack_.top();
    Component top_comp = comp_manager_.component(top.super_component());
    SampleAssignment cache_sample = top.random_cache_sample();
    comp_manager_.cacheModelCountAndAssignment(top.super_component(), top.
↪getTotalModelCount(),
                                               cache_sample, top_comp);
    stack_.on_deck().set_cache_sample(cache_sample);
  }
}
```

(continues on next page)

(continued from previous page)

```
void Solver::ProcessSampleComponentSplitBacktrack() {
  // Combine the results of a component split
  if (stack_.top().isComponentSplit()) {
    // Merge the component split with its parent
    if (config_.verbose) {
      std::stringstream ss;
      ss << "Component " << stack_.top().super_component() << " merging component␣
↪split "
         << "at depth " << StackLevel::componentSplitDepth() << ".";
      PrintInColor(ss, COLOR_MAGENTA);
    }

    if (config_.store_sampled_models()) {
      SampleAssignment cached_sample;
      VariableIndex on_deck = samples_stack_.size() - 2;
      samples_stack_[on_deck].merge(samples_stack_.back(),
                                    stack_.top().getSamplerSolutionMultiplier(),
                                    stack_.top().emancipated_vars(),
                                    stack_.top().cached_comp_ids(),
                                    stack_.top().cached_assn(), cached_sample);
      samples_stack_.pop_back();
      if (config_.perform_sample_caching()) {
        // ToDo modify to support multiple sample caching
        stack_.top().ClearCachedAssn();
        stack_.top().set_cache_sample(cached_sample);
      }
//      assert(samples_stack_.back().VerifySolutions(statistics_.input_file_, true));␣
↪ // DebugZH
    }
    stack_.top().unsetAsComponentSplit();
  }
  // Only close a component branch after testing both true and false paths.
  // If there is a back to back component split, may need to close both the component
  // split and the branch in a single round.
  if (stack_.size() <= 1)
    return;
  if (stack_.top().isSecondBranch() && stack_.on_deck().isComponentSplit()) {
    if (config_.verbose) {
      std::stringstream ss;
      ss << "Component branch #" << stack_.top().super_component() << " split end␣
↪reached at depth "
         << StackLevel::componentSplitDepth() << ".  Stitching the sample.";
      PrintInColor(ss, COLOR_BLUE);
    }
    if (config_.store_sampled_models()) {
      samples_stack_[samples_stack_.size() - 2].stitch(samples_stack_.back());
      samples_stack_.pop_back();
//      assert(samples_stack_.back().VerifySolutions(statistics_.input_file_, true));␣
↪ // DebugZH
    }
  }
}

SolverNextAction Solver::resolveConflict() {
  recordLastUIPCauses();

  if (statistics_.num_clauses_learned_ - last_ccl_deletion_time_
```

(continues on next page)

(continued from previous page)

```
      > statistics_.clause_deletion_interval()) {
    deleteConflictClauses();
    last_ccl_deletion_time_ = statistics_.num_clauses_learned_;
  }

  if (statistics_.num_clauses_learned_ - last_ccl_cleanup_time_ > 100000) {
    compactConflictLiteralPool();
    last_ccl_cleanup_time_ = statistics_.num_clauses_learned_;
  }

  statistics_.num_conflicts_++;

  assert(stack_.top_const().remaining_components_ofs() <= comp_manager_.component_
↪stack_size());

  assert(uip_clauses_.size() == 1);

  // DEBUG
  if (uip_clauses_.back().empty() && !config_.quiet)
    std::cerr << "EMPTY CLAUSE FOUND" << std::endl;
  // END DEBUG

  stack_.top().mark_branch_unsat();
  // BEGIN Backtracking
  // maybe the other branch had some solutions
  if (stack_.top().isSecondBranch()) {
    return SolverNextAction::BACKTRACK;
  }

  Antecedent ant(NOT_A_CLAUSE);
  // this has to be checked since using implicit BCP
  // and checking literals there not exhaustively
  // we cannot guarantee that uip_clauses_.back().front() == TOS_decLit().neg()
  // this is because we might have checked a literal
  // during implict BCP which has been a failed literal
  // due only to assignments made at lower decision levels
  if (uip_clauses_.back().front() == TOS_decLit().neg()) {
    assert(TOS_decLit().neg() == uip_clauses_.back()[0]);
    var(TOS_decLit().neg()).ante = addUIPConflictClause(
        uip_clauses_.back());
    ant = var(TOS_decLit()).ante;
  }
//  // RRR
//  else if (var(uip_clauses_.back().front()).decision_level
//      < stack_.get_decision_level()
//      && assertion_level_ <  stack_.get_decision_level()) {
//         stack_.top().set_both_branches_unsat();
//         return BACKTRACK;
//  }
//
//
//  // RRR
  assert(stack_.get_decision_level() > 0);
  assert(stack_.top_const().branch_found_unsat());

  // we do not have to remove pollutions here,
  // since conflicts only arise directly before
```

(continues on next page)

(continued from previous page)

```
  // remaining components are stored
  // hence
  assert(
      stack_.top_const().remaining_components_ofs() == comp_manager_.component_stack_
↪size());

  stack_.top().changeBranch();
  LiteralID lit = TOS_decLit();
  reactivateTOS();
  setLiteralIfFree(lit.neg(), ant);
  // END Backtracking
  return SolverNextAction::RESOLVED;
}

bool Solver::bcp() {
// the asserted literal has been set, so we start
// bcp on that literal
  VariableIndex start_ofs = literal_stack_.size() - 1;

// BEGIN process unit clauses
  for (auto lit : unit_clauses_)
    setLiteralIfFree(lit);
// END process unit clauses

  bool bSucceeded = BCP(start_ofs);

  if (config_.perform_failed_lit_test && bSucceeded) {
    bSucceeded = implicitBCP();
  }
  return bSucceeded;
}

bool Solver::BCP(VariableIndex start_at_stack_ofs) {
  for (VariableIndex i = start_at_stack_ofs; i < literal_stack_.size(); i++) {
    LiteralID unLit = literal_stack_[i].neg();
    // BEGIN Propagate Bin Clauses
    for (auto bt = literal(unLit).binary_links_.begin();
         *bt != SENTINEL_LIT; bt++) {
      if (isResolved(*bt)) {
        setConflictState(unLit, *bt);
        return false;
      }
      setLiteralIfFree(*bt, Antecedent(unLit));
    }
    // END Propagate Bin Clauses
    for (auto itcl = literal(unLit).watch_list_.rbegin();
         *itcl != SENTINEL_CL; itcl++) {
      bool isLitA = (*beginOf(*itcl) == unLit);
      auto p_watchLit = beginOf(*itcl) + 1 - isLitA;
      auto p_otherLit = beginOf(*itcl) + isLitA;

      if (isSatisfied(*p_otherLit))
        continue;
      auto itL = beginOf(*itcl) + 2;
      while (isResolved(*itL))
        itL++;
      // either we found a free or satisfied lit
```

(continues on next page)

(continued from previous page)

```
      if (*itL != SENTINEL_LIT) {
        literal(*itL).addWatchLinkTo(*itcl);
        std::swap(*itL, *p_watchLit);
        *itcl = literal(unLit).watch_list_.back();
        literal(unLit).watch_list_.pop_back();
      } else {
        // or p_unLit stays resolved
        // and we have hence no free literal left
        // for p_otherLit remain poss: Active or Resolved
        if (setLiteralIfFree(*p_otherLit, Antecedent(*itcl))) {  // implication
          if (isLitA)
            std::swap(*p_otherLit, *p_watchLit);
        } else {
          setConflictState(*itcl);
          return false;
        }
      }
    }
  }
  return true;
}

//bool Solver::implicitBCP() {
//  static std::vector<LiteralID> test_lits(num_variables());
//  static LiteralIndexedVector<unsigned char> viewed_lits(num_variables() + 1,
//      0);
//
//  unsigned stack_ofs = stack_.top().literal_stack_ofs();
//  while (stack_ofs < literal_stack_.size()) {
//    test_lits.clear();
//    for (auto it = literal_stack_.begin() + stack_ofs;
//        it != literal_stack_.end(); it++) {
//      for (auto cl_ofs : occurrence_lists_[it->neg()])
//        if (!isSatisfied(cl_ofs)) {
//          for (auto lt = beginOf(cl_ofs); *lt != SENTINEL_LIT; lt++)
//            if (isActive(*lt) && !viewed_lits[lt->neg()]) {
//              test_lits.push_back(lt->neg());
//              viewed_lits[lt->neg()] = true;
//
//            }
//        }
//    }
//
//    stack_ofs = literal_stack_.size();
//    for (auto jt = test_lits.begin(); jt != test_lits.end(); jt++)
//      viewed_lits[*jt] = false;
//
//    statistics_.num_failed_literal_tests_ += test_lits.size();
//
//    for (auto lit : test_lits)
//      if (isActive(lit)) {
//        unsigned sz = literal_stack_.size();
//        // we increase the decLev artificially
//        // s.t. after the tentative BCP call, we can learn a conflict clause
//        // relative to the assn_ of *jt
//        stack_.startFailedLitTest();
//        setLiteralIfFree(lit);
```

(continues on next page)

(continued from previous page)

```
//
//        assert(!hasAntecedent(lit));
//
//        bool bSucceeded = BCP(sz);
//        if (!bSucceeded)
//          recordAllUIPCauses();
//
//        stack_.stopFailedLitTest();
//
//        while (literal_stack_.size() > sz) {
//          unSet(literal_stack_.back());
//          literal_stack_.pop_back();
//        }
//
//        if (!bSucceeded) {
//          statistics_.num_failed_literals_detected_++;
//          sz = literal_stack_.size();
//          for (auto it = uip_clauses_.rbegin(); it != uip_clauses_.rend();
//              it++) {
//            setLiteralIfFree(it->front(), addUIPConflictClause(*it));
//          }
//          if (!BCP(sz))
//            return false;
//        }
//      }
//  }
//  return true;
//}

// this is IBCP 30.08
bool Solver::implicitBCP() {
  static std::vector<LiteralID> test_lits(num_variables());
  static LiteralIndexedVector<unsigned char> viewed_lits(num_variables() + 1, 0);

  VariableIndex stack_ofs = stack_.top().literal_stack_ofs();
  while (stack_ofs < literal_stack_.size()) {
    test_lits.clear();
    for (auto it = literal_stack_.begin() + stack_ofs;
         it != literal_stack_.end(); it++) {
      for (auto cl_ofs : occurrence_lists_[it->neg()])
        if (!isSatisfied(cl_ofs)) {
          for (auto lt = beginOf(cl_ofs); *lt != SENTINEL_LIT; lt++)
            if (isActive(*lt) && !viewed_lits[lt->neg()]) {
              test_lits.push_back(lt->neg());
              viewed_lits[lt->neg()] = true;
            }
        }
    }
    VariableIndex num_curr_lits = literal_stack_.size() - stack_ofs;
    stack_ofs = literal_stack_.size();
    for (auto jt = test_lits.begin(); jt != test_lits.end(); jt++)
      viewed_lits[*jt] = false;

    std::vector<float> scores;
    scores.clear();
    for (auto &test_lit : test_lits) {
      scores.push_back(literal(test_lit).activity_score_);
```

(continues on next page)

(continued from previous page)

```
    }
    sort(scores.begin(), scores.end());
    num_curr_lits = 10 + num_curr_lits / 20;
    float threshold = 0.0;
    if (scores.size() > num_curr_lits) {
      threshold = scores[scores.size() - num_curr_lits];
    }

    statistics_.num_failed_literal_tests_ += test_lits.size();

    for (auto lit : test_lits)
      if (isActive(lit) && threshold <= literal(lit).activity_score_) {
        VariableIndex sz = literal_stack_.size();
        // we increase the decLev artificially
        // s.t. after the tentative BCP call, we can learn a conflict clause
        // relative to the assn_ of *jt
        stack_.startFailedLitTest();
        setLiteralIfFree(lit);

        assert(!hasAntecedent(lit));

        bool bSucceeded = BCP(sz);
        if (!bSucceeded)
          recordAllUIPCauses();

        stack_.stopFailedLitTest();

        while (literal_stack_.size() > sz) {
          unSet(literal_stack_.back());
          literal_stack_.pop_back();
        }

        if (!bSucceeded) {
          statistics_.num_failed_literals_detected_++;
          sz = literal_stack_.size();
          for (auto it = uip_clauses_.rbegin();
               it != uip_clauses_.rend(); it++) {
            // DEBUG
            if (it->empty() && !config_.quiet)
              PrintError("EMPTY CLAUSE FOUND");
            // END DEBUG
            setLiteralIfFree(it->front(), addUIPConflictClause(*it));
          }
          if (!BCP(sz))
            return false;
        }
      }
  }

  // BEGIN TEST
//  float max_score = -1;
//  float score;
//  unsigned max_score_var = 0;
//  for (auto it =
//       component_analyzer_.superComponentOf(stack_.top()).varsBegin();
//       *it != varsSENTINEL; it++)
//    if (isActive(*it)) {
```

(continues on next page)

(continued from previous page)

```
//      score = scoreOf(*it);
//      if (score > max_score) {
//        max_score = score;
//        max_score_var = *it;
//      }
//    }
//  LiteralID theLit(max_score_var,
//      literal(LiteralID(max_score_var, true)).activity_score_
//          > literal(LiteralID(max_score_var, false)).activity_score_);
//  if (!fail_test(theLit.neg())) {
//    std::cout << ".";
//
//    statistics_.num_failed_literals_detected_++;
//    unsigned sz = literal_stack_.size();
//    for (auto it = uip_clauses_.rbegin(); it != uip_clauses_.rend(); it++) {
//      setLiteralIfFree(it->front(), addUIPConflictClause(*it));
//    }
//    if (!BCP(sz))
//      return false;
//
//  }
  // END
  return true;
}

// BEGIN module conflictAnalyzer

void Solver::minimizeAndStoreUIPClause(LiteralID uipLit,
                                       std::vector<LiteralID> &tmp_clause,
                                       const bool seen[]) {
  static std::deque<LiteralID> clause;
  clause.clear();
  assertion_level_ = 0;
  for (auto lit : tmp_clause) {
    if (existsUnitClauseOf(lit.var()))
      continue;
    bool resolve_out = false;
    if (hasAntecedent(lit)) {
      resolve_out = true;
      if (getAntecedent(lit).isAClause()) {
        for (auto it = beginOf(getAntecedent(lit).asCl()) + 1;
             *it != SENTINEL_CL; it++)
          if (!seen[it->var()]) {
            resolve_out = false;
            break;
          }
      } else if (!seen[getAntecedent(lit).asLit().var()]) {
        resolve_out = false;
      }
    }

    if (!resolve_out) {
      // uipLit should be the sole literal of this Decision Level
      if (var(lit).decision_level >= assertion_level_) {
        assertion_level_ = var(lit).decision_level;
        clause.push_front(lit);
      } else {
```

(continues on next page)

(continued from previous page)

```
        clause.push_back(lit);
      }
    }
  }

  if (uipLit.var())
    assert(var_const(uipLit).decision_level == stack_.get_decision_level());

//  assert(uipLit.var() != 0);
  if (uipLit.var() != 0)
    clause.push_front(uipLit);
  uip_clauses_.emplace_back(std::vector<LiteralID>(clause.begin(), clause.end()));
}

void Solver::recordLastUIPCauses() {
// note:
// variables of lower dl: if seen we dont work with them anymore
// variables of this dl: if seen we incorporate their
// antecedent and set to unseen
  bool seen[num_variables() + 1];
  memset(seen, false, sizeof(bool) * (num_variables() + 1));

  static std::vector<LiteralID> tmp_clause;
  tmp_clause.clear();

  assertion_level_ = 0;
  uip_clauses_.clear();

  unsigned long lit_stack_ofs = literal_stack_.size();
  long DL = stack_.get_decision_level();
  unsigned lits_at_current_dl = 0;

  for (auto l : violated_clause) {
    if (var(l).decision_level == 0 || existsUnitClauseOf(l.var()))
      continue;
    if (var(l).decision_level < DL)
      tmp_clause.push_back(l);
    else
      lits_at_current_dl++;
    literal(l).increaseActivity();
    seen[l.var()] = true;
  }

  LiteralID curr_lit;
  while (lits_at_current_dl) {
    assert(lit_stack_ofs != 0);
    curr_lit = literal_stack_[--lit_stack_ofs];

    if (!seen[curr_lit.var()])
      continue;

    seen[curr_lit.var()] = false;

    if (lits_at_current_dl-- == 1) {
      // perform UIP stuff
      if (!hasAntecedent(curr_lit)) {
        // this should be the decision literal when in first branch
```

(continues on next page)

(continued from previous page)

```
        // or it is a literal decided to explore in failed literal testing
//        assert(stack_.TOS_decLit() == curr_lit);
//        std::cout << "R" << curr_lit.toInt() << "S"
//                  << var(curr_lit).ante.isAnt() << " "  << std::endl;
        break;
      }
    }

    assert(hasAntecedent(curr_lit));

//    std::cout << "{" << curr_lit.toInt() << "}";
    if (getAntecedent(curr_lit).isAClause()) {
      updateActivities(getAntecedent(curr_lit).asCl());
      assert(curr_lit == *beginOf(antecedent(curr_lit).asCl()));

      for (auto it = beginOf(getAntecedent(curr_lit).asCl()) + 1;
           *it != SENTINEL_CL; it++) {
        if (seen[it->var()] || (var(*it).decision_level == 0)
            || existsUnitClauseOf(it->var()))
          continue;
        if (var(*it).decision_level < DL)
          tmp_clause.push_back(*it);
        else
          lits_at_current_dl++;
        seen[it->var()] = true;
      }
    } else {
      LiteralID alit = getAntecedent(curr_lit).asLit();
      literal(alit).increaseActivity();
      literal(curr_lit).increaseActivity();
      if (!seen[alit.var()] && var(alit).decision_level != 0
          && !existsUnitClauseOf(alit.var())) {
        if (var(alit).decision_level < DL)
          tmp_clause.push_back(alit);
        else
          lits_at_current_dl++;
        seen[alit.var()] = true;
      }
    }
    curr_lit = NOT_A_LIT;
  }

//  std::cout << "T" << curr_lit.toInt() << "U "
//            << var(curr_lit).decision_level << ", " << stack_.get_decision_level()
//            << "\n"
//            << "V"  << var(curr_lit).ante.isAnt() << " "  << std::endl;
  minimizeAndStoreUIPClause(curr_lit.neg(), tmp_clause, seen);

//  if (var(curr_lit).decision_level > assertion_level_)
//    assertion_level_ = var(curr_lit).decision_level;
}

void Solver::recordAllUIPCauses() {
// note:
// variables of lower dl: if seen we dont work with them anymore
// variables of this dl: if seen we incorporate their
// antecedent and set to unseen
```

(continues on next page)

(continued from previous page)

```
  bool seen[num_variables() + 1];
  memset(seen, false, sizeof(bool) * (num_variables() + 1));

  static std::vector<LiteralID> tmp_clause;
  tmp_clause.clear();

  assertion_level_ = 0;
  uip_clauses_.clear();

  unsigned long lit_stack_ofs = literal_stack_.size();
  long DL = stack_.get_decision_level();
  unsigned lits_at_current_dl = 0;

  for (auto l : violated_clause) {
    if (var(l).decision_level == 0 || existsUnitClauseOf(l.var()))
      continue;
    if (var(l).decision_level < DL)
      tmp_clause.push_back(l);
    else
      lits_at_current_dl++;
    literal(l).increaseActivity();
    seen[l.var()] = true;
  }
  unsigned n = 0;
  LiteralID curr_lit;
  while (lits_at_current_dl) {
    assert(lit_stack_ofs != 0);
    curr_lit = literal_stack_[--lit_stack_ofs];

    if (!seen[curr_lit.var()])
      continue;

    seen[curr_lit.var()] = false;

    if (lits_at_current_dl-- == 1) {
      n++;
      if (!hasAntecedent(curr_lit)) {
        // this should be the decision literal when in first branch
        // or it is a literal decided to explore in failed literal testing
        //assert(stack_.TOS_decLit() == curr_lit);
        break;
      }
      // perform UIP stuff
      minimizeAndStoreUIPClause(curr_lit.neg(), tmp_clause, seen);
    }

    assert(hasAntecedent(curr_lit));

    if (getAntecedent(curr_lit).isAClause()) {
      updateActivities(getAntecedent(curr_lit).asCl());
      assert(curr_lit == *beginOf(getAntecedent(curr_lit).asCl()));

      for (auto it = beginOf(getAntecedent(curr_lit).asCl()) + 1;
           *it != SENTINEL_CL; it++) {
        if (seen[it->var()] || (var(*it).decision_level == 0)
            || existsUnitClauseOf(it->var()))
          continue;
```

(continues on next page)

(continued from previous page)

```
        if (var(*it).decision_level < DL)
          tmp_clause.push_back(*it);
        else
          lits_at_current_dl++;
        seen[it->var()] = true;
      }
    } else {
      LiteralID alit = getAntecedent(curr_lit).asLit();
      literal(alit).increaseActivity();
      literal(curr_lit).increaseActivity();
      if (!seen[alit.var()] && var(alit).decision_level != 0
          && !existsUnitClauseOf(alit.var())) {
        if (var(alit).decision_level < DL)
          tmp_clause.push_back(alit);
        else
          lits_at_current_dl++;
        seen[alit.var()] = true;
      }
    }
  }
  if (!hasAntecedent(curr_lit)) {
    minimizeAndStoreUIPClause(curr_lit.neg(), tmp_clause, seen);
  }
//  if (var(curr_lit).decision_level > assertion_level_)
//    assertion_level_ = var(curr_lit).decision_level;
}

void Solver::printOnlineStats() {
  if (config_.quiet)
    return;

  std::cout << "\ntime elapsed: " << sampler_stopwatch_.getElapsedSeconds() << "s" <<␣
↪std::endl;
  if (config_.verbose) {
    std::cout << "conflict clauses (all / bin / unit) \t"
              << num_conflict_clauses()
              << "/" << statistics_.num_binary_conflict_clauses_ << "/"
              << unit_clauses_.size() << "\n"
              << "failed literals found by implicit BCP \t "
              << statistics_.num_failed_literals_detected_ << "\n";

    std::cout << "implicit BCP miss rate \t "
              << statistics_.implicitBCP_miss_rate() * 100 << "%\n";

    comp_manager_.gatherStatistics();

    std::cout << "cache size " << statistics_.cache_MB_memory_usage() << "MB" << "\n"
              << "components (stored / hits) \t\t"
              << statistics_.cached_component_count() << "/"
              << statistics_.cache_hits() << "\n"
              << "avg. variable count (stored / hits) \t"
              << statistics_.getAvgComponentSize() << "/"
              << statistics_.getAvgCacheHitSize() << "\n"
              << "cache miss rate " << statistics_.cache_miss_rate() * 100 << "%"
              << std::endl;
  }
}
```

(continues on next page)

(continued from previous page)

```
//bool Solver::IsVarInLiteralStack(const VariableIndex var) {
//  for (VariableIndex i = 0; i < literal_stack_.size(); i++) {
//    if (literal_stack_[i].var() == var) {
//      std::stringstream ss;
//      ss << "Var #" << var << " is in the literal stack at level " << i
//          << " with val " << literal_stack_[i].sign();
//      PrintError(ss);
//      return true;
//    }
//  }
//  return false;
//}
//
//
//void Solver::PrintLiteralStackLocation(VariableIndex var) {
//  for (VariableIndex i = 0; i < literal_stack_.size(); i++) {
//    if (literal_stack_[i].var() == var) {
//      std::stringstream ss;
//      ss << "Var #" << var << " is located in slot " << i << " and has sign "
//         << (literal_stack_[i].sign() ? "POS" : "NEG");
//      PrintError(ss);
//    }
//  }
//}

Solver::Solver(int argc, char *argv[]) : final_samples_(0, config_) {
  // Store the configuration for visibility by the stack.
  LinkConfigAndStatistics();

  time(&statistics_.start_time_);
  config_.num_samples_ = 0;  // By default initialize the model count to zero
  for (int i = 1; i < argc; i++) {
//    if (strcmp(argv[i], "-noCC") == 0)
//      config_.perform_component_caching = false;
//    else if (strcmp(argv[i], "-noIBCP") == 0)
//      config_.perform_failed_lit_test = false;
//    else if (strcmp(argv[i], "-noPP") == 0)
//      config_.perform_pre_processing = false;
//    else if (strcmp(argv[i], "-q") == 0)
    if (strcmp(argv[i], "-q") == 0) {
      config_.quiet = true;
    } else if (strcmp(argv[i], "-v") == 0) {
      config_.verbose = true;
    } else if (strcmp(argv[i], "-d") == 0) {
      config_.debug_mode = true;
    } else if (strcmp(argv[i], "-tp") == 0) {
      config_.perform_two_pass_sampling_ = true;
    } else if (strcmp(argv[i], "-s") == 0 || strcmp(argv[i], "-out") == 0) {
      if (argc <= i + 1 || !config_.perform_random_sampling_)
        ExitInvalidParam("Invalid parameters for sampling");
      if (strcmp(argv[i], "-s") == 0) {
        long num_samples = strtoul(argv[++i], nullptr, STR_DECIMAL_BASE);
        config_.num_samples_ = static_cast<SampleSize>(num_samples);
        if (config_.num_samples_ < 1)
          ExitInvalidParam("Must sample at least one model");
```

(continues on next page)

(continued from previous page)

```
      } else {
        config_.samples_output_file = argv[++i];
      }
    } else if (strcmp(argv[i], "-no-sample-write") == 0) {
      config_.disable_samples_write_ = true;
    } else if (strcmp(argv[i], "-count-only") == 0) {
      config_.perform_random_sampling_ = false;
      if (config_.num_samples_ > 0 || !config_.samples_output_file.empty())
        ExitInvalidParam("Invalid parameters for counting and sampling");
    } else if (strcmp(argv[i], "-t") == 0) {
      if (argc <= i + 1)
        ExitInvalidParam("Time bound missing");
      config_.time_bound_seconds = strtoul(argv[++i], nullptr, STR_DECIMAL_BASE);
    } else if (strcmp(argv[i], "-cs") == 0) {
      if (argc <= i + 1)
        ExitInvalidParam("No cache size specified");
      statistics_.maximum_cache_size_bytes_ = strtoul(argv[++i], nullptr, STR_DECIMAL_
↪BASE)
                                              * (uint64_t) 1000000;
    } else if (strcmp(argv[i], "-cnf") == 0) {
      if (argc <= i + 1)
        ExitInvalidParam("No CNF file specified");
      statistics_.input_file_ = argv[++i];
    } else if (strcmp(argv[i], "-no-partial-fill") == 0) {
      // This is a debug only feature. Functionality is not guaranteed.
      config_.skip_partial_assignment_fill = true;
    } else if (strcmp(argv[i], "-top-tree") == 0) {
      config_.perform_top_tree_sampling = true;
    } else if (strcmp(argv[i], "-top-tree-depth") == 0) {
      long top_tree_depth = strtoul(argv[++i], nullptr, STR_DECIMAL_BASE);
      if (top_tree_depth <= 1)
        ExitInvalidParam("The top tree depth must be greater than 1.");
      config_.max_top_tree_depth_ = static_cast<TreeNodeIndex>(top_tree_depth);
    } else if (strcmp(argv[i], "-max-leaf-size") == 0) {
      long max_leaf_size = strtoul(argv[++i], nullptr, STR_DECIMAL_BASE);
      if (max_leaf_size <= 1)
        ExitInvalidParam("The top tree leaf size must be greater than 1.");
      config_.max_top_tree_leaf_sample_count = static_cast<SampleSize>(max_leaf_size);
    } else {
      ExitInvalidParam(static_cast<std::string>("Unknown parameter found \"") +␣
↪argv[i] + "\"");
    }
  }

  // Perform additional cross-checking of input parameters
  if (config_.quiet && config_.verbose)
    ExitInvalidParam("Invalid combination of verbose and quiet");
  if (config_.num_samples_ < 1 && config_.perform_random_sampling_)
    ExitInvalidParam("If sampling is enabled, a sample count greater than or equal \n"
                     "to 1 must be specified.");
  if (config_.perform_two_pass_sampling_ && !config_.perform_random_sampling_)
    ExitInvalidParam("The two pass sampling flag is only applicable when sampling is␣
↪enabled.\n");
  if (config_.perform_top_tree_sampling && !config_.perform_random_sampling_)
    ExitInvalidParam("Top tree sampling cannot be enabled if random sampling is␣
↪disabled.");
  if (config_.disable_samples_write_ && !config_.perform_random_sampling_)
```

(continues on next page)

(continued from previous page)

```
    ExitInvalidParam("Disabling sample write cannot be selected if sampling is␣
↪disabled.");

  if (config_.samples_output_file.empty() && config_.perform_random_sampling_) {
    std::string input_file = statistics_.input_file_;
    uint64_t os_sep_loc = input_file.rfind('\\');
    os_sep_loc = (os_sep_loc != std::string::npos) ? os_sep_loc : input_file.rfind('/
↪');

    std::string out_file;
    if (os_sep_loc != std::string::npos)
      out_file = statistics_.input_file_.substr(0, os_sep_loc + 1);
    // Prepend "results_" to the samples filename
    out_file += "samples_";

    // Append the filename and change extension to ".txt"
    uint64_t filename_start_loc = os_sep_loc + 1;
    uint64_t file_ext_start = input_file.rfind('.');
    if (file_ext_start <= filename_start_loc || file_ext_start == std::string::npos)
      file_ext_start = statistics_.input_file_.size();
    uint64_t filename_len = file_ext_start - filename_start_loc;
    out_file += statistics_.input_file_.substr(filename_start_loc, filename_len);
    out_file += ".txt";
    config_.samples_output_file = out_file;
    if (!config_.quiet)
      PrintWarning(static_cast<std::string>("No sample results file specified.\n")
                   + "Using default filename: \"" + out_file + "\"");
  }
  if (config_.perform_two_pass_sampling_ && config_.num_samples_ > 1 && !config_.
↪quiet)
    PrintWarning("The two pass sampling flag only has an effect\n"
                     "when the number of samples equals 1. Ignoring...");

  // Initialize the time bound.
  sampler_stopwatch_.setTimeBound(config_.time_bound_seconds);

  // Any time the same count is larger than one, always do two pass sampling
  if (config_.num_samples_ > 1)
    config_.perform_two_pass_sampling_ = true;
  // Must parse at the end in case debug mode is selected.
  Random::init(&config_);
}
```

Includes

- deque
- primitive_types.h (*File primitive_types.h*)
- sampler_tools.h (*File sampler_tools.h*)
- solver.h (*File solver.h*)
- solver_config.h (*File solver_config.h*)
- sstream
- string

- unordered_map
- utility
- vector

File solver.h

Parent directory (src)

Contents

- *Definition (src/solver.h)*
- *Includes*
- *Included By*
- *Classes*

Definition (src/solver.h)

Program Listing for File solver.h

Return to documentation for file (src/solver.h)

```
#ifndef SOLVER_H_
#define SOLVER_H_

#include <sys/time.h>

#include <algorithm>
#include <utility>
#include <vector>
#include <string>

#include "statistics.h"
#include "instance.h"
#include "component_management.h"

#include "solver_config.h"
#include "model_sampler.h"
#include "cached_assignment.h"
//#include "top_tree_sampler.h"


class StopWatch {
 public:
  StopWatch();
  bool timeBoundBroken() {
    timeval actual_time = (struct timeval) {0};
    gettimeofday(&actual_time, nullptr);
    return actual_time.tv_sec - start_time_.tv_sec > time_bound_;
  }
```

(continues on next page)

(continued from previous page)

```
  bool start() {
    auto ret = static_cast<bool>(gettimeofday(&last_interval_start_, nullptr));
    start_time_ = stop_time_ = last_interval_start_;
    return !ret;
  }
  bool stop() {
    return gettimeofday(&stop_time_, nullptr) == 0;
  }

  double getElapsedSeconds() {
    timeval r = getElapsedTime();
    return r.tv_sec + static_cast<double>(r.tv_usec) / 1000000;
  }

  bool interval_tick() {
    timeval actual_time = (struct timeval) {0};
    gettimeofday(&actual_time, nullptr);
    if (actual_time.tv_sec - last_interval_start_.tv_sec
        > interval_length_.tv_sec) {
      gettimeofday(&last_interval_start_, nullptr);
      return true;
    }
    return false;
  }

  void setTimeBound(uint64_t seconds) {
    time_bound_ = seconds;
  }
  // ZH Appears to neither be implemented or used.
  //long int getTimeBound();

 private:
  timeval start_time_;
  timeval stop_time_;

  uint64_t time_bound_ = UINT64_MAX;

  timeval interval_length_;
  timeval last_interval_start_;

  timeval getElapsedTime();
};

class Solver: public Instance {
 public:
  Solver(int argc, char *argv[]);

  Solver(SolverConfiguration &config, DataAndStatistics &statistics,
         SampleSize num_samples = 0, bool two_pass = true)
      : config_(config), final_samples_(num_samples, config_) {
    LinkConfigAndStatistics();

    statistics_.input_file_ = statistics.input_file_;

    // ToDo modify to support caching multiple samples
    //config_.num_samples_to_cache_ = num_samples;
    config_.num_samples_to_cache_ = 1;
```

(continues on next page)

(continued from previous page)

```
    sampler_stopwatch_.setTimeBound(config_.time_bound_seconds);  // Initialize the␣
↪time bound.
    config_.samples_output_file = "";

    config_.disable_samples_write_ = true;
    config_.num_samples_ = num_samples;
    time(&statistics_.start_time_);
    config_.perform_two_pass_sampling_ = two_pass;
  }
  Solver(const Solver &other)
      : config_(other.config_), final_samples_(other.final_samples_.num_samples(),␣
↪config_) {
    // Bring over the configuration and statistics
    statistics_ = other.statistics_;
    LinkConfigAndStatistics();
    config_.num_samples_to_cache_ = 1;
    config_.disable_samples_write_ = true;

    // Data structures used in the createFromFile Function
    literal_pool_ = other.literal_pool_;
    variables_ = other.variables_;
    literal_values_ = other.literal_values_;
    unit_clauses_ = other.unit_clauses_;
    unused_vars_ = other.unused_vars_;

    independent_support = other.independent_support;
    has_independent_support_ = other.has_independent_support_;
    conflict_clauses_ = other.conflict_clauses_;
    occurrence_lists_ = other.occurrence_lists_;
    literals_ = other.literals_;

    // Initialize other variables as needed
    statistics_.start_time_ = other.statistics_.start_time_;  // Needed to ensure␣
↪timeout
    sampler_stopwatch_.setTimeBound(config_.time_bound_seconds);  // Initialize the␣
↪time bound.
    original_lit_pool_size_ = other.original_lit_pool_size_;
  }
  void solve(const PartialAssignment &partial_assn = PartialAssignment());
  void sample_models();
  const SolverConfiguration &config() {
    return config_;
  }
  void DisableTopTreeSampling() {
    assert(config_.perform_top_tree_sampling);
    config_.perform_top_tree_sampling = false;
  }
  const DataAndStatistics &statistics() {
    return statistics_;
  }
  void PushNewSamplesManagerOnStack() {
    if (config_.store_sampled_models())
      samples_stack_.emplace_back(SamplesManager(config_.num_samples_, config_));
  }
//  /**
//   * Literal Stack Location Printer
//   *
```

(continues on next page)

(continued from previous page)

```
//    * Debug only tool.  Often, a CNF formula has hundreds of thousands
//    * of variables on the literal stack.  Finding where a specific variable
//    * is on the stack (if at all) can be time consuming.  This function
//    * prints whether the specified variable is on the stack.
//    *
//    * @param var Variable number of interest.
//    */
//  void PrintLiteralStackLocation(VariableIndex var);

  //  End Sampler Objects

 private:
  SolverConfiguration config_;
  DecisionStack stack_;
  std::vector<LiteralID> literal_stack_;
  ComponentManager comp_manager_ = ComponentManager(config_, statistics_, literal_
↪values_);
  StopWatch sampler_stopwatch_;
  uint64_t last_ccl_deletion_time_ = 0;
  uint64_t last_ccl_cleanup_time_ = 0;
  std::vector<SamplesManager> samples_stack_;
  SamplesManager final_samples_;
  bool simplePreProcess();

  bool prepFailedLiteralTest();
  void reservoir_sample_models(const PartialAssignment &partial_assn, Solver &temp_
↪solver);
//  /**
//    * MT - we assert that the formula is consistent and has not been
//    * found UNSAT yet hard wires all assertions in the literal stack
//    * into the formula removes all set variables and essentially
//    * reinitializes all further data.
//    */
//  void HardWireAndCompact();
  SolverExitState countSAT();
  void decideLiteral();
  bool bcp();
  void decayActivitiesOf(Component & comp) {
    for (auto it = comp.varsBegin(); *it != varsSENTINEL; it++) {
      literal(LiteralID(*it, true)).activity_score_ *=0.5;
      literal(LiteralID(*it, false)).activity_score_ *=0.5;
    }
  }
  bool implicitBCP();
  bool BCP(VariableIndex start_at_stack_ofs);
  SolverNextAction backtrack();
  void ProcessCacheStore();
  void ProcessSampleComponentSplitBacktrack();
  SolverNextAction resolveConflict();
  void PrintFinalSamplerResults();

  //  BEGIN small helper functions

  float scoreOf(VariableIndex v) {
    float score = comp_manager_.scoreOf(v);
    score += 10.0 * literal(LiteralID(v, true)).activity_score_;
    score += 10.0 * literal(LiteralID(v, false)).activity_score_;
```

(continues on next page)

(continued from previous page)

```
//    score += (10*stack_.get_decision_level()) * literal(LiteralID(v, true)).
↪activity_score_;
//    score += (10*stack_.get_decision_level()) * literal(LiteralID(v, false)).
↪activity_score_;
    return score;
  }
  bool setLiteralIfFree(LiteralID lit, Antecedent ant = Antecedent(NOT_A_CLAUSE)) {
    if (literal_values_[lit] != X_TRI)
      return false;
    var(lit).decision_level = stack_.get_decision_level();
    var(lit).ante = ant;
    literal_stack_.push_back(lit);
    if (ant.isAClause() && ant.asCl() != NOT_A_CLAUSE)
      getHeaderOf(ant.asCl()).increaseScore();
    literal_values_[lit] = T_TRI;
    literal_values_[lit.neg()] = F_TRI;
    return true;
  }
  void printOnlineStats();
//  /**
//   * Literal Stack Contents Checker
//   *
//   * Checks whether the specified variable is in the literal stack.
//   *
//   * @param var Variable of interest
//   * @return true if the specified variable @see var is in the literal stack.
//   */
//  bool IsVarInLiteralStack(VariableIndex var);

  void setConflictState(LiteralID litA, LiteralID litB) {
    violated_clause.clear();
    violated_clause.push_back(litA);
    violated_clause.push_back(litB);
  }

  void setConflictState(ClauseOfs cl_ofs) {
    getHeaderOf(cl_ofs).increaseScore();
    violated_clause.clear();
    for (auto it = beginOf(cl_ofs); *it != SENTINEL_LIT; it++)
      violated_clause.push_back(*it);
  }

  std::vector<LiteralID>::const_iterator TOSLiteralsBegin() {
    return literal_stack_.begin() + stack_.top().literal_stack_ofs();
  }
  void initStack(unsigned long resSize = 0) {
    stack_.clear();
    if (resSize != 0)
      stack_.reserve(resSize);
    literal_stack_.clear();
    if (resSize != 0)
      literal_stack_.reserve(resSize);
    // initialize the stack to contain at least level zero
    stack_.push_back(StackLevel(1, 0, 2));
    stack_.back().changeBranch();
    // Reset the samples stack.
    if (config_.store_sampled_models()) {
```

(continues on next page)

(continued from previous page)

```
    samples_stack_.clear();
    samples_stack_.reserve(30);
  }
  set_variable_depth_ = 0;
  statistics_.max_component_split_depth_ = 0;
}

const LiteralID &TOS_decLit() const {
  assert(stack_.top_const().literal_stack_ofs() < literal_stack_.size());
  return literal_stack_[stack_.top_const().literal_stack_ofs()];
}
void reactivateTOS() {
  for (auto it = TOSLiteralsBegin(); it != literal_stack_.end(); it++)
    unSet(*it);
  comp_manager_.cleanRemainingComponentsOf(stack_.top());
  literal_stack_.resize(stack_.top().literal_stack_ofs());
  stack_.top().resetRemainingComps();
}

bool fail_test(LiteralID lit) {
  VariableIndex sz = literal_stack_.size();
  // we increase the decLev artificially
  // s.t. after the tentative BCP call, we can learn a conflict clause
  // relative to the assn_ of *jt
  stack_.startFailedLitTest();
  setLiteralIfFree(lit);

  assert(!hasAntecedent(lit));

  bool bSucceeded = BCP(sz);
  if (!bSucceeded)
    recordAllUIPCauses();

  stack_.stopFailedLitTest();

  while (literal_stack_.size() > sz) {
    unSet(literal_stack_.back());
    literal_stack_.pop_back();
  }
  return bSucceeded;
}
//  BEGIN conflict analysis

// if the state name is CONFLICT,
// then violated_clause contains the clause determining the conflict;
std::vector<LiteralID> violated_clause;
// this is an array of all the clauses found
// during the most recent conflict analysis
// it might contain more than 2 clauses
// but always will have:
//      uip_clauses_.front() the 1UIP clause found
//      uip_clauses_.back() the lastUIP clause found
//  possible clauses in between will be other UIP clauses
std::vector<std::vector<LiteralID>> uip_clauses_;

// the assertion level of uip_clauses_.back()
// or (if the decision variable did not have an antecedent
```

(continues on next page)

(continued from previous page)

```
  // before) then assertionLevel_ == DL;
  int64_t assertion_level_ = 0;

  // build conflict clauses from most recent conflict
  // as stored in state_.violated_clause
  // solver state must be CONFLICT to work;
  // this first method record only the last UIP clause
  // so as to create clause that asserts the current decision
  // literal
  void recordLastUIPCauses();
  void recordAllUIPCauses();

  void minimizeAndStoreUIPClause(LiteralID uipLit,
                                 std::vector<LiteralID> & tmp_clause,
                                 const bool seen[]);

  // Commented out by ZH as not implemented
//  void storeUIPClause(LiteralID uipLit, std::vector<LiteralID> & tmp_clause);

//  int getAssertionLevel() const {
//    return assertion_level_;
//  }

  //  END conflict analysis

  VariableIndex set_variable_depth_ = 0;
  bool InitializeSolverAndPreprocess(const PartialAssignment &partial_assn);

  void applyPartialAssignment(const PartialAssignment &partial_assn);
  void ReportSharpSatResults();
  void PerformInitialSampling();
  inline void FillPartialAssignments(Solver &temp_solver);
  inline bool IsEndSamplesStackSizeValid() const {
    return samples_stack_.size() == 1
           || (samples_stack_.size() == 2 && samples_stack_[0].model_count() == 0);
  }
//  /**
//   * Checks whether the current point in the solver execution is valid for storing␣
↪a top
//   * tree sample.
//   *
//   * @return true If there none of the current literal assignments led to a␣
↪component split,
//   * the current depth is below the maximum and top tree mode is enabled.
//   */
//  inline bool IsValidTopTreeNodeStorePoint() {
//    return config_.perform_top_tree_sampling && set_variable_depth_ < config_.max_
↪top_tree_depth_;
//  }
  inline void LinkConfigAndStatistics() {
    // ToDo If the solver is multithreaded, static for config and statistics will␣
↪need to change.
    StackLevel::set_solver_config_and_statistics(config_, statistics_);
//    SamplesManager::set_solver_config(config_);
  }
};
```

(continues on next page)

(continued from previous page)

```
#endif // SOLVER_H_
```

Includes

- algorithm
- cached_assignment.h (*File cached_assignment.h*)
- component_management.h (*File component_management.h*)
- instance.h (*File instance.h*)
- model_sampler.h (*File model_sampler.h*)
- solver_config.h (*File solver_config.h*)
- statistics.h (*File statistics.h*)
- string
- sys/time.h
- utility
- vector

Included By

- *File main.cpp*
- *File solver.cpp*

Classes

- *Class Solver*
- *Class StopWatch*

File solver_config.h

Parent directory (src)

Contents

- *Definition (src/solver_config.h)*
- *Includes*
- *Included By*
- *Classes*

Definition (`src/solver_config.h`)

Program Listing for File solver_config.h

Return to documentation for file (`src/solver_config.h`)

```
#ifndef SOLVER_CONFIG_H_
#define SOLVER_CONFIG_H_

#include <cstdlib>
#include <cassert>
#include <string>

#include "primitive_types.h"
#include "sampler_tools.h"

struct SolverConfiguration {
  // Support for these features is removed in the sampler.
//  /**
//   * Support to modify the default state of this feature is
//   * disabled in the sampler.
//   *
//   * Forces variable backtracking in the case of a conflict
//   * being founded.
//   */
//  bool perform_non_chron_back_track = true;
  bool perform_component_caching = true;
  bool perform_failed_lit_test = true;
  bool perform_pre_processing = true;
  bool perform_random_sampling_ = true;
  bool perform_top_tree_sampling = false;
  bool disable_samples_write_ = false;
  bool store_sampled_models() {
    return perform_random_sampling_ && !perform_top_tree_sampling;
  }
  std::string samples_output_file = "";
  bool debug_mode = false;
  uint64_t time_bound_seconds = UINT64_MAX;
  bool verbose = false;
  bool quiet = false;
  bool perform_sample_caching() {
    assert(perform_random_sampling_ || !perform_sample_caching_);
//    return perform_random_sampling_ && perform_sample_caching_;
    return perform_sample_caching_;
  }
  void EnableSampleCaching() {
    assert(perform_random_sampling_);
    perform_sample_caching_ = true;
  }
  bool perform_two_pass_sampling_ = false;
  unsigned num_samples_to_cache_ = 1;
  bool skip_partial_assignment_fill = false;
  SampleSize num_samples_ = 0;
  TreeNodeIndex max_top_tree_depth_ = 25;
  SampleSize max_top_tree_leaf_sample_count = 50;
  std::string top_tree_samples_output_file_ = "." FILE_PATH_SEPARATOR
                                              "__final_top_tree_samples.txt";
```

(continues on next page)

(continued from previous page)

```
 private:
  bool perform_sample_caching_ = false;
};

#endif /* SOLVER_CONFIG_H_ */
```

Includes

- cassert
- cstdlib
- primitive_types.h (*File primitive_types.h*)
- sampler_tools.h (*File sampler_tools.h*)
- string

Included By

- *File statistics.h*
- *File sampler_tools.h*
- *File model_sampler.h*
- *File rand_distributions.h*
- *File stack.h*
- *File component_cache.h*
- *File component_management.h*
- *File solver.h*
- *File solver.cpp*

Classes

- *Struct SolverConfiguration*

File stack.cpp

Parent directory (src)

Contents

- *Definition (src/stack.cpp)*
- *Includes*

Definition (`src/stack.cpp`)

Program Listing for File stack.cpp

Return to documentation for file (`src/stack.cpp`)

```
#include <string>
#include "stack.h"

SolverConfiguration* StackLevel::config_ = nullptr;
DataAndStatistics* StackLevel::statistics_ = nullptr;
VariableIndex StackLevel::component_split_depth_ = 0;


void StackLevel::includeSolutionVar(const mpz_class &solutions, mpz_class solution_
↪var[],
                                    const std::string &name_var) {
  if (branch_found_unsat_[active_branch_]) {
    assert(solution_var[active_branch_] == 0);
    return;
  }
  if (solutions == 0)
    branch_found_unsat_[active_branch_] = true;
  if (solution_var[active_branch_] == 0) {
    solution_var[active_branch_] = solutions;
    if (config_->verbose && solutions > 0)
      std::cout << "\t" << name_var << ": Solution count MPZ set to " << solutions <<␣
↪std::endl;
  } else {
    solution_var[active_branch_] *= solutions;
    if (config_->verbose)
      std::cout << "\t" << name_var << ": Solution count MPZ multiplied by "
                << solutions << " for a product of " << solution_var[active_branch_] <
↪< std::endl;
  }
}

SampleAssignment StackLevel::random_cache_sample() const {
  // Only get the sample after both branches explored.
  assert(active_branch_);
  // Must have at least one valid sample to have a random sample.
  assert(branch_model_count_[0] > 0 || branch_model_count_[1] > 0);

  // If either side is UNSAT, make the easy choice
  if (branch_model_count_[0] == 0)
    return cache_sample_[1];
  if (branch_model_count_[1] == 0)
    return cache_sample_[0];

  // Randomly select either the positive or negative side.
  mpz_class rand_mpz, total_count = branch_model_count_[0] + branch_model_count_[1];
//    SamplesManager::get_rand_mpz(total_count, uniform_mpz);
  Random::Mpz::uniform(total_count, rand_mpz);
  if (rand_mpz < branch_model_count_[0])
    return cache_sample_[0];
  else
    return cache_sample_[1];
```

(continues on next page)

(continued from previous page)

```
}

void StackLevel::includeSolution(unsigned solutions) {
  if (branch_found_unsat_[active_branch_]) {
    assert(branch_model_count_[active_branch_] == 0);
    return;
  }
  if (solutions == 0)
    branch_found_unsat_[active_branch_] = true;
  if (branch_model_count_[active_branch_] == 0) {
    branch_model_count_[active_branch_] = solutions;
    if (config_->verbose) {
      std::cout << "\tSolution count set to " << solutions << std::endl;
    }
  } else {
    branch_model_count_[active_branch_] *= solutions;
    if (config_->verbose) {
      std::cout << "\tSolution count multiplied by " << solutions << " for a product␣
↪of "
                << branch_model_count_[active_branch_] << std::endl;
    }
  }
}
```

Includes

- `stack.h` (*File stack.h*)
- `string`

File stack.h

Parent directory (`src`)

Contents

- *Definition (`src/stack.h`)*
- *Includes*
- *Included By*
- *Classes*

Definition (`src/stack.h`)

Program Listing for File stack.h

Return to documentation for file (`src/stack.h`)

```
#ifndef STACK_H_
#define STACK_H_


#include <gmpxx.h>

#include <cassert>
#include <vector>
#include <string>

#include "model_sampler.h"
#include "cached_assignment.h"
#include "solver_config.h"


class StackLevel {
  const VariableIndex super_component_ = 0;
  bool active_branch_ = false;
  const VariableIndex literal_stack_ofs_ = 0;
  bool is_component_split_ = false;
  bool first_component_ = false;
  SampleAssignment cache_sample_[2];
  mpz_class branch_model_count_[2] = {0, 0};
  bool branch_found_unsat_[2] = {false, false};
  void includeSolutionVar(const mpz_class &solutions, mpz_class solution_var[],
                          const std::string &name_var);
  const VariableIndex remaining_components_ofs_ = 0;
  VariableIndex unprocessed_components_end_ = 0;
  static VariableIndex component_split_depth_;

  /*----------------------------------------*
   *       Begin Sampler Variables          *
   *----------------------------------------*/

  mpz_class stack_solution_count_multiplier_[2] = {1, 1};
  std::vector<std::vector<VariableIndex>> stack_emancipated_vars_;
  std::vector<CacheEntryID> cache_comp_ids_[2];
  static SolverConfiguration* config_;
  static DataAndStatistics* statistics_;
  CachedAssignment cached_assn_;

 public:
  bool hasUnprocessedComponents() const {
    assert(unprocessed_components_end_ >= remaining_components_ofs_);
    return unprocessed_components_end_ > remaining_components_ofs_;
  }
  void nextUnprocessedComponent() {
    assert(unprocessed_components_end_ > remaining_components_ofs_);
    unprocessed_components_end_--;
  }
  void resetRemainingComps() {
    unprocessed_components_end_ = remaining_components_ofs_;
  }
  const VariableIndex super_component() const {
    return super_component_;
  }
  const VariableIndex remaining_components_ofs() const {
```

(continues on next page)

(continued from previous page)

```
    return remaining_components_ofs_;
  }
  const VariableIndex unprocessed_components_end() const {
    return unprocessed_components_end_;
  }
  void set_unprocessed_components_end(VariableIndex end) {
    unprocessed_components_end_ = end;
    assert(remaining_components_ofs_ <= unprocessed_components_end_);
  }

  StackLevel() {
    stack_emancipated_vars_.resize(2);
  }

  StackLevel(VariableIndex super_comp, VariableIndex lit_stack_ofs,
             ClauseOfs comp_stack_ofs) :
      super_component_(super_comp),
      literal_stack_ofs_(lit_stack_ofs),
      remaining_components_ofs_(comp_stack_ofs),
      unprocessed_components_end_(comp_stack_ofs) {
    assert(super_comp < comp_stack_ofs);
    stack_emancipated_vars_.resize(2);
  }
  VariableIndex currentRemainingComponent() {
    assert(remaining_components_ofs_ <= unprocessed_components_end_ - 1);
    return unprocessed_components_end_ - 1;
  }
  bool isSecondBranch() {
    return active_branch_;
  }
  void changeBranch() {
    active_branch_ = true;
  }
  bool anotherCompProcessible() {
    return (!branch_found_unsat()) && hasUnprocessedComponents();
  }

  VariableIndex literal_stack_ofs() const {
    return literal_stack_ofs_;
  }
  void includeSolution(const mpz_class &solutions) {
    includeSolutionVar(solutions, branch_model_count_, "branch_model_count");
  }
  void includeSolutionSampleMultiplier(const mpz_class &solutions) {
    includeSolutionVar(solutions, stack_solution_count_multiplier_,
                       "sampler_solution_multiplier");
  }
  void includeSolution(unsigned solutions);
  void configureNewLevel(const StackLevel &top, const DecisionLevel decision_level) {
    // Do not push down any counts at the top of the decision stack when doing␣
↪reservoir sampling
    // This is required in some cases in top tree sampling since the solution count␣
↪at each leaf
    // is required while traversing the tree.
    if (decision_level == 0 && !(config_->perform_top_tree_sampling && !top.
↪isComponentSplit()))
      return;
```

(continues on next page)

(continued from previous page)

```
    // Push down the solution multiplier and freed variable list
    // Make sure to only use the active branch.
    for (auto &multiplier : stack_solution_count_multiplier_)
      multiplier = top.getSamplerSolutionMultiplier();

    for (auto &freed_variables : stack_emancipated_vars_)
      freed_variables = top.emancipated_vars();

    for (auto &cache_comp_ids : cache_comp_ids_)
      cache_comp_ids = top.cached_comp_ids();
  }
  void addFreeVariables(const std::vector<VariableIndex> &freed_vars) {
    int idx = (active_branch_) ? 1 : 0;
    stack_emancipated_vars_[idx].insert(stack_emancipated_vars_[idx].end(),
                                        freed_vars.begin(), freed_vars.end());
  }
  void addCachedCompIds(const std::vector<CacheEntryID> &ids) {
    int idx = (active_branch_) ? 1 : 0;
//    assert(cached_comp_ids_.empty());
    if (ids.empty())
      return;
    cache_comp_ids_[idx].insert(cache_comp_ids_[idx].end(), ids.begin(), ids.end());
  }
  inline bool branch_found_unsat() const {
    return branch_found_unsat_[active_branch_];
  }
  inline void mark_branch_unsat() {
    branch_found_unsat_[active_branch_] = true;
  }
  const mpz_class getTotalModelCount() const {
    return branch_model_count_[0] + branch_model_count_[1];
  }
  const mpz_class& getActiveModelCount() const {
    if (!active_branch_)
      return branch_model_count_[0];
    else
      return branch_model_count_[1];
  }
  const mpz_class& getSamplerSolutionMultiplier() const {
    if (!active_branch_)
      return stack_solution_count_multiplier_[0];
    else
      return stack_solution_count_multiplier_[1];
  }
//  /**
//   * Calculates the total descendent model count for this stack level.  It is equal␣
↪to:
//   *
//   * FalseModelCount * FalseCountMultiplier + TrueModelCount * TrueCountMultiplier
//   *
//   * @return Total descendent model count.
//   */
//  const mpz_class GetDescendentModelCount() const {
//    mpz_class total_model_count = 0;
//    for (int i = 0; i < 2; i++) {
//      if (stack_solution_count_multiplier_[i] == 1)
//        total_model_count += branch_model_count_[i];
```

(continues on next page)

(continued from previous page)

```
//      else
//        total_model_count += branch_model_count_[i] * stack_solution_count_
↪multiplier_[i];
//    }
//    return total_model_count;
//  }
  const bool HasNoDescendentModels() {
    return branch_found_unsat_[0] && branch_found_unsat_[1];
  }
  inline const std::vector<VariableIndex> &emancipated_vars() const {
    if (!active_branch_)
      return stack_emancipated_vars_[0];
    else
      return stack_emancipated_vars_[1];
  }
  inline const std::vector<CacheEntryID> &cached_comp_ids() const {
    if (!active_branch_)
      return cache_comp_ids_[0];
    else
      return cache_comp_ids_[1];
  }
  inline const bool isComponentSplit() const {
    return is_component_split_ && config_->perform_random_sampling_;
  }
  inline void setAsComponentSplit() {
    if (!config_->perform_random_sampling_)
      return;

    if (!is_component_split_) {
      component_split_depth_++;
      if (component_split_depth_ > statistics_->max_component_split_depth_)
        statistics_->max_component_split_depth_ = component_split_depth_;
    }
    is_component_split_ = true;
    first_component_ = true;
  }
  inline void unsetAsComponentSplit() {
    if (!config_->perform_random_sampling_)
      return;

    if (is_component_split_)
      component_split_depth_--;
    assert(component_split_depth_>= 0);
    is_component_split_ = false;
    first_component_ = false;
  }
  inline static VariableIndex componentSplitDepth() {
    return component_split_depth_;
  }
  inline const bool isFirstComponent() const {
    assert(is_component_split_);
    return first_component_ && config_->perform_random_sampling_;
  }
  inline void markFirstComponentComplete() {
    assert(is_component_split_);
    first_component_ = false;
  }
```

(continues on next page)

(continued from previous page)

```
  inline static void set_solver_config_and_statistics(SolverConfiguration &config,
                                                      DataAndStatistics &statistics) {
    config_ = &config;
    statistics_ = &statistics;
  }
  inline void set_cached_assn(CachedAssignment &cached_assn) {
    assert(config_->perform_random_sampling_);
    cached_assn_ = cached_assn;
  }
  inline const CachedAssignment& cached_assn() const { return cached_assn_; }
  inline void ClearCachedAssn() { cached_assn_.clear(); }
  inline void set_cache_sample(const SampleAssignment & cache_sample) {
    if (!active_branch_)
      cache_sample_[0] = cache_sample;
    else
      cache_sample_[1] = cache_sample;
  }
  SampleAssignment random_cache_sample() const;
};

class DecisionStack: public std::vector<StackLevel> {
  VariableIndex failed_literal_test_active = 0;

 public:
  DecisionStack() : std::vector<StackLevel>() {}
  void startFailedLitTest() {
//    failed_literal_test_active = true;  // Thurley's old code. Relies on casting
    failed_literal_test_active = 1;
  }
  void stopFailedLitTest() {
//    failed_literal_test_active = false;  // Thurley's old code. Relies on casting
    failed_literal_test_active = 0;
  }
  // end for implicit BCP
  StackLevel &top() {
    return const_cast<StackLevel&>(top_const());  // Strip off the const
  }
  const StackLevel &top_const() const {
    assert(!empty());
    return back();
  }
  StackLevel &on_deck() {
    assert(size() >= 2);
    return (*this)[size() - 2];
  }
  DecisionLevel get_decision_level() const {
    assert(!empty());
    return size() - 1 + failed_literal_test_active;
  }  // 0 means pre-1st-decision
};

#endif  // STACK_H_
```

Includes

- `cached_assignment.h` (*File cached_assignment.h*)

- cassert
- gmpxx.h
- model_sampler.h (*File model_sampler.h*)
- solver_config.h (*File solver_config.h*)
- string
- vector

Included By

- *File alt_component_analyzer.cpp*
- *File component_cache.h*
- *File component_management.h*
- *File model_sampler.cpp*
- *File stack.cpp*

Classes

- *Class DecisionStack*
- *Class StackLevel*

File statistics.cpp

Parent directory (src)

Contents

- *Definition (src/statistics.cpp)*
- *Includes*

Definition (src/statistics.cpp)

Program Listing for File statistics.cpp

Return to documentation for file (src/statistics.cpp)

```
#include <fstream>
#include <string>
#include "statistics.h"



void DataAndStatistics::printShort() {
  std::cout << "\n\n"
```

(continues on next page)

(continued from previous page)

```
            << "variables (total / active / free)\t" << num_variables_ << "/"
            << num_used_variables_ << "/" << num_variables_ - num_used_variables_
            << "\n"
            << "clauses (removed) \t\t" << num_original_clauses_ << " ("
            << num_original_clauses_ - num_clauses() << ")" << "\n"
            << "decisions \t\t\t\t" << num_decisions_ << "\n"
            << "conflicts \t\t\t\t" << num_conflicts_ << "\n"
            << "conflict clauses (all/bin/unit) \t"
            << num_conflict_clauses()
            << "/" << num_binary_conflict_clauses_ << "/" << num_unit_clauses_
            << "\n"
            << "failed literals found by implicit BCP \t "
            << num_failed_literals_detected_ << "\n";


  std::cout << "implicit BCP miss rate \t" << implicitBCP_miss_rate() * 100 << "%\n"
            << "bytes cache size     \t" << cache_bytes_memory_usage()  << "\t\n";

  std::cout << "bytes cache (overall) \t" << overall_cache_bytes_memory_stored()
            << "\n"
            << "bytes cache (infra / comps) "
            << (cache_infrastructure_bytes_memory_usage_) << "/"
            << sum_bytes_cached_components_  << "\t\n";

  std::cout << "bytes pure comp data (curr)    "
            << sum_bytes_pure_cached_component_data_  << "\n"
            << "bytes pure comp data (overall) "
            << overall_bytes_pure_stored_component_data_ << "\n";

  std::cout << "bytes cache with sysoverh (curr)    "
            << sys_overhead_sum_bytes_cached_components_  << "\n"
            << "bytes cache with sysoverh (overall) "
            << sys_overhead_overall_bytes_components_stored_ << "\n";


  std::cout << "cache (stores / hits) \t\t\t" << num_cached_components_ << "/"
            << num_cache_hits_ << "\n"
            << "cache miss rate \t\t" << cache_miss_rate() * 100 << "%\n"
            << "avg. variable count (stores / hits) \t" << getAvgComponentSize()
            << "/" << getAvgCacheHitSize() << "\n\n"
            << "\n# solutions " << "\n"
            << final_solution_count_.get_str() << "\n"
            << "\n# END\n\n"
            << "time: " << sampler_time_elapsed_ << "s\n\n";
}
```

Includes

- fstream
- statistics.h (*File statistics.h*)
- string

File statistics.h

Parent directory (`src`)

Contents

- *Definition (`src/statistics.h`)*
- *Includes*
- *Included By*
- *Classes*

Definition (`src/statistics.h`)

Program Listing for File statistics.h

Return to documentation for file (`src/statistics.h`)

```
#ifndef STATISTICS_H_
#define STATISTICS_H_

#include <gmpxx.h>

#include <string>
#include <cstdint>
#include <vector>
#include <cfloat>

#include "structures.h"
#include "component_types/cacheable_component.h"

#include "primitive_types.h"
#include "solver_config.h"


class DataAndStatistics {
 public:
  DataAndStatistics() {
    time(&start_time_);
    for (unsigned i = 0; i < static_cast<unsigned>(TopTreeNodeType::NUM_TREE_NODE_
↪TYPES); i++) {
      num_models_by_tree_node_type_[i] = 0;
      num_tree_nodes_by_type_[i] = 0;
    }
  }
  std::string input_file_;
  //double time_elapsed_ = 0.0;
  time_t start_time_;
  uint64_t maximum_cache_size_bytes_ = 0;

  SolverExitState exit_state_ = SolverExitState::NO_STATE;
  // different variable counts
  // number of variables  and clauses before preprocessing
```

(continues on next page)

(continued from previous page)

```
  VariableIndex num_original_variables_ = 0;
  ClauseIndex num_original_clauses_ = 0;
  ClauseIndex num_original_binary_clauses_ = 0;
  ClauseIndex num_original_unit_clauses_ = 0;

  VariableIndex num_variables_ = 0;
  VariableIndex num_used_variables_ = 0;
  VariableIndex num_free_variables_ = 0;
  VariableIndex num_indep_support_variables_ = 0;
  ClauseIndex num_long_clauses_ = 0;
  ClauseIndex num_binary_clauses_ = 0;

  ClauseIndex num_long_conflict_clauses_ = 0;
  ClauseIndex num_binary_conflict_clauses_ = 0;

  ClauseIndex times_conflict_clauses_cleaned_ = 0;

  ClauseIndex num_unit_clauses_ = 0;
  uint64_t num_decisions_ = 0;
  uint64_t num_failed_literals_detected_ = 0;
  uint64_t num_failed_literal_tests_ = 0;
  uint64_t num_conflicts_ = 0;

  // number of clauses overall learned
  uint64_t num_clauses_learned_ = 0;
  VariableIndex max_component_split_depth_ = 0;
  VariableIndex max_branch_var_depth_ = 0;
  TreeNodeIndex num_top_tree_nodes_ = 0;
  TreeNodeIndex num_tree_nodes_by_type_[static_cast<long>(TopTreeNodeType::NUM_TREE_
↪NODE_TYPES)];
  mpz_class num_models_by_tree_node_type_[static_cast<long>(TopTreeNodeType::NUM_TREE_
↪NODE_TYPES)];
  void UpdateNodeTypeStatistics(TopTreeNodeType node_type, const mpz_class & num_
↪models) {
    auto node_type_id = static_cast<long>(node_type);
    num_tree_nodes_by_type_[node_type_id]++;
    num_models_by_tree_node_type_[node_type_id] += num_models;
  }
  TreeNodeIndex num_tree_nodes(TopTreeNodeType node_type) {
    auto node_type_id = static_cast<long>(node_type);
    return num_tree_nodes_by_type_[node_type_id];
  }
  mpz_class num_tree_node_models(TopTreeNodeType node_type) {
    auto node_type_id = static_cast<long>(node_type);
    return num_models_by_tree_node_type_[node_type_id];
  }
  double percent_tree_node_models(TopTreeNodeType node_type) {
    if (final_solution_count_ == 0)
      return -1;
    mpf_class percent_models = num_tree_node_models(node_type);
    percent_models /= static_cast<mpf_class>(final_solution_count_);
    return percent_models.get_d();
  }

  /* cache statistics */
  uint64_t num_cache_hits_ = 0;
  uint64_t num_cache_look_ups_ = 0;
```

(continues on next page)

(continued from previous page)

```
  uint64_t sum_cache_hit_sizes_ = 0;

  uint64_t num_cached_components_ = 0;
  uint64_t sum_size_cached_components_ = 0;

  uint64_t sum_bytes_cached_components_ = 0;
  // the same number, summing over all components ever stored
  uint64_t overall_bytes_components_stored_ = 0;

  // the above numbers, but without any overhead,
  // counting only the pure data size of the components - without model counts
  uint64_t sum_bytes_pure_cached_component_data_ = 0;
  // the same number, summing over all components ever stored
  uint64_t overall_bytes_pure_stored_component_data_ = 0;


  uint64_t sys_overhead_sum_bytes_cached_components_ = 0;
    // the same number, summing over all components ever stored
  uint64_t sys_overhead_overall_bytes_components_stored_ = 0;

  uint64_t cache_infrastructure_bytes_memory_usage_ = 0;

  uint64_t overall_num_cache_stores_ = 0;

  /*end statistics */

  void reset_statistics() {
    // Reset cache statistics
    num_cache_hits_ = 0;
    num_cache_look_ups_ = 0;

    // Reset literal search operations
    num_failed_literals_detected_ = 0;
    num_failed_literal_tests_ = 0;
    num_conflicts_ = 0;
    num_decisions_ = 0;
//    num_implications_ = 0;
    num_unit_clauses_ = 0;

    // Reset execution parameters
    sampler_time_elapsed_ = 0.0;
    exit_state_ = SolverExitState::NO_STATE;

    // Reset all sampler only variables
    numb_second_pass_vars_.clear();

    sampler_time_elapsed_ = DBL_MAX;
    sampler_pass_1_time_ = DBL_MAX;
    sampler_pass_2_time_ = DBL_MAX;
  }

  bool cache_full() const {
    return cache_bytes_memory_usage() >= maximum_cache_size_bytes_;
  }

  uint64_t cache_bytes_memory_usage() const {
    return cache_infrastructure_bytes_memory_usage_
```

(continues on next page)

(continued from previous page)

```
          + sum_bytes_cached_components_;
}

uint64_t overall_cache_bytes_memory_stored() {
    return cache_infrastructure_bytes_memory_usage_
           + overall_bytes_components_stored_;
  }

void incorporate_cache_store(CacheableComponent &ccomp) {
  sum_bytes_cached_components_ += ccomp.SizeInBytes();
  sum_size_cached_components_ += ccomp.num_variables();
  num_cached_components_++;
  overall_bytes_components_stored_ += ccomp.SizeInBytes();
  overall_num_cache_stores_ += ccomp.num_variables();
  sys_overhead_sum_bytes_cached_components_ += ccomp.sys_overhead_SizeInBytes();
  sys_overhead_overall_bytes_components_stored_ += ccomp.sys_overhead_SizeInBytes();


  sum_bytes_pure_cached_component_data_ += ccomp.data_only_byte_size();
  overall_bytes_pure_stored_component_data_ += ccomp.data_only_byte_size();
}
void incorporate_cache_erase(CacheableComponent &ccomp) {
  sum_bytes_cached_components_ -= ccomp.SizeInBytes();
  sum_size_cached_components_ -= ccomp.num_variables();
  num_cached_components_--;
  sum_bytes_pure_cached_component_data_ -= ccomp.data_only_byte_size();

  sys_overhead_sum_bytes_cached_components_ -= ccomp.sys_overhead_SizeInBytes();
}

void incorporate_cache_hit(CacheableComponent &ccomp) {
  num_cache_hits_++;
  sum_cache_hit_sizes_ += ccomp.num_variables();
}
unsigned long cache_MB_memory_usage() {
  return cache_bytes_memory_usage() / 1000000;
}
mpz_class final_solution_count_ = 0;

double implicitBCP_miss_rate() {
  if (num_failed_literal_tests_ == 0) return 0.0;
  return (num_failed_literal_tests_ - num_failed_literals_detected_)
         / static_cast<double>(num_failed_literal_tests_);
}
unsigned long num_clauses() const {
  return num_long_clauses_ + num_binary_clauses_ + num_unit_clauses_;
}
unsigned long num_conflict_clauses() {
  return num_long_conflict_clauses_ + num_binary_conflict_clauses_;
}

unsigned long clause_deletion_interval() {
  return 10000 + 10 * times_conflict_clauses_cleaned_;
}
void set_final_solution_count(const mpz_class &count) {
  // set final_solution_count_ = count * 2^(num_variables_ - num_used_variables_)
  mpz_mul_2exp(final_solution_count_.get_mpz_t(), count.get_mpz_t(),
```

(continues on next page)

(continued from previous page)

```
                num_variables_ - num_used_variables_);
  }
  const mpz_class &final_solution_count() const {
    return final_solution_count_;
  }

//  void incorporateConflictClauseData(const std::vector<LiteralID> &clause) {
//    if (clause.size() == 1)
//      num_unit_clauses_++;
//    else if (clause.size() == 2)
//      num_binary_conflict_clauses_++;
//    num_long_conflict_clauses_++;
//  }
  void incorporateClauseData(const std::vector<LiteralID> &clause) {
    if (clause.size() == 1)
      num_unit_clauses_++;
    else if (clause.size() == 2)
      num_binary_clauses_++;
    else
      num_long_clauses_++;
  }

  void printShort();
  void printShortFormulaInfo() {
    std::cout << "variables (all/used/free): \t"
              << num_variables_ << "/" << num_used_variables_ << "/"
              << num_variables_ - num_used_variables_ << "\n"
              << "independent support size:  \t" << num_indep_support_variables_ <<
↪"\n";

    std::cout << "clauses (all/long/binary/unit): "
              << num_clauses() << "/" << num_long_clauses_
              << "/" << num_binary_clauses_ << "/" << num_unit_clauses_ << std::endl;
  }
  unsigned long getTime() const {
    return num_decisions_;
  }

  long double getAvgComponentSize() {
    return sum_size_cached_components_ / static_cast<long double>(num_cached_
↪components_);
  }
  unsigned long cached_component_count() {
    return num_cached_components_;
  }
  unsigned long cache_hits() {
    return num_cache_hits_;
  }
  double cache_miss_rate() {
    if (num_cache_look_ups_ == 0) return 0.0;
    return (num_cache_look_ups_ - num_cache_hits_) / static_cast<double>(num_cache_
↪look_ups_);
  }
  long double getAvgCacheHitSize() {
    if (num_cache_hits_ == 0) return 0.0;
    return sum_cache_hit_sizes_ / static_cast<long double>(num_cache_hits_);
  }
```

(continues on next page)

(continued from previous page)

```
  //--------------------------------------//
  //        Sampler Fields and Methods        //
  //--------------------------------------//
  double sampler_time_elapsed_ = DBL_MAX;
  double sampler_pass_1_time_ = DBL_MAX;
  double sampler_pass_2_time_ = DBL_MAX;
  std::vector<VariableIndex> numb_second_pass_vars_;
};

#endif /* STATISTICS_H_ */
```

Includes

- cfloat
- component_types/cacheable_component.h (*File cacheable_component.h*)
- cstdint
- gmpxx.h
- primitive_types.h (*File primitive_types.h*)
- solver_config.h (*File solver_config.h*)
- string
- structures.h (*File structures.h*)
- vector

Included By

- *File model_sampler.h*
- *File sampler_tools.h*
- *File alt_component_analyzer.h*
- *File component_cache.h*
- *File instance.h*
- *File solver.h*
- *File statistics.cpp*

Classes

- *Class DataAndStatistics*

File structures.h

Parent directory (src)

Contents

- *Definition (src/structures.h)*
- *Includes*
- *Included By*
- *Classes*
- *Functions*
- *Defines*
- *Typedefs*

Definition (src/structures.h)

Program Listing for File structures.h

Return to documentation for file (src/structures.h)

```
#ifndef STRUCTURES_H_
#define STRUCTURES_H_

#include <assert.h>
#include <vector>
#include <iostream>

#include "primitive_types.h"

#define INVALID_DL -1

typedef unsigned char TriValue;
#define   F_TRI  0
#define   T_TRI  1
#define   X_TRI  2

class LiteralID {
 public:
  LiteralID() {
    value_ = 0;
  }
  // This code does screwy casting.  Cannot use explicit
  LiteralID(int lit) {  // NOLINT (runtime/explicit)
    value_ = (abs(lit) << 1) + static_cast<unsigned>(lit > 0);
  }
  LiteralID(VariableIndex var, bool sign) {
    value_ = (var << 1) + static_cast<unsigned>(sign);
  }

  VariableIndex var() const {
    return (value_ >> 1);
  }
  int toInt() const {
    return (static_cast<int>(value_) >> 1) * ((sign()) ? 1 : -1);
```

(continues on next page)

(continued from previous page)

```
  }
  void inc() {++value_;}
  void copyRaw(unsigned int v) {
    value_ = v;
  }
  const bool sign() const {
    return static_cast<bool>(value_ & 0x01);
  }
  const bool operator!=(const LiteralID &rL2) const {
    return value_ != rL2.value_;
  }
  const bool operator==(const LiteralID &rL2) const {
    return value_ == rL2.value_;
  }
  const bool operator<(const LiteralID &rL2) const {
    return value_ < rL2.value_;
  }
  const LiteralID neg() const {
    return LiteralID(var(), !sign());
  }

  void print() const {
    std::cout << (sign() ? " " : "-") << var() << " ";
  }
  inline unsigned raw() const { return value_;}

 private:
  unsigned value_;

  template <class _T> friend class LiteralIndexedVector;
};

static const LiteralID NOT_A_LIT(0, false);
#define SENTINEL_LIT NOT_A_LIT

class Literal {
 public:
  std::vector<LiteralID> binary_links_ = std::vector<LiteralID>(1, SENTINEL_LIT);
  std::vector<ClauseOfs> watch_list_ = std::vector<ClauseOfs>(1, SENTINEL_CL);  //␣
↪Watch clauses
  float activity_score_ = 0.0f;

  void increaseActivity(unsigned u = 1) {
    activity_score_+= u;
  }
  void resetActivity() { activity_score_ = 0;}
  void removeWatchLinkTo(ClauseOfs clause_ofs) {
    for (auto it = watch_list_.begin(); it != watch_list_.end(); it++)
      if (*it == clause_ofs) {
        *it = watch_list_.back();  // Get last element in vector and replace item to␣
↪be removed
        watch_list_.pop_back();  // Delete the last element
        return;
      }
  }

  void replaceWatchLinkTo(ClauseOfs clause_ofs, ClauseOfs replace_ofs) {
```

(continues on next page)

(continued from previous page)

```
    for (auto it = watch_list_.begin(); it != watch_list_.end(); it++)
      if (*it == clause_ofs) {
        *it = replace_ofs;
        return;
      }
  }
  void addWatchLinkTo(ClauseIndex clause_ofs) {
    watch_list_.push_back(clause_ofs);
  }
  void addBinLinkTo(LiteralID lit) {
    binary_links_.back() = lit;
    binary_links_.push_back(SENTINEL_LIT);
  }
  void resetWatchList() {
    watch_list_.clear();
    watch_list_.push_back(SENTINEL_CL);
  }
  bool hasBinaryLinkTo(LiteralID lit) {
    for (auto l : binary_links_) {
      if (l == lit)
        return true;
    }
    return false;
  }
  bool hasBinaryLinks() {
    return !binary_links_.empty();
  }
};

class Antecedent {
  unsigned int val_;

 public:
  Antecedent() {
    val_ = 1;
  }
  explicit Antecedent(const ClauseOfs cl_ofs) {
    val_ = (cl_ofs << 1) | 1;
  }
  explicit Antecedent(const LiteralID idLit) {
    val_ = (idLit.raw() << 1);
  }
  bool isAClause() const {
    return val_ & 0x01;
  }
  ClauseOfs asCl() const {
    assert(isAClause());  // ZH - Verify the antecedent actually is a CLAUSE.
    return val_ >> 1;
  }
  LiteralID asLit() {
    assert(!isAClause());  // ZH - Verify the antecedent actually is a LITERAL.
    LiteralID idLit;
    idLit.copyRaw(val_ >> 1);
    return idLit;
  }
  bool isAnt() const {
    return val_ != 1;  // i.e. NOT a NOT_A_CLAUSE;
```

(continues on next page)

(continued from previous page)

```
  }
};

struct Variable {
  Antecedent ante;
  long decision_level = INVALID_DL;
};


class ClauseHeader {
  unsigned creation_time_;  // number of conflicts seen at creation time
  unsigned score_;
  unsigned length_;

 public:
  void increaseScore() {
    score_++;
  }
  void decayScore() {
    score_ >>= 1;
  }
  unsigned score() const {
    return score_;
  }

  unsigned creation_time() {
    return creation_time_;
  }
  unsigned length() { return length_; }
  void set_length(unsigned length) { length_ = length; }

  void set_creation_time(unsigned time) {
    creation_time_ = time;
  }
  static unsigned overheadInLits() { return sizeof(ClauseHeader)/sizeof(LiteralID); }
};

#endif /* STRUCTURES_H_ */
```

Includes

- assert.h
- iostream
- primitive_types.h (*File primitive_types.h*)
- vector

Included By

- *File model_sampler.h*
- *File statistics.h*
- *File containers.h*

- *File cached_assignment.h*
- *File instance.h*

Classes

- *Struct Variable*
- *Class Antecedent*
- *Class ClauseHeader*
- *Class Literal*
- *Class LiteralID*

Functions

- *Function NOT_A_LIT*

Defines

- *Define F_TRI*
- *Define INVALID_DL*
- *Define SENTINEL_LIT*
- *Define T_TRI*
- *Define X_TRI*

Typedefs

- *Typedef TriValue*

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

Index

A

AltComponentAnalyzer (C++ class), 5
AltComponentAnalyzer::AltComponentAnalyzer (C++ function), 6
AltComponentAnalyzer::current_archetype (C++ function), 6
AltComponentAnalyzer::exploreRemainingCompOf (C++ function), 6
AltComponentAnalyzer::getArchetype (C++ function), 7
AltComponentAnalyzer::initialize (C++ function), 6
AltComponentAnalyzer::isUnseenAndActive (C++ function), 6
AltComponentAnalyzer::makeComponentFromArcheType (C++ function), 6
AltComponentAnalyzer::manageSearchOccurrenceAndScoreOf (C++ function), 6
AltComponentAnalyzer::manageSearchOccurrenceOf (C++ function), 6
AltComponentAnalyzer::max_clause_id (C++ function), 6
AltComponentAnalyzer::max_variable_id (C++ function), 6
AltComponentAnalyzer::scoreOf (C++ function), 6
AltComponentAnalyzer::setSeenAndStoreInSearchStack (C++ function), 6
AltComponentAnalyzer::setupAnalysisContext (C++ function), 6
Antecedent (C++ class), 7
Antecedent::Antecedent (C++ function), 7
Antecedent::asCl (C++ function), 7
Antecedent::asLit (C++ function), 7
Antecedent::isAClause (C++ function), 7
Antecedent::isAnt (C++ function), 7
AssignmentContainer (C++ type), 70
AssignmentEncoding (C++ type), 55
ASSN_F (C++ enumerator), 55
ASSN_T (C++ enumerator), 55
ASSN_U (C++ enumerator), 55

B

BACKTRACK (C++ enumerator), 57
BasePackedComponent (C++ class), 8
BasePackedComponent::_bits_of_data_size (C++ member), 10
BasePackedComponent::_bits_per_block (C++ member), 10
BasePackedComponent::_bits_per_clause (C++ member), 10
BasePackedComponent::_bits_per_variable (C++ member), 10
BasePackedComponent::_clause_mask (C++ member), 10
BasePackedComponent::_data_size_mask (C++ member), 10
BasePackedComponent::_debug_static_val (C++ member), 9
BasePackedComponent::_variable_mask (C++ member), 10
BasePackedComponent::~BasePackedComponent (C++ function), 8
BasePackedComponent::adjustPackSize (C++ function), 9
BasePackedComponent::alloc_of_model_count (C++ function), 8
BasePackedComponent::BasePackedComponent (C++ function), 8
BasePackedComponent::bits_of_data_size (C++ function), 9
BasePackedComponent::bits_per_block (C++ function), 9
BasePackedComponent::bits_per_clause (C++ function), 9
BasePackedComponent::bits_per_variable (C++ function), 9
BasePackedComponent::clear (C++ function), 8
BasePackedComponent::creation_time (C++ function), 8
BasePackedComponent::creation_time_ (C++ member), 10

BasePackedComponent::data_ (C++ member), 9
BasePackedComponent::hashkey (C++ function), 8
BasePackedComponent::hashkey_ (C++ member), 9
BasePackedComponent::isDeletable (C++ function), 8
BasePackedComponent::length_solution_period_and_flags_ (C++ member), 10
BasePackedComponent::log2 (C++ function), 9
BasePackedComponent::model_count (C++ function), 8
BasePackedComponent::model_count_ (C++ member), 10
BasePackedComponent::modelCountFound (C++ function), 8
BasePackedComponent::outbit (C++ function), 9
BasePackedComponent::set_creation_time (C++ function), 8
BasePackedComponent::set_deletable (C++ function), 8
BasePackedComponent::set_model_count (C++ function), 8
BasePackedComponent::variable_mask (C++ function), 9
BITS_PER_BYTE (C macro), 62
BitStuffer (C++ class), 11
BitStuffer::assert_size (C++ function), 11
BitStuffer::BitStuffer (C++ function), 11
BitStuffer::stuff (C++ function), 11

C

CA_CL_ALL_LITS_ACTIVE (C macro), 63
CA_CL_IN_OTHER_COMP (C macro), 63
CA_CL_IN_SUP_COMP_UNSEEN (C macro), 63
CA_CL_MASK (C macro), 63
CA_CL_SEEN (C macro), 63
CA_NIL (C macro), 64
CA_SearchState (C++ type), 70
CA_VAR_IN_OTHER_COMP (C macro), 64
CA_VAR_IN_SUP_COMP_UNSEEN (C macro), 64
CA_VAR_MASK (C macro), 64
CA_VAR_SEEN (C macro), 64
CacheableComponent (C++ type), 70
CACHED_VARIABLE_LEN (C macro), 65
CachedAssignment (C++ class), 12
CachedAssignment::CachedAssignment (C++ function), 12
CachedAssignment::clear (C++ function), 13
CachedAssignment::emancipated_vars (C++ function), 12
CachedAssignment::empty (C++ function), 12
CachedAssignment::IncreaseSize (C++ function), 12
CachedAssignment::literals (C++ function), 12
CachedAssignment::num_components (C++ function), 12
CachedAssignment::ProcessComponent (C++ function), 12
CachedAssignment::Sort (C++ function), 12
CachedAssignment::vars (C++ function), 12
CacheEntryID (C++ type), 71
CLAUSE_ADDING_ERROR (C macro), 65
ClauseHeader (C++ class), 13
ClauseHeader::creation_time (C++ function), 13
ClauseHeader::decayScore (C++ function), 13
ClauseHeader::increaseScore (C++ function), 13
ClauseHeader::length (C++ function), 13
ClauseHeader::overheadInLits (C++ function), 13
ClauseHeader::score (C++ function), 13
ClauseHeader::set_creation_time (C++ function), 13
ClauseHeader::set_length (C++ function), 13
ClauseIndex (C++ type), 71
ClauseOfs (C++ type), 71
clsSENTINEL (C macro), 65
COLOR_BLACK (C++ enumerator), 56
COLOR_BLUE (C++ enumerator), 56
COLOR_CYAN (C++ enumerator), 56
COLOR_GREEN (C++ enumerator), 56
COLOR_MAGENTA (C++ enumerator), 56
COLOR_RED (C++ enumerator), 56
COLOR_YELLOW (C++ enumerator), 56
Component (C++ class), 13
Component::addCl (C++ function), 14
Component::addVar (C++ function), 14
Component::clauses_ofs (C++ function), 14
Component::clear (C++ function), 14
Component::closeClauseData (C++ function), 14
Component::closeVariableData (C++ function), 14
Component::clsBegin (C++ function), 14
Component::createAsDummyComponent (C++ function), 14
Component::empty (C++ function), 14
Component::getData (C++ function), 15
Component::id (C++ function), 14
Component::num_variables (C++ function), 14
Component::numLongClauses (C++ function), 14
Component::reserveSpace (C++ function), 14
Component::set_id (C++ function), 14
Component::varsBegin (C++ function), 14
COMPONENT_SPLIT (C++ enumerator), 57
ComponentAnalyzer (C++ type), 71
ComponentArchetype (C++ class), 15
ComponentArchetype::clause_all_lits_active (C++ function), 15
ComponentArchetype::clause_nil (C++ function), 16
ComponentArchetype::clause_seen (C++ function), 15
ComponentArchetype::clause_seen_in_peer_comp (C++ function), 16
ComponentArchetype::clause_unseen_in_sup_comp (C++ function), 16
ComponentArchetype::clearArrays (C++ function), 16
ComponentArchetype::ComponentArchetype (C++ function), 15

ComponentArchetype::current_comp_for_caching_ (C++ member), 16
ComponentArchetype::initArrays (C++ function), 16
ComponentArchetype::makeComponentFromState (C++ function), 16
ComponentArchetype::reInitialize (C++ function), 15
ComponentArchetype::setClause_all_lits_active (C++ function), 15
ComponentArchetype::setClause_in_other_comp (C++ function), 15
ComponentArchetype::setClause_in_sup_comp_unseen (C++ function), 15
ComponentArchetype::setClause_nil (C++ function), 15
ComponentArchetype::setClause_seen (C++ function), 15
ComponentArchetype::setVar_in_other_comp (C++ function), 15
ComponentArchetype::setVar_in_sup_comp_unseen (C++ function), 15
ComponentArchetype::setVar_nil (C++ function), 15
ComponentArchetype::setVar_seen (C++ function), 15
ComponentArchetype::stack_level (C++ function), 15
ComponentArchetype::super_comp (C++ function), 15
ComponentArchetype::var_nil (C++ function), 15
ComponentArchetype::var_seen (C++ function), 15
ComponentArchetype::var_seen_in_peer_comp (C++ function), 16
ComponentArchetype::var_unseen_in_sup_comp (C++ function), 16
ComponentCache (C++ class), 16
ComponentCache::~ComponentCache (C++ function), 17
ComponentCache::cleanPollutionsInvolving (C++ function), 18
ComponentCache::ComponentCache (C++ function), 16
ComponentCache::compute_byte_size_infrastructure (C++ function), 17
ComponentCache::debug_dump_data (C++ function), 18
ComponentCache::deleteEntries (C++ function), 18
ComponentCache::entry (C++ function), 17
ComponentCache::entry_const (C++ function), 17
ComponentCache::eraseEntry (C++ function), 18
ComponentCache::hasEntry (C++ function), 17
ComponentCache::init (C++ function), 17
ComponentCache::manageNewComponent (C++ function), 18
ComponentCache::removeFromDescendantsTree (C++ function), 18
ComponentCache::removeFromHashTable (C++ function), 18
ComponentCache::storeAsEntry (C++ function), 18
ComponentCache::storeValueOf (C++ function), 18
ComponentCache::test_descendantstree_consistency (C++ function), 18
ComponentManager (C++ class), 19
ComponentManager::buildFreedVariableList (C++ function), 21
ComponentManager::buildResidualComponent (C++ function), 21
ComponentManager::cacheModelCountAndAssignment (C++ function), 19
ComponentManager::cacheModelCountOf (C++ function), 19
ComponentManager::cleanRemainingComponentsOf (C++ function), 20
ComponentManager::component (C++ function), 20
ComponentManager::component_stack_size (C++ function), 20
ComponentManager::ComponentManager (C++ function), 19
ComponentManager::findNextRemainingComponentOf (C++ function), 20
ComponentManager::gatherStatistics (C++ function), 21
ComponentManager::initialize (C++ function), 19
ComponentManager::recordRemainingCompsFor (C++ function), 20
ComponentManager::removeAllCachePollutionsOf (C++ function), 21
ComponentManager::scoreOf (C++ function), 19
ComponentManager::sortComponentStackRange (C++ function), 21
ComponentManager::super_component (C++ function), 20
ComponentManager::superComponentOf (C++ function), 20
ComponentManager::top_stack (C++ function), 21
ComponentVarAndCls (C++ type), 72
CYLINDER (C++ enumerator), 57

D

DataAndStatistics (C++ class), 22
DataAndStatistics::cache_bytes_memory_usage (C++ function), 23
DataAndStatistics::cache_full (C++ function), 23
DataAndStatistics::cache_hits (C++ function), 24
DataAndStatistics::cache_infrastructure_bytes_memory_usage_ (C++ member), 26
DataAndStatistics::cache_MB_memory_usage (C++ function), 23
DataAndStatistics::cache_miss_rate (C++ function), 24
DataAndStatistics::cached_component_count (C++ function), 24
DataAndStatistics::clause_deletion_interval (C++ function), 23
DataAndStatistics::DataAndStatistics (C++ function), 22
DataAndStatistics::exit_state_ (C++ member), 25
DataAndStatistics::final_solution_count (C++ function), 23

DataAndStatistics::final_solution_count_ (C++ member), 26
DataAndStatistics::getAvgCacheHitSize (C++ function), 24
DataAndStatistics::getAvgComponentSize (C++ function), 24
DataAndStatistics::getTime (C++ function), 24
DataAndStatistics::implicitBCP_miss_rate (C++ function), 23
DataAndStatistics::incorporate_cache_erase (C++ function), 23
DataAndStatistics::incorporate_cache_hit (C++ function), 23
DataAndStatistics::incorporate_cache_store (C++ function), 23
DataAndStatistics::incorporateClauseData (C++ function), 23
DataAndStatistics::input_file_ (C++ member), 24
DataAndStatistics::max_branch_var_depth_ (C++ member), 25
DataAndStatistics::max_component_split_depth_ (C++ member), 25
DataAndStatistics::maximum_cache_size_bytes_ (C++ member), 24
DataAndStatistics::num_binary_clauses_ (C++ member), 25
DataAndStatistics::num_binary_conflict_clauses_ (C++ member), 25
DataAndStatistics::num_cache_hits_ (C++ member), 25
DataAndStatistics::num_cache_look_ups_ (C++ member), 26
DataAndStatistics::num_cached_components_ (C++ member), 26
DataAndStatistics::num_clauses (C++ function), 23
DataAndStatistics::num_clauses_learned_ (C++ member), 25
DataAndStatistics::num_conflict_clauses (C++ function), 23
DataAndStatistics::num_conflicts_ (C++ member), 25
DataAndStatistics::num_decisions_ (C++ member), 25
DataAndStatistics::num_failed_literal_tests_ (C++ member), 25
DataAndStatistics::num_failed_literals_detected_ (C++ member), 25
DataAndStatistics::num_free_variables_ (C++ member), 25
DataAndStatistics::num_indep_support_variables_ (C++ member), 25
DataAndStatistics::num_long_clauses_ (C++ member), 25
DataAndStatistics::num_long_conflict_clauses_ (C++ member), 25
DataAndStatistics::num_models_by_tree_node_type_ (C++ member), 25
DataAndStatistics::num_original_binary_clauses_ (C++ member), 25
DataAndStatistics::num_original_clauses_ (C++ member), 25
DataAndStatistics::num_original_unit_clauses_ (C++ member), 25
DataAndStatistics::num_original_variables_ (C++ member), 25
DataAndStatistics::num_top_tree_nodes_ (C++ member), 25
DataAndStatistics::num_tree_node_models (C++ function), 22
DataAndStatistics::num_tree_nodes (C++ function), 22
DataAndStatistics::num_tree_nodes_by_type_ (C++ member), 25
DataAndStatistics::num_unit_clauses_ (C++ member), 25
DataAndStatistics::num_used_variables_ (C++ member), 25
DataAndStatistics::num_variables_ (C++ member), 25
DataAndStatistics::numb_second_pass_vars_ (C++ member), 26
DataAndStatistics::overall_bytes_components_stored_ (C++ member), 26
DataAndStatistics::overall_bytes_pure_stored_component_data_ (C++ member), 26
DataAndStatistics::overall_cache_bytes_memory_stored (C++ function), 23
DataAndStatistics::overall_num_cache_stores_ (C++ member), 26
DataAndStatistics::percent_tree_node_models (C++ function), 23
DataAndStatistics::printShort (C++ function), 23
DataAndStatistics::printShortFormulaInfo (C++ function), 24
DataAndStatistics::reset_statistics (C++ function), 23
DataAndStatistics::sampler_pass_1_time_ (C++ member), 26
DataAndStatistics::sampler_pass_2_time_ (C++ member), 26
DataAndStatistics::sampler_time_elapsed_ (C++ member), 26
DataAndStatistics::set_final_solution_count (C++ function), 23
DataAndStatistics::start_time_ (C++ member), 24
DataAndStatistics::sum_bytes_cached_components_ (C++ member), 26
DataAndStatistics::sum_bytes_pure_cached_component_data_ (C++ member), 26
DataAndStatistics::sum_cache_hit_sizes_ (C++ member), 26
DataAndStatistics::sum_size_cached_components_ (C++ member), 26
DataAndStatistics::sys_overhead_overall_bytes_components_stored_

(C++ member), 26
DataAndStatistics::sys_overhead_sum_bytes_cached_components_ (C++ member), 26
DataAndStatistics::times_conflict_clauses_cleaned_ (C++ member), 25
DataAndStatistics::UpdateNodeTypeStatistics (C++ function), 22
DecisionLevel (C++ type), 72
DecisionStack (C++ class), 26
DecisionStack::DecisionStack (C++ function), 27
DecisionStack::get_decision_level (C++ function), 27
DecisionStack::on_deck (C++ function), 27
DecisionStack::startFailedLitTest (C++ function), 27
DecisionStack::stopFailedLitTest (C++ function), 27
DecisionStack::top (C++ function), 27
DecisionStack::top_const (C++ function), 27
DifferencePackedComponent (C++ class), 27
DifferencePackedComponent::data_only_byte_size (C++ function), 28
DifferencePackedComponent::data_size (C++ function), 28
DifferencePackedComponent::DifferencePackedComponent (C++ function), 28
DifferencePackedComponent::equals (C++ function), 28
DifferencePackedComponent::num_variables (C++ function), 28
DifferencePackedComponent::raw_data_byte_size (C++ function), 28
DifferencePackedComponent::sys_overhead_raw_data_byte_size (C++ function), 28

E

END_ESCAPE (C macro), 65
ESCAPE_CHAR (C macro), 66
EXIT (C++ enumerator), 56
EXIT_TIMEOUT (C macro), 66
ExitInvalidParam (C++ function), 57
ExitWithError (C++ function), 57

F

F_TRI (C macro), 66
FILE_PATH_SEPARATOR (C macro), 66
FileExists (C++ function), 58
FIRST_VAR (C macro), 67

G

GenericCacheableComponent (C++ class), 28
GenericCacheableComponent::father (C++ function), 29
GenericCacheableComponent::first_descendant (C++ function), 29
GenericCacheableComponent::GenericCacheableComponent (C++ function), 28
GenericCacheableComponent::next_bucket_element (C++ function), 29
GenericCacheableComponent::next_sibling (C++ function), 29
GenericCacheableComponent::set_father (C++ function), 28
GenericCacheableComponent::set_first_descendant (C++ function), 29
GenericCacheableComponent::set_next_bucket_element (C++ function), 29
GenericCacheableComponent::set_next_sibling (C++ function), 29
GenericCacheableComponent::SizeInBytes (C++ function), 28
GenericCacheableComponent::sys_overhead_SizeInBytes (C++ function), 28

I

Instance (C++ class), 29
Instance::addBinaryClause (C++ function), 31
Instance::addClause (C++ function), 31
Instance::addUIPConflictClause (C++ function), 31
Instance::addUnitClause (C++ function), 31
Instance::antecedent (C++ function), 29
Instance::beginOf (C++ function), 32
Instance::cleanClause (C++ function), 30
Instance::compactClauses (C++ function), 30
Instance::compactConflictLiteralPool (C++ function), 30
Instance::compactVariables (C++ function), 30
Instance::conflict_clauses_ (C++ member), 32
Instance::conflict_clauses_begin (C++ function), 32
Instance::createfromFile (C++ function), 30
Instance::decayActivities (C++ function), 30
Instance::deleteConflictClauses (C++ function), 30
Instance::existsUnitClauseOf (C++ function), 31
Instance::free (C++ function), 30
Instance::getAntecedent (C++ function), 29
Instance::getHeaderOf (C++ function), 32
Instance::has_independent_support_ (C++ member), 33
Instance::hasAntecedent (C++ function), 29
Instance::independent_support (C++ member), 33
Instance::isActive (C++ function), 32
Instance::isAntecedentOf (C++ function), 29
Instance::isolated (C++ function), 30
Instance::isResolved (C++ function), 32
Instance::isSatisfied (C++ function), 32
Instance::isUnitClause (C++ function), 30
Instance::literal (C++ function), 32
Instance::literal_pool_ (C++ member), 32
Instance::literal_values_ (C++ member), 32
Instance::literals_ (C++ member), 32
Instance::markClauseDeleted (C++ function), 30
Instance::num_conflict_clauses (C++ function), 30
Instance::num_variables (C++ function), 30
Instance::occurrence_lists_ (C++ member), 32
Instance::original_lit_pool_size_ (C++ member), 32

Instance::ParseCnfCommentForSupport (C++ function), 31
Instance::statistics_ (C++ member), 32
Instance::unit_clauses_ (C++ member), 32
Instance::unSet (C++ function), 29
Instance::unused_vars_ (C++ member), 33
Instance::updateActivities (C++ function), 30
Instance::var (C++ function), 31
Instance::var_const (C++ function), 31
Instance::variables_ (C++ member), 32
INVALID_DL (C macro), 67
ITEM_SEP (C macro), 67

L

ListOfSamples (C++ type), 72
Literal (C++ class), 33
Literal::activity_score_ (C++ member), 34
Literal::addBinLinkTo (C++ function), 33
Literal::addWatchLinkTo (C++ function), 33
Literal::binary_links_ (C++ member), 34
Literal::hasBinaryLinks (C++ function), 34
Literal::hasBinaryLinkTo (C++ function), 34
Literal::increaseActivity (C++ function), 33
Literal::removeWatchLinkTo (C++ function), 33
Literal::replaceWatchLinkTo (C++ function), 33
Literal::resetActivity (C++ function), 33
Literal::resetWatchList (C++ function), 34
Literal::watch_list_ (C++ member), 34
LiteralID (C++ class), 34
LiteralID::copyRaw (C++ function), 35
LiteralID::inc (C++ function), 35
LiteralID::LiteralID (C++ function), 34
LiteralID::neg (C++ function), 35
LiteralID::operator!= (C++ function), 35
LiteralID::operator== (C++ function), 35
LiteralID::operator< (C++ function), 35
LiteralID::print (C++ function), 35
LiteralID::raw (C++ function), 35
LiteralID::sign (C++ function), 35
LiteralID::toInt (C++ function), 35
LiteralID::var (C++ function), 35
LiteralIndexedVector (C++ class), 36
LiteralIndexedVector::begin (C++ function), 36
LiteralIndexedVector::end_lit (C++ function), 37
LiteralIndexedVector::LiteralIndexedVector (C++ function), 36
LiteralIndexedVector::operator[] (C++ function), 36
LiteralIndexedVector::reserve (C++ function), 37
LiteralIndexedVector::resize (C++ function), 37
LT (C macro), 67

M

MAX_DEPTH (C++ enumerator), 57

N

NIL_ENTRY (C macro), 67
NO_STATE (C++ enumerator), 56
NUM_TREE_NODE_TYPES (C++ enumerator), 57

P

PartialAssignment (C++ type), 72
PRINT_BOLD (C macro), 68
PrintColor (C++ type), 56
PrintWarning (C++ function), 61
PROCESS_COMPONENT (C++ enumerator), 56

R

Random (C++ class), 37
Random::binom (C++ function), 38
Random::DownsampleList (C++ function), 38
Random::init (C++ function), 38
Random::Mpz (C++ class), 39, 40
Random::Mpz::binom (C++ function), 39, 40
Random::Mpz::uniform (C++ function), 39, 40
Random::SelectRangeInts (C++ function), 39
Random::shuffle (C++ function), 39
Random::uniform (C++ function), 38
RESET_CONSOLE_COLOR (C macro), 68
RESOLVED (C++ enumerator), 56

S

SampleAssignment (C++ class), 41
SampleAssignment::cache_comp_ids (C++ function), 42
SampleAssignment::clear_cache_comp_ids (C++ function), 43
SampleAssignment::GetPartialAssignment (C++ function), 41
SampleAssignment::IsComplete (C++ function), 41
SampleAssignment::IsVarEmancipated (C++ function), 42
SampleAssignment::num_set_vars (C++ function), 42
SampleAssignment::num_set_vars_const (C++ function), 42
SampleAssignment::num_unset_vars (C++ function), 42
SampleAssignment::sample_count (C++ function), 42
SampleAssignment::SampleAssignment (C++ function), 41
SampleAssignment::set_num_var (C++ function), 43
SampleAssignment::setVarAssignment (C++ function), 41
SampleAssignment::ToString (C++ function), 41
SampleAssignment::var_assignment (C++ function), 41
SampleSize (C++ type), 72
SamplesManager (C++ class), 43
SamplesManager::AddCachedCompIds (C++ function), 46
SamplesManager::AddEmancipatedVars (C++ function), 46

SamplesManager::append (C++ function), 47
SamplesManager::clear (C++ function), 48
SamplesManager::exportFinal (C++ function), 43
SamplesManager::GenerateSamplesToReplace (C++ function), 44
SamplesManager::GetActualSampleCount (C++ function), 48
SamplesManager::IsComplete (C++ function), 46
SamplesManager::KeepSamples (C++ function), 47
SamplesManager::merge (C++ function), 45
SamplesManager::model_count (C++ function), 46
SamplesManager::num_samples (C++ function), 47
SamplesManager::RemoveSamples (C++ function), 47
SamplesManager::reservoirSample (C++ function), 43
SamplesManager::samples (C++ function), 46
SamplesManager::SamplesManager (C++ function), 43
SamplesManager::SplitAndStitch (C++ function), 44
SamplesManager::stitch (C++ function), 44
SamplesManager::StitchShuffledArray (C++ function), 44
SamplesManager::TransferVariableAssignments (C++ function), 46
SamplesManager::verifyPostStitchingCorrectness (C++ function), 45
SamplesManager::verifySampleCount (C++ function), 47
SamplesManager::VerifySolutions (C++ function), 45
SENTINEL_CL (C macro), 68
SENTINEL_LIT (C macro), 68
Solver (C++ class), 48
Solver::config (C++ function), 49
Solver::DisableTopTreeSampling (C++ function), 49
Solver::PushNewSamplesManagerOnStack (C++ function), 49
Solver::sample_models (C++ function), 49
Solver::solve (C++ function), 48
Solver::Solver (C++ function), 48
Solver::statistics (C++ function), 49
SolverConfiguration (C++ class), 3
SolverConfiguration::debug_mode (C++ member), 4
SolverConfiguration::disable_samples_write_ (C++ member), 4
SolverConfiguration::EnableSampleCaching (C++ function), 4
SolverConfiguration::max_top_tree_depth_ (C++ member), 5
SolverConfiguration::max_top_tree_leaf_sample_count (C++ member), 5
SolverConfiguration::num_samples_ (C++ member), 5
SolverConfiguration::num_samples_to_cache_ (C++ member), 5
SolverConfiguration::perform_component_caching (C++ member), 4
SolverConfiguration::perform_failed_lit_test (C++ member), 4
SolverConfiguration::perform_pre_processing (C++ member), 4
SolverConfiguration::perform_random_sampling_ (C++ member), 4
SolverConfiguration::perform_sample_caching (C++ function), 4
SolverConfiguration::perform_top_tree_sampling (C++ member), 4
SolverConfiguration::perform_two_pass_sampling_ (C++ member), 5
SolverConfiguration::quiet (C++ member), 5
SolverConfiguration::samples_output_file (C++ member), 4
SolverConfiguration::skip_partial_assignment_fill (C++ member), 5
SolverConfiguration::store_sampled_models (C++ function), 4
SolverConfiguration::time_bound_seconds (C++ member), 4
SolverConfiguration::top_tree_samples_output_file_ (C++ member), 5
SolverConfiguration::verbose (C++ member), 4
SolverExitState (C++ type), 56
SolverNextAction (C++ type), 56
StackLevel (C++ class), 49
StackLevel::addCachedCompIds (C++ function), 52
StackLevel::addFreeVariables (C++ function), 51
StackLevel::anotherCompProcessible (C++ function), 51
StackLevel::branch_found_unsat (C++ function), 52
StackLevel::cached_assn (C++ function), 53
StackLevel::cached_comp_ids (C++ function), 52
StackLevel::changeBranch (C++ function), 50
StackLevel::ClearCachedAssn (C++ function), 53
StackLevel::componentSplitDepth (C++ function), 54
StackLevel::configureNewLevel (C++ function), 51
StackLevel::currentRemainingComponent (C++ function), 50
StackLevel::emancipated_vars (C++ function), 52
StackLevel::getActiveModelCount (C++ function), 52
StackLevel::getSamplerSolutionMultiplier (C++ function), 52
StackLevel::getTotalModelCount (C++ function), 52
StackLevel::HasNoDescendentModels (C++ function), 52
StackLevel::hasUnprocessedComponents (C++ function), 50
StackLevel::includeSolution (C++ function), 51
StackLevel::includeSolutionSampleMultiplier (C++ function), 51
StackLevel::isComponentSplit (C++ function), 53
StackLevel::isFirstComponent (C++ function), 53
StackLevel::isSecondBranch (C++ function), 50
StackLevel::literal_stack_ofs (C++ function), 51
StackLevel::mark_branch_unsat (C++ function), 52

StackLevel::markFirstComponentComplete (C++ function), 53
StackLevel::nextUnprocessedComponent (C++ function), 50
StackLevel::random_cache_sample (C++ function), 53
StackLevel::remaining_components_ofs (C++ function), 50
StackLevel::resetRemainingComps (C++ function), 50
StackLevel::set_cache_sample (C++ function), 53
StackLevel::set_cached_assn (C++ function), 53
StackLevel::set_solver_config_and_statistics (C++ function), 54
StackLevel::set_unprocessed_components_end (C++ function), 50
StackLevel::setAsComponentSplit (C++ function), 53
StackLevel::StackLevel (C++ function), 50
StackLevel::super_component (C++ function), 50
StackLevel::unprocessed_components_end (C++ function), 50
StackLevel::unsetAsComponentSplit (C++ function), 53
StopWatch (C++ class), 54
StopWatch::getElapsedSeconds (C++ function), 55
StopWatch::interval_tick (C++ function), 55
StopWatch::setTimeBound (C++ function), 55
StopWatch::start (C++ function), 55
StopWatch::stop (C++ function), 55
StopWatch::StopWatch (C++ function), 54
StopWatch::timeBoundBroken (C++ function), 55
STR_DECIMAL_BASE (C macro), 68
SUCCESS (C++ enumerator), 56

T

T_TRI (C macro), 69
toDEBUGOUT (C macro), 69
TopTreeLiteral (C++ type), 73
TopTreeNodeType (C++ type), 57
TreeNodeIndex (C++ type), 73
TriValue (C++ type), 73

U

UpdateVarDescedantsList (C++ function), 61, 62

V

Variable (C++ class), 5
Variable::ante (C++ member), 5
Variable::decision_level (C++ member), 5
VariableIndex (C++ type), 73
varsSENTINEL (C macro), 69

X

X_TRI (C macro), 69