Charles University in Prague
Faculty of Mathematics and Physics

MASTER'S THESIS

Jan Kouba

Memory Representation for Model Checker of C/C++

Department of Software Engineering

Supervisor: RNDr. Ondřej Šerý
Study programme: Informatics, Software Systems

2009

I would like to thank to my supervisor Ondřej Šerý for his valuable advices and appreciated comments.

I declare that I wrote my master's thesis independently and exclusively with the use of the cited sources. I agree with lending and publishing the thesis.

In Prague, 9.12.2009 Jan Kouba

Contents

Název práce: Memory Representation for Model Checker of C/C++
Autor: Jan Kouba
Katedra (ústav): Katedra softwarového inženýrství
Vedoucí diplomové práce: RNDr. Ondřej Šerý
e-mail vedoucího: Ondrej.Sery@mff.cuni.cz

Abstrakt: V předložené práci je popsán návrh a C++ implementace nově vytvořeného paměťového modulu, který bude použit k reprezentaci paměti zkoumaného programu v GIMPLE Model Checkeru (explicit state model checkeru). Modul se liší od většiny ostatních code model checkerů v tom, že umožňuje uložit do simulované paměti libovolné C++ objekty splňující jisté rozhraní. To umožňuje ukládat například data hodnot spolu s jejich typy, symbolické hodnoty používané při symbolickém vykonávání programu nebo predikáty o hodnotách používané při predikátové abstrakci. Pro efektivní ukládání stavů, kanonikalizaci haldy a výpočet hashe stavu používá modul delta ukládání, inkrementální hashování a inkrementální kanonikalizaci haldy.

Klíčová slova: code model checking, reprezentace paměti, GIMPLE, C++

Title: Memory Representation for Model Checker of C/C++
Author: Jan Kouba
Department: Department of Software Engineering
Supervisor: RNDr. Ondřej Šerý
Supervisor's e-mail address: Ondrej.Sery@mff.cuni.cz

Abstract: We describe the design and C++ implementation of the newly created memory module (MM) in this work. It will be used in the GIMPLE Model Checker, an explicit state model checker, to represent the memory of checked program. MM differs from other code model checkers in the fact, that it stores ordinary C++ objects fulfilling a given interface as values. This allows to store, e.g., value data together with its type, a symbolic value used in a symbolic execution or a predicate over a stored value used in predicate abstraction. MM uses delta saving, incremental hashing and incremental heap canonicalization to save the state, canonicalize the heap and compute the hash of the state efficiently.

Keywords: code model checking, memory representation, GIMPLE, C++

Chapter 1

Introduction

Model checking is an algorithmic technique to verify a system description against specification. Given a system description and logical specification, the model checking algorithm proves that the system description satisfies the specification, reports a counter-example that violates the specification, or runs out of time or memory, since model checking is an undecidable problem in general.

Historically, model checking has been successfully used to find errors in hardware and protocol designs. Later, it was adapted to analyze software source code. First code model checkers like Java PathFinder 1 [1], JCat [2] or AX [3] transformed the source code into the input notation of their back end model checker (SPIN in all mentioned examples). The drawback of this approach was that some language constructs were hard to translate (e.g., dynamic memory operations). Later, *explicit state model checkers* like Java PathFinder[1] [4] or MoonWalker [13, 14] emerged. They do not transform the source code to some other notation, they rather search the program state space by directly simulating execution of the program. This removes the transformation step and moreover it allows the model checker to represent the *state* of the checked program more efficiently.

The state of a program is defined by the content of its modifiable memory, so integral part of every explicit state model checker is representation of the program memory. To lower memory consumption, explicit state model checker does not construct the whole state graph, but rather generates the graph on the fly by simulating execution of the program: The search is started in the initial state. Whenever a non-determinism is encountered dur-

[1]Successor of Java PathFinder 1.

ing the execution, the code model checker continues the search in one of the branches, but first it saves the current state, so it will be able to return to it later on and explore all the remaining branches. When the execution reaches a state that has been visited before (*explored state*), the search does not descend to the branches, but rather a state with some unexplored branches left is loaded and the search continues from there. When all branches of a saved state has been explored, it can be freed, since the model checker will never return to this state again. To sum up, explicit state model checker must be able to save current state, load it later on, free a saved state and also to maintain a set of explored states in order to perform the state space search. Note that the only two operations that must be supported by the set of explored states are addition of a state and test if the current state is in the set.

This thesis is a part of an effort to make the GIMPLE Model Checker (GMC), an explicit state model checker of GIMPLE. The input of GMC will be a GIMPLE representation of the program, which is a source language and target architecture independent representation of program source code internally used in GNU Compiler Collection (GCC) [5] compilers. Since GIMPLE is source language independent, GMC will be able to check programs written in any of the programming languages the GCC can compile, which are currently C, C++, Objective C, Fortran, Java and Ada. Also GMC will not have to parse directly the source code, which can be challenging, e.g., for C++ programs with templates.

The goal of this thesis was to design and implement in C++ a memory module (MM) for GMC (see Figure 1.1). MM is responsible only for the representation of checked program memory.

Because GMC does not exist yet, two other helper modules were created in its place (see Figure 1.1) to test MM and to verify, that MM provides enough functionality for GMC. The GIMPLE Iterator module transforms the GIMPLE representation of the program into a read-only representation (from now on called *GIMPLE++*), that is easier to work with from within C++. This module can be used by the future GMC. The second module, GIMPLE Interpreter, then uses the GIMPLE++ representation to interpret the program. This module also acts as an example of the usage of both MM and the GIMPLE Iterator module.

The rest of this text is organized as follows. In Chapter 2 is discussed, how existing code model checkers represent the memory of the programs they check. Chapter 3 contains the description of MM, that was implemented as a

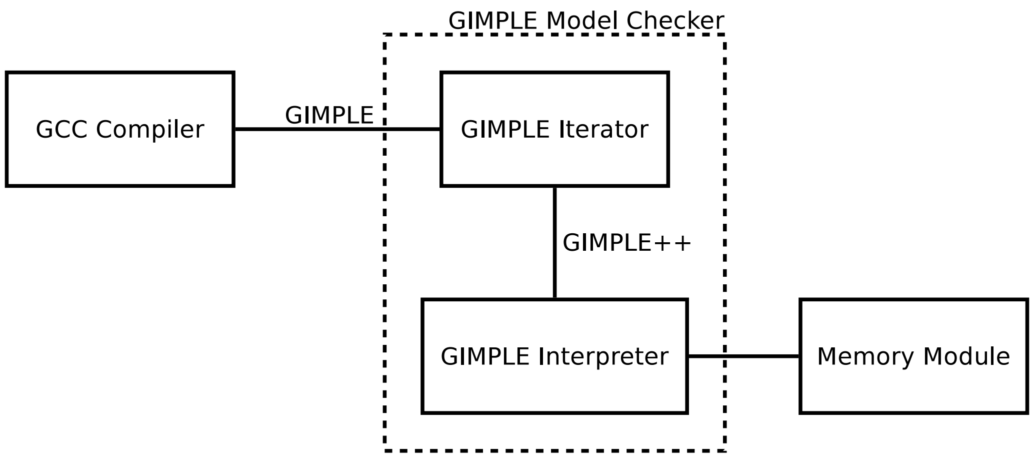

Figure 1.1: The diagram showing the modules created as a part of this thesis and the future GMC.

part of this thesis. Chapter 4 describes the GIMPLE Interpreter and Chapter 5 describes a GIMPLE Iterator Module.

Chapter 2

Related work

This chapter describes how the existing code model checkers represent the memory of the checked program. First, there is a brief description of the popular code model checkers. Then it is discussed, how they represent saved and explored states and how they deal with equivalent heaps.

2.1 Model checkers

2.1.1 CMC

C Model Checker (CMC) [7] is a code model checker of C and C++ programs used to check implementation of network protocols. It runs unmodified C or C++ code from the implementation. CMC can simulate multiple processes, where the state of the process consists of a copy of its stack, static data, heap and context registers.

2.1.2 ZING

ZING [11, 12] is a model checker that verifies ZING models, which support constructs that can be found in languages like Java or C#. First, the program needs to be translated into the ZING model, which is then compiled by ZING into the ZING object model, a CIL[1] object code that is used to explore the program state space.

[1]CIL stands for Common Intermediate Language (formerly called Microsoft Intermediate Language or MSIL) which is the bytecode used in Common Language Infrastructure

2.1.3 SPIN

SPIN [8] is a generic verification system, that accepts design specifications written in the verification language PROMELA (a Process Meta Language) and it accepts correctness claims specified in the syntax of the standard linear temporal logic. SPIN represents state as a simple sequence of bytes and heavily depends on it. SPIN does not have any explicit support for checking of programs that use dynamic memory.

As mentioned before, some code model checkers use SPIN or one of its extensions as their back end model checker.

2.1.4 dSPIN

“Its intention is to provide SPIN with a number of dynamic features which allow for object-oriented programs to be modeled in a natural manner and efficiently verified.” [9]

The state is represented by more variable sized vectors. Every simulated process has its own vector with static data and a stack. There is one global *heap vector* that contains all dynamically created objects and global data. Because SPIN represents the state as a sequence of bytes, all the vectors are linearized into a single state vector at every step of the model checker. In this linearization step the heap is canonicalized (see 2.4.5).

2.1.5 Java PathFinder

Java PathFinder (JPF) [4] is a code model checker of Java. It accepts Java bytecode as an input, so any language that can be compiled into the bytecode can be analyzed. JPF uses its own Java virtual machine to search program state space.

The state is composed of three components: *static area*, *dynamic area* and *thread list*. Static area is an array of entries, one for each class loaded. The entry contains the values of static fields of the class and monitor associated with it. Dynamic area is an array of entries, one for each object. The entry contains the values of the fields of the object and the monitor associated with it. Thread list is a list containing information about each thread. It contains ,e.g., the status of the thread, but it also contains all the stack frames created by method calls.

2.1.6 MoonWalker

MoonWalker [13][14] is a code model checker of CIL that is inspired by JPF. The state is represented in a very similar way as it is represented in JPF.

2.2 Memory representation

According to the experience from existing code model checkers, the biggest problem is limited memory and not the time. The code model checkers usually stop, because they have run out of memory. This is caused by the fact that the implementation of a problem contains more details than only its high-level description. Therefore the states that the code model checkers work with, when checking the implementation of the problem, are usually much bigger than the states that the model checkers work with, when checking only its high-level description. Also the state space of the implementation is usually much bigger, which increases the number of saved states and explored states, that must be maintained by the code model checker.

The state of the program is represented by the content of its memory, which – from the point of view of the program – usually consists of three different structures: *static data*, *stack for every thread* and *heap*. Static data occupy constant number of bytes and therefore is the simplest structure. It usually contains global variables and constants. Every thread has its own stack composed of stack frames. Stack frame is pushed onto the stack, whenever a function is called, and it is popped from the stack when the function returns. Heap is the most complicated memory structure. It allows the program to allocate and free arbitrary sized blocks of memory (*areas*). Heap is a place where all dynamically allocated data are stored.

The following text describes, how existing code model checkers deal with the limited memory, especially how they represent saved and explored states.

2.3 Saving states

The representation of saved states is influenced by the way the code model checkers search the state graph. The code model checkers like ZING, JPF and MoonWalker explore the state space in a depth first manner[11, 4, 13], while for example SPIN allows to use more complex algorithms[8].

When a depth first search algorithm is used to explore the state space, every time a state is loaded, it is a predecessor of the current state on the current execution path. This allows the code model checker to store all saved states on the stack. When a state should be saved, it is pushed onto the stack. Later, when it is loaded, all the states on the stack above the loaded state can be freed, because they are not needed any more. This approach, when the execution returns only to the states on the current execution path, is called *backtracking*.

2.3.1 Delta saving

Saving the whole state on a stack would be very inefficient, so all the three code model checkers that use backtracking (mentioned above) save only differences between states (*deltas*). When a state is about to be saved, only the delta is pushed onto the stack. When restoring the state, the modifications saved in the delta(s) are undone and the delta(s) are removed from the stack.

2.3.2 Collapse compression

Collapse compression (or *recursive indexing*) is another approach to lower the memory consumption. The code model checkers like JPF, MoonWalker and SPIN use this technique[4, 13, 19]. The principle of collapse compression is to split the program state into parts. The parts are then stored in a global pool, which assigns a unique index to every stored part. Once the part is stored in the pool, it is never removed and therefore its index remains the same. That means, the index of two parts is the same if and only if the two parts are the same. Then every state can be collapsed by substituting the parts by their indexes in the pool. E.g., both JPF and MoonWalker collapse the heap by storing whole areas in the pool. The collapse compression has the disadvantage, that all different parts (e.g., areas in case of JPF and MoonWalker) that appeared in some saved state are stored in the memory, even the ones that are not used in any saved state any more.

2.4 Detection of explored states

The simplest approach to detect explored states is to maintain a set of explored states and every time the model checker needs to test if a given state is explored, it tries to find the state in the set. If the state is found, it means that it has already been explored. If the state is not found, it means that it has not been explored yet and therefore is added into the set. With this approach the model checker must store all explored states, even though their content will never be used.

2.4.1 Hashing

To lower the memory occupied by the set of explored states, only hashes of the explored states can be saved, not the whole states. When the code model checker needs to check, if a state has already been explored, it calculates the hash of the given state first and then tries to find the hash in the set of hashes of explored states. Note that comparing only hashes and not the states as a whole can lead to hash collisions and therefore mark a state as explored, even if it has not been explored yet, but probability of such a collision can be made very small. All the code model checkers listed at the beginning of this chapter are using hashing.

2.4.2 Bitstate hashing

In order to save more memory, SPIN uses a technique called *bitstate hashing*[8]. This approach represents the set of hashes of explored states as a huge array of bits. Let a be the array of bits and s the number of elements in the array a. Then hash h is in the set represented by the array a if and only if $a[h \bmod s]$ is set to `true`.

2.4.3 Incremental hashing

Representing explored states only by their hashes greatly reduces the memory used by the code model checker, but there still stays the performance bottleneck that is the hash computation. When no optimizations are used, every time a state hash is computed, a whole state must be examined. This is mostly redundant work, because two consecutive states usually do not differ that much.

A technique called *incremental hashing*[15] can be used to speed up the hash computation. This technique assumes, that the hash of the whole state is composed of a hashes of its parts (in case of heap e.g. from *partial hashes* of areas). There must also exist a function, that given the hash of the previous state and the partial hashes of the modified parts before and after the modifications, computes the hash of the new state. When the hash of the state is computed, only the modified parts must be examined. The time needed to calculate the hash depends only on the size of modified parts and not on the size of the entire state.

An example of an incremental hash function used in MM can be found in Section 3.1.12.

2.4.4 Area placement independent program

In order to do model checking, a program being model checked must be *area placement independent*. This means that the behavior of the program must not depend on the placement of areas on the heap. If the program was area placement dependent[2], the code model checker would have to check every possible placement of the areas in order to validate the program, because the address on which a dynamically allocated area is placed is undefined.

Examples of problematic operations, that make a program area placement dependent, are as follows: In languages that allow low-level access to memory (like C and C++), these are comparison and computing a difference of pointers to different areas[3]. Java does not allow low-level access to memory, but there is a similar problem. The default implementation of the method `Object.hashCode()` can return various values between different invocations of the same program and therefore acts like the address of the object[4]. Therefore area placement independent program should not call the default implementation of `Object.hashCode()` at all.

2.4.5 Equivalent heaps

When a code model checker is checking a program that uses heap, it can happen that it comes across a state, that differs from some explored state only in the positions of areas on the heap. An example of such two states can be seen in Figure 2.1.

Since code model checkers usually represent the heap as a sequence of bytes, from the code model checker's point of view these two states are different, but from the checked program point of view they are the same, because

[2]A program that is not area placement independent is *area placement dependent*.

[3]C99 standard states, that such operations have undefined behavior, which means that performing such operation can even cause the program to crash.

[4]In fact `Object.hashCode()` is typically implemented by converting the internal address of the object into an integer.

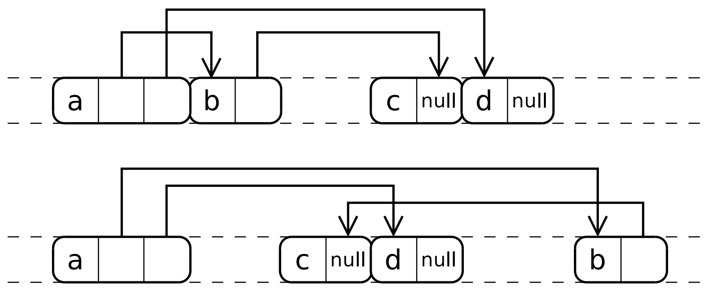

Figure 2.1: Two equivalent heaps.

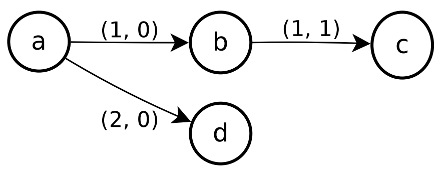

Figure 2.2: The heap graph for the heaps shown in Figure 2.1.

the program is area placement independent. This is caused by the fact, that the sequential representation of the heap contains excess information (in the form of area addresses) which has no influence on program behavior.

Heap graph

In order to get rid of this excess information, we can define the heap graph. The vertices are the areas on the heap. Each vertex in the heap graph is labeled by the non-pointer values stored in the area it represents. For each pointer p, that resides in an area u at an offset o_u and points to an area v to an offset o_v, there is the directed edge (u, v) labeled by the pair of values (o_u, o_v). The heap graph captures the state of the heap, but abstracts the absolute addresses of the areas. An example of the heap graph for the heaps shown in Figure 2.1 can be seen in Figure 2.2.

In order to capture the whole state of a program in the heap graph, we can emulate all program memory structures on the heap. All the global variables can be considered as a part of a *root* area. A thread stack can be represented by a linked list of stack frames, where each stack frame is an area. The root area then contains a constant sized array that represents the threads, where each element contains the pointer to the stack of the thread.

Now the heap graph represents the entire state of the program. Also the heap graph has a well defined root, which is the vertex representing the area that contains all the global variables.

Now we can say that two states are *equivalent*, if their (rooted) heap graphs are isomorphic[5]. Since checked programs are area placement independent, they can not differentiate between equivalent states, therefore it is sufficient if a code model checker explores only one state among all the equivalent states.

In order to achieve that, code model checkers try to rearrange the areas on the heap in order to form the same canonical representation for all equivalent states. They use two strategies: Some use a heuristic to place the area on the right position at the time it is allocated, while others exploit the heap graph and rearrange the areas right before the state hash is computed to form a really canonicalized heap.

Heap pseudo–canonicalization

JPF or MoonWalker use the first strategy [4, 14]. They try to place the same areas from different execution paths on the same address at the time, the areas are allocated. During the state space exploration when a new allocation is encountered, it is remembered where the area has been put. The next time the *same* allocation is made, the area is put on the same address. Two allocations are considered the same, if they were invoked by the same instruction I, executed by the same thread T and the number of executions of the instruction I by the thread T is the same. Note that when using this approach not all heap equivalences are found.

Heap canonicalization

dSPIN is an example of the code model checker that rearranges the heap right before the state hash is computed [10, 9] to form a really canonicalized heap. First, it sorts the areas on the heap in the order in which they are visited by the depth first traversal of the heap graph, where the edges are

[5]The graph isomorphism problem on undirected non-labeled graphs is the NP problem not known to be solvable in polynomial time, but a graph isomorphism for two rooted directed labeled graphs that have unique labels for all edges originating from each vertex can be found easily in polynomial time.

traversed in the order given by their labels[6]. Then the areas are laid down one after another in that order. Note that all the equivalent heaps are rearranged to the same canonical representation.

This simple algorithm has the disadvantage that it causes relocation of many areas and therefore hinders the effect of incremental hashing. Even small changes to the objects on the heap can lead to relocation of many areas in the canonicalized heap. E.g., when a new area that is visited early by the depth first search appears, all the following areas must be moved in the canonicalized memory[7]. This will lead to the recomputation of the partial hashes of all the moved areas, because the partial hash of an area must depend on its position in the simulated memory to minimize the number of state hash collisions.

Incremental heap canonicalization

Zing and CMC use the *incremental heap canonicalization algorithm*[15], that also detects all equivalent heaps, but does not hinder the effect of incremental hashing that much. This algorithm tries to do as little relocations as possible and hence minimize the number of areas whose partial hashes must be updated. The main idea of the algorithm is to compute the canonical address of an area from its *shortest path* to the root area. When a transition makes only small changes to the heap, the shortest path to the root area of most of the areas is likely to remain the same. Therefore the code model checker needs to examine only those areas, whose shortest path to root has changed.

The canonical addresses of the areas can be computed by performing the breadth first traversal of the heap graph, where the order in which the output edges are traversed is given by the edge labels. The root area is placed at some fixed canonical address. Let $\mathbb{P}$ be a set of all memory addresses. When a vertex (area) v is discovered for the first time by following an edge $e = (u, v)$ with a label (o_u, o_v), a canonical address $C(r, s)$ is assigned to the area v, where r is the canonical address where the pointer corresponding to the edge e is stored (the address can be computed as the sum of the canonical address of the area u and the offset o_u), s is the size of the area v and $C : \mathbb{P} \times \mathbb{N} \rightarrow \mathbb{P}$

[6]All outgoing edges of a vertex have different labels; more precisely, they have different first values in their labels, therefore it suffices to compare only the first values of the labels.

[7]Code model checkers (including MM) usually do not really copy the areas. They only update the canonical addresses of the beginning of the areas and then use them to update the partial hashes of the moved areas (and values).

is a function such that

$$\forall p_1, p_2 \in \mathbb{P}, \forall s_1, s_2 \in \mathbb{N} : C(p_2, s_2) < C(p_1, s_1) \vee C(p_1, s_1) + s_1 \leq C(p_2, s_2) \tag{2.1}$$

holds true[8]. Note that the areas whose shortest path to root have not changed are really placed to the same addresses by this algorithm.

The last problem is how to define the function C. The function can not be expressed in a closed form, but it can be constructed incrementally. Whenever it is called with a new pair of parameters, a value fulfilling Equation 2.1 is returned. The next time the function is called with the same pair of parameters, the same value is returned. The easiest way, how to achieve that the returned values fulfil Equation 2.1, is to maintain the pointer p_f to the next free canonical address. Every time the function is called with a new pair of parameters (p, s), the value of the pointer p_f is returned and p_f is incremented by s.

[8]This says that the areas in the canonicalized memory do not overlap.

Chapter 3

Memory module

The description of MM can be found in this chapter. Section 3.1 describes the design decisions made when implementing MM. Section 3.2 describes the interface of MM, while Section 3.3 describes the implementation of MM. Section 3.4 discusses the space and time complexities of MM, in Section 3.6 is the benchmark of MM and in Section 3.5 it is noted how was MM tested.

3.1 Design

The decisions made when designing MM are described in this chapter. First, the requirements on MM are listed. Then the methods that are used to access values in simulated memory are discussed and how are pointers represented is show. Finally, it is stated how MM saves states, detects explored states and canonicalizes the heap.

3.1.1 GIMPLE

GIMPLE is the source language and target architecture independent representation of a program source code internally used in GCC [5] compilers. Since GCC supports many input languages and target architectures, the compilers are split into three parts: The *front-end* parses a source code of a program and transforms (*gimplifies*) it into the GIMPLE representation. Then the *middle-end* performs most analysis and optimizations on the GIMPLE representation. Finally, the *back-end* uses the GIMPLE representation to generate an executable code for a target architecture. This division allows GCC to maintain only one back-end for each target architecture, one front-end for each input programming language and single middle-end. Currently, GCC supports the front-ends for C, C++, Objective C, Fortran, Java and Ada.

In GIMPLE, program statements are expressed by tuples with no more than three operands (with some exceptions like function calls). Temporary variables are introduced to hold intermediate values needed to compute complex expressions.

3.1.2 Requirements

MM must support two kinds of operations: *memory manipulation* operations and *state manipulation* operations. The memory manipulation operations allow GMC to manipulate with the values in the simulated memory. In order to compile C and C++ programs, GIMPLE supports low-level access to memory, therefore MM must allow low-level (pointer) access to all memory structures (global data, stack, heap). It must also detect and forbid all attempts to do a memory operation with undefined behavior. The examples of such operations are access to unallocated memory and reading of uninitialized memory. MM must also detect memory operations with area placement dependent behavior. The examples of such operations are a computation of the difference between two pointers pointing to different areas and moving a pointer between different areas.

The state manipulation operations allow GMC to manipulate with the state of the simulated memory as a whole. These operations will be used by GMC to do a state space search. MM must support saving and loading the current state and it must also allow GMC to detect the explored states.

3.1.3 Simplification of program memory structures

To ease the implementation of MM, it simulates only the most complex memory structure, the heap. The other memory structures (global data and stack for every thread) can be easily emulated on the heap. An example of how to do this can be seen in Section 4.1.2.

3.1.4 High-level vs. low-level memory access

Programming languages allow two kinds of memory access: low-level and high-level. Language with the low-level memory access allows the program to load and store values at any memory addresses (by using pointers). On the other hand, a programming language that allows the high-level memory access abstracts from the details of how are values laid out in the memory and lets the programmer to access the values by using variables. E.g., C and C++ are the languages that provide both the high-level and low-level memory access, while Java supports only the high-level memory access.

Since GCC can compile C++, GIMPLE allows both the high-level and low-level memory access, but MM provides only low-level memory access methods. The first reason why MM does not provide methods for high-level memory access is, that we do not want MM to depend on GIMPLE. It is required to know the types of variables and the sizes and alignment requirements of these types in order to implement the high-level memory access, therefore it is required to access the information in the GIMPLE representation of the program. The other reason is, that GMC can have special requirements on which variables to store in the simulated memory[1] and on the layout of the variables within an area, therefore it is natural, that the high-level access will be implemented in GMC (or in some other module).

3.1.5 Memory manipulation methods

MM provides only five methods to support the low-level memory access:

`alloc()` allocates a new area and returns a pointer to the beginning of the allocated area.

`free(p)` frees the area that begins at the address `p`.

`save(p, v)` stores the value `v` at the address `p`.

`load(p)` loads the value stored at the address `p`.

`loadOverlapping(p, s)` loads all the values that overlap with the range of addresses $\langle \texttt{p}, \texttt{p+}_i\texttt{s})$.

The detail description of all the memory manipulation methods can be found in Section 3.2.

[1]Not all GIMPLE variables must reside in the simulated memory. In the compiled program, values of some variables are never written into the memory, but they are kept only in CPU registers. So GMC do not have to store these variables in the simulated memory neither.

3.1.6 Type safety

GIMPLE supports the low-level memory access and therefore allows a program to perform a type unsafe operations. A type unsafe operation is every operation that tries to interpret data as a value of different type. Type unsafe operations can be avoided in most cases (but sometimes they are justifiable[2]), therefore MM must allow GMC to detect all type unsafe operations[3].

To detect a type unsafe operation GMC must be able to detect, if the data loaded from the simulated memory are really of the expected type. Programs using the low-level memory access are able to store a value of any type to any writable address in the memory, therefore GMC can not make any assumptions about what type of data can be stored at a specific address. This means, that both the type and the data of a value must be stored into and loaded from the simulated memory.

3.1.7 Values

Since the data and the type of a value are tied together, MM allows to store any object fulfilling a given interface into the simulated memory. MM does not need to understand the type or the data of a stored value. It only needs to know the size of a value being stored and, if it is a pointer, than also to which address it is pointing. MM needs to know about all the pointers stored in the simulated memory to be able to perform the canonicalization algorithm and to check for unreachable areas. Representing stored values as GMC supplied objects will allow GMC to store anything as a value (e.g. a symbolic value used in a symbolic execution or a predicate over a stored value used in predicate abstraction) not only its bytes. The other advantage of this approach is, that MM allows to simulate programs compiled for a different architecture than the one it is running on, since the sizes of values and the layout of values in areas are completely under the control of GMC.

[2]E.g., device drivers and network protocol implementations often need to serialize or deserialize a structure, which is done most efficiently and easily using some kind of a type unsafe operation.

[3]It is up to GMC when or if ever it will allow type unsafe operations.

Value overwriting

When an old value is overwritten by a new one, it is removed from the simulated memory. The old value is removed, even if it is overwritten only partially, because values are opaque to MM, and therefore it has not enough information to update the old value. However GMC can use the method `loadOverlapping()` to load all values that would be overwritten and than it can update the values itself.

3.1.8 Pointers

MM provides a pointer type for representing the memory addresses. It is used as the parameter type and the return type of memory manipulation methods, but it will be probably used to represents pointers in GMC as well. Instances of the pointer type (pointers) represent *raw* memory addresses, which means there is no type information stored in them. The pointer type supports the operators $\texttt{+}_i$, $\texttt{-}_i$ (adding/subtracting a byte offset), $\texttt{-}_p$ (computing the difference between two pointers), `==`, `!=`,`<`, `<=`, `>` and `>=` with the semantics similar to the same operators on the type `char*` in C.

In order to detect area placement dependent operations, MM must be able to check, if the relative positions of a pointer p and an area A would be the same in all the equivalent heaps (for short, if the pointer p *belongs to* the area A). This is needed, because a value stored in an area A can be accessed only by using pointers that belong to the area A. Similarly, MM must check, if operands of $\texttt{-}_p$, `<`, `<=`, `>` and `>=` operators belong to the same area. If they do not belong to the same area, the result of the operation would not be the same in all the equivalent heaps and therefore the operation is area placement dependent.

Knowledge of the area a pointer belongs to is also needed to implement the operators `==` and `!=`. Two pointers are equal (`==`) if and only if they belong to the same area and point to the same address. Two pointers are unequal (`!=`) if and only if they are not equal.

A pointer must hold the information about the area it belongs to, because it is impossible to determine this information only from the memory address the pointer is pointing to. Consider a program that has two areas A and B and the pointers to their beginnings a and b. The program creates the new pointer p by adding the size of the area A to the pointer a, so that the pointer p points just beyond the area A. Now, if the area B was placed right after the area A (as shown in Figure 3.1), the pointer p and the pointer b

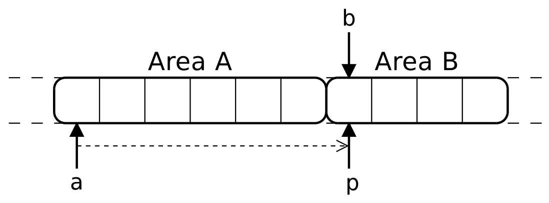

Figure 3.1: Example of the two pointers p and b pointing to the same address, but belonging to the different areas.

would both point to the same address, but the pointer p belongs to the area A and the pointer b belongs to the area B.

Because every pointer must contain the information about the area it belong to, new pointers can be obtained only from other pointers using the operators $+_i$ and $-_i$ (the new pointer belongs to the same address as the pointer it has been created from) and by calling the method `alloc()` (the new pointer belongs to the allocated area). MM also provides GMC with a null pointer.

Since it is forbidden to move a pointer between areas, the program does not need to use the operators $+_i$ and $-_i$ to produce a result that does not point into or just beyond the area it belongs to[4], therefore the pointer type detects these *pointer overflows*. Because this checking is done early, GMC will be informed about the program misbehavior much closer to the source of the error, than if the checking was done only in the memory manipulation methods. In order to do the pointer overflow checking, every pointer must be able to find out the size of the area it is pointing to.

3.1.9 Area life-cycle

The life-cycle of an area during the execution of a program can be seen in Figure 3.2. Once an area is allocated (transition 1), values can be stored in it. When the area is freed (transition 2), there can still be pointers to it in the simulated memory, so MM must keep the freed area in the state, in order to detect access to it. It also matters if two pointers are pointing to the same freed area, or to two different freed areas[5]. All values stored in an

[4]The behavior of such operation is marked as undefined by the C99 standard.

[5]E.g., the result of the comparison of two pointers pointing to the beginning of the same freed area differs from the result of the comparison of two pointers pointing to the

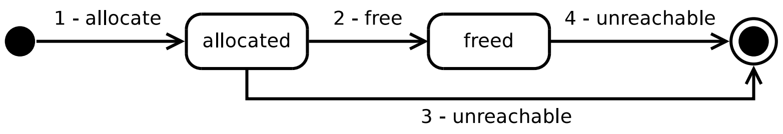

Figure 3.2: Area life-cycle.

area must be discarded after it is freed, because the content of a freed area is undefined. An area is removed from the state once it becomes unreachable[6] (transitions 3 and 4), because it has no influence on the program behavior. When an allocated area becomes unreachable (transition 3), it is a sign that the program leaks memory (if it is not a language with garbage collection), so MM reports the area to GMC.

3.1.10 Saving states

As stated before, the biggest problem of code model checkers is, that they need huge amount of memory, so MM should be optimized for space.

Delta saving

To lower memory consumption, MM uses delta saving. When a state is saved, only a delta containing the modifications since the last saved state is stored. The existing code model checkers like JPF and MoonWalker usually save in the deltas the whole modified areas, because they represent the area as a simple sequence of bytes and copying a simple sequence of bytes is simple and efficient. But MM represents values stored in the simulated memory as opaque objects, so the representation of the area is more complex than a simple sequence of bytes, therefore copying of the area would not be so fast. MM takes the advantage of the more complex structure of the area

beginning of two different freed areas.

[6]An area A is *reachable*, if there is a path from the root area to the area A in the heap graph. Otherwise it is *unreachable*.

and stores only the modified values in the deltas instead of saving the whole areas.

Because MM creates the deltas with value granularity, the deltas do not contain any redundancies, unlike the other code model checkers that store the whole areas in the deltas. This means that MM does not need to use techniques like collapse compression to remove these redundancies.

Since MM uses delta saving, loading of saved states is limited to backtracking only. MM maintains a stack of saved states and it allows GMC to save the current state on the stack (`push()`), remove the top state from the stack (`pop()`) and restore the current state to the top state on the stack (`backtrack()`). A detail description of the state manipulation methods can be found in Section 3.2.

3.1.11 Detecting explored states

GMC must be able to maintain the set of already explored states. To limit the memory used to represent the set of explored states it should store only the hashes of the states (as all other code model checkers do), or even use bitstate hashing. In order to do that, MM must be able to compute the hash of the state.

3.1.12 State hash computation

MM uses incremental hashing to speed up the state hash computation. The state hash function used in MM is based on the hash function described in [15][7]

The state is defined by the values stored in the memory, but also by the positions and sizes of all the reachable non-freed areas. The positions and sizes of the reachable non-freed areas are needed, because two states differing only in one area that is freed in one state and non-freed and empty

[7]The hash function described in [15] is defined as

$$H = \sum_{p \in \mathbb{P}} V(p) \cdot R(p) \pmod{M}.$$

where $\mathbb{P}$ is the set of all memory addresses, $\mathbb{H}$ is the set of all the state hash values, $V(p)$ is the value of the byte at the address p, $R : \mathbb{P} \to \mathbb{H}$ is a function assigning random values to memory addresses and $M \in \mathbb{H}$ is a big prime number. The bytes with undefined values are set to 0.

in the other should have different state hashes, because the program behavior would be different for such two states. E.g., it is perfectly valid to store a value into the area in the second state, but it would be an error to store a value into the area in the first state.

To minimize the probability of a state hash collision, a state hash is composed from partial hashes of both areas and values and the partial hashes involve in their computation the position, type and data of the value and the size and position of the area respectively. The definition of the state hash function used in MM can be found in the following text.

State hash function

Let $\mathbb{H}$ be the set of all state hash values, $M \in \mathbb{H}$ be a big prime number, $V(s)$ be the set of all the values in a state s and $A(s)$ be the set of all the non-freed areas in a state s. Also assume, that $H_a(a)$ is the area partial hash of an area a and $H_v(v)$ is the value partial hash of a value v (the partial hashes are described later). The state hash of a state s is computed as:

$$H(s) = \sum_{a \in A(s)} H_a(a) + \sum_{v \in V(s)} H_v(v) \pmod{M} \tag{3.1}$$

With such state hash function it is easy to update a state hash, knowing which values were removed and added and which areas were allocated and freed or became unreachable. If s and s' are states, then the state hash $H(s)$ of the state s can be expressed as:

$$\begin{aligned} H(s) \quad = \quad & H(s') \\ & + \sum_{a \in A(s) \setminus A(s')} H_a(a) - \sum_{a \in A(s') \setminus A(s)} H_a(a) \\ & + \sum_{v \in V(s) \setminus V(s')} H_v(v) - \sum_{v \in V(s') \setminus V(s)} H_v(v) \pmod{M} \end{aligned} \tag{3.2}$$

In this way, the time needed to update the state hash depends only on the number of the modified values and areas.

The area partial hash must involve the information about the position and size of the area, so it is defined like this. Let $\mathbb{P}$ be the set of all memory addresses, $S(a)$ be the size of an area a, $B(a) \in \mathbb{P}$ be the address of the beginning of an area a and $R_a : \mathbb{P} \to \mathbb{H}$ be a function assigning random values to memory addresses. Then the area partial hash of an area a is computed as:

$$H_a(a) = R_a(B(a)) \cdot S(a) \pmod M \tag{3.3}$$

The value partial hash must involve the information about the data, type and position of the value, so it is defined in the following way. Let $B(v) \in \mathbb{P}$ be the address of the beginning of a value v, $R_v : \mathbb{P} \to \mathbb{H}$ be a function assigning random values to memory addresses. Then the value partial hash of a value v is computed as:

$$H_v(v) = (R_v(B(v))) \cdot H_o(v) \pmod M \tag{3.4}$$

where $H_o(v)$ is the *object hash* of the value v. The only remaining question is, how to compute the object hash $H_o(v)$.

The object hash computation depends on the kind of the value. If the value is a non-pointer value, MM does not know anything about the data or the type of the value, therefore the object hash of the value must be supplied directly by the object that represents the value. If the value is a pointer value, MM knows its data (the address where it points) but does not know its type (the other information contained in the object representing the value), so GMC supplied object must provide the *type hash* for the type of the value it represents. The object hash is then computed in MM by combining the type hash and the address the value it pointing to. How are the address a pointer value is pointing to and the object hash of a non-pointer value passed to MM is described in Section 3.2.

3.1.13 Heap canonicalization

GMC should examine only one state among all the equivalent states to speed up state space exploration (see Section 2.4.5). MM must return the same state hash for all the equivalent states, because GMC will use the returned state hash to detect explored states.

MM canonicalizes the heap before every state hash computation in order to return the same state hash for all equivalent states. Because MM uses incremental hashing, the canonicalization step should move as few areas as possible (see Section 2.4.3). The best algorithm for this is the incremental heap canonicalization algorithm (see Section 2.4.5), so MM uses it. In order to implement the incremental heap canonicalization algorithm, MM must know about all the pointer values stored in the memory, to be able to walk the heap graph.

3.2 Interface

The interface of MM is described in this section. Only the classes and the methods that are meant to be used directly by GMC are described here. All these classes, types and functions are in the namespace `mmodule`. For a more detailed description of the classes, types and functions see the Doxygen documentation on the attached CD.

The class `MemoryState` is the main class of MM interface. It represents the current state of the simulated memory, but also all the saved states. The methods of this class can be divided into two groups: The memory manipulation methods and state manipulation methods. The memory manipulation methods are used to inspect and modify the values and areas in the current state, while the state manipulation methods are used to save and load the states and to compute the hash of a state.

The memory manipulation methods come in two forms: templated and non-templated. The memory manipulation methods of the templated form allow GMC to work with the areas that have fixed layout (known at the compile time of MM), while the methods of the non-templated form allow to store any value at any offset within the area. The non-templated methods are needed to simulate the execution of a program, while the templated methods considerably simplify the manipulation with internal areas, that are not accessible to a simulated program.

The methods of the templated form allow to store any objects in the simulated memory, not only objects derived from the interface `PtrContent` or `NonPtrContent`. The objects are automatically wrapped into a helper classes that implements the interface `PtrContent` or `NonPtrContent`.

The rest of this section is organized as follows. First, the requirements on the objects representing stored values are stated. Then the classes `Ptr` and `Ref` representing pointers are introduced and finally the main class `MemoryState` is described.

3.2.1 Values

MM represents the values stored in the simulated memory as arbitrary objects implementing the interface `PtrContent` or `NonPtrContent`. If GMC wants to save an object that represents a pointer value, the object must implement the interface `PtrContent`. If GMC wants to save an object that represents a non-pointer value, the object must implement the interface `NonPtrContent`.

Content class

The interface `Content` is the common parent of the interfaces `PtrContent` and `NonPtrContent`. The objects supplied by the GMC must *not* implement directly the interface `Content`, rather they must implement one of the interfaces `PtrContent` or `NonPtrContent`. The interface `Content` contains only one method to implement, `getUnitSize()`. This method returns the size of the value in bytes.

PtrContent class

The interface `PtrContent` is derived from the interface `Content` and contains two additional methods to implement, `getPtr()` and `getTypeHash()`. The method `getPtr()` returns the address to which the pointer represented by the object is pointing, while the method `getTypeHash()` returns the type hash of the value the object represents.

NonPtrContent class

The interface `NonPtrContent` is derived from the interface `Content` and contains only one additional method to implement, `getHash()`. This method returns the object hash of the non-pointer value represented by the object.

3.2.2 Ptr class

The class `Ptr` represents a memory address. It is represented as a pair (area; offset within area). The class `Ptr` has only one no-argument public constructor, which creates a `null` pointer. An objects of the class `Ptr` is returned by the non-templated form of the method `MemoryState::alloc()` (see Section 3.2.4).

The class `Ptr` has `+=`, `-=`, `+` and `-` operators, that take `size_t` as their second argument. These operators move a pointer by the number of bytes specified in the second argument.

Two pointers can be compared for equality or inequality. Two pointers pointing to the same area can be compared by `<`, `<=`, `>` and `>=` operators. The result of the comparison of their offsets is returned. The exception `UndefinedPtrOperationException` is thrown, when an attempt is made to compare two pointers that belong to different areas.

The class `Ptr` supports computation of the difference between two pointers using the operator `-`, but they must point to the same area. As the result the difference in bytes is returned. If the two pointers belong to different areas, the exception `UndefinedPtrOperationException` is thrown.

The class `Ptr` contains few other utility methods. The method `isNull()` checks, if the pointer is `null`, the method `getAreaId()` returns the ID of the area to which the pointer is pointing and the method `getOffset()` returns the offset within the area to which the pointer is pointing.

3.2.3 Ref class

The class `Ref` represents a reference to an area allocated by the templated form of the method `MemoryState::alloc()`. The class has one template parameter `S` that must be a POD class[8]. `S` specifies the types and positions of the objects that can be stored in an area. Each field of the class `S` that has a type `T` specifies, that an object of the type `T` can be stored in the area at the same offset, as is the offset of the field in the class `S`. The size of the value the object represents is `sizeof(T)` bytes. This means that the layout of the objects in the area depends on the architecture on which is the MM compiled, but this is not a problem, because the areas allocated by the templated form of the method `MemoryState::alloc()` are not accessible from the simulated program.

An object of the class `Ref` (reference) does not represent the pointer to a specific address, but rather it represents the pointer to an area. When a value needs to be loaded from or stored into an area, the offset must be specified as a pointer to the member of the class `S`.

In order to store an object of the type `T` into the simulated memory using the templated form of the memory manipulation methods, the type `T` must fulfil some requirements: The expression `boost::hash<T> ()` must be a valid expression returning a functor that can compute the *data hash* of an

[8]POD stands for Plain Old Data. For example every C structure is a POD class in C++.

object of type `T`[9], or `T` must be the class `Ptr` or `Ref`. Note that the data hash returned by the functor does not need to depend on the type of the object, because when the wrapper class computes the object hash, it combines the data hash returned by the functor with the hash of the type `T`[10]. To extend `boost::hash()` function to cover another type `T`, a function

```
size_t hash_value (const T& value);
```

can be defined. The function must have exactly this signature and exactly the same name. The declaration of the function must be available at the point, where the templated form of `MemoryState::store()` method is called. The next requirement on the type `T` is, that it must be copy constructible, because the copy of a value passed to the templated form of the method `MemoryState::store()` is stored in the simulated memory. An example of a simple class that can be used as the template parameter of `Ref` class can be seen on Example A.1.2;

Only the `Ptr` and `Ref` objects are interpreted as the pointer values, when they are stored using the templated form of `MemoryState::store()`. The objects of all other types are considered non-pointer values.

The class `Ref` has only one no-argument constructor, that creates the `null` reference, which represents the reference that does not point to any area. The class supports only two operators `==` and `!=`. They check if two references point to the same area. Whether a reference is `null` can be found out using the method `isNull()`.

3.2.4 MemoryState class

The class `MemoryState` is the main class of MM's interface. The object of this class represents the current state of the simulated memory, but also all the saved states.

Memory manipulation methods

The current state can be changed by the methods `alloc()`, `free()`, `store()` and `setRootArea()` and inspected by `load()`, `loadOverlapping()` and

[9]All the primitive types fulfil this requirement as well as many classes from STL. For a more detailed information see the boost documentation[17].

[10]The hash of a type is computed from the address of the `type_info` object that is returned by the call to `typeid()`.

`getSize()`. The methods `alloc()`, `free()`, `store()`, `setRootArea()` and `load()` exist in two forms: templated and non-templated.

The non-templated methods take as a parameter or return an object of the class `Ptr`. These methods are used to save and load values, whose type is *not* known at the compile time of MM. These methods allow to store values of any size to any offset within an area. These methods will be typically used for storing and inspecting the values of a simulated program. The non-templated methods are described in Table 3.1 and their usage can be seen on example in Appendix A.1.1.

When loading a value from the memory using the non-templated form of the method `load()`, a constant pointer to `Content` is returned, because either a pointer or a non-pointer value can be returned. The exact type of the value can be found out at runtime using `dynamic_cast` or `typeid`.

The templated methods take as a parameter or return an object of the class `Ref`. These methods are used to store and load values, whose sizes and positions are know at the compile time of MM. They exist to save GMC from dynamic casting the `Content` objects returned from the non-templated forms of the memory manipulation methods and from wrapping every value it wants to store into an object derived from `PtrContent` or `NonPtrContent` class. These methods will be typically used for the manipulation with the internal data of GMC, that are not accessible to the simulated program, but are also part of the state. The manipulation with the areas containing the internal data is greatly simplified when using the templated methods. The templated methods are described in Table 3.2 and their usage can be seen on example in Appendix A.1.2.

There is no way how to convert the class `Ref` to the class `Ptr` or vice versa, so once an area is allocated, it can be later used only by the methods of the same form as was the method `alloc()` that was called to allocate the area.

MM must know which area is the root area, so it can perform the incremental heap canonicalization algorithm. The methods `setRootArea()` exist for this purpose. One of these methods must be called exactly once prior to the first call to `push()`. The root memory area must not be freed.

Function	Description

`Ptr`
`alloc(size_t size)`
	Allocates an area of `size` bytes and returns the pointer to the beginning of the allocated area.

`void`
`free(Ptr ptr)`
	Frees the area to which `ptr` is pointing. The pointer `ptr` must point to the beginning of the area.

`void`
`setRootArea(Ptr ptr)`
	Sets the area to which the pointer `ptr` is pointing as the root area.

`void`
`store(Ptr p,`
`AP<Content> v)`
	Stores the value represented by the object `v` on address `p`. (`AP` is a shorthand for `std::auto_ptr`)

`const Content*`
`load(Ptr ptr)`
	Loads the value from the address `ptr`.

`std::pair<CI, CI>`
`loadOverlapping(Ptr f,`
`size_t s)`
	Returns the range of values that overlap (even partially) with the range $\langle f; f + s)$. (`CI` is a shorthand for `MemoryState::ContentIterator`.)

`size_t`
`getSize(Ptr ptr)`
	Returns the size of the area to which `ptr` is pointing.

Table 3.1: The non-templated state manipulation methods.

Function	Description

`Ref<T>`
`alloc<T>()`
	Allocates an area that can hold fields of class `T`.

`void`
`free(Ref<T> ref)`
	Frees the memory area referenced by `ref`.

`void`
`setRootArea(Ref<T> ref)`
	Sets the area referenced by `ref` as the root area.

`void`
`store<T, V>(Ref<T> r,`
`            V T::* f,`
`            const V v)`
	Stores the value represented by the object `v` into the area referenced by `r` at the offset equal to the offset of the field `f` in the class `T`.

`const V&`
`load<T, V>(Ref<T> ref,`
`           V T::* f)`
	Loads the value from the area referenced by `ref` stored at the offset that is equal to the offset of the field `f` in the class `T`.

Table 3.2: Templated state manipulation methods.

State manipulation methods

The saved states are manipulated using the methods `pop()`, `push()` and `backtrack()`. The hash of the state is returned by the method `hash()`. These methods are described in Table 3.3 and Appendix A.2 demonstrates their usage.

From GMC's point of view, MM saves the states on the stack. The method `push()` saves the current state on the stack and the method `pop()` removes the top state. When `backtrack()` is called, the current state is restored to the top state on the stack.

The `MemoryState` class supports only getting the hash of the top saved state by calling the method `hash()`. If GMC wants to get the hash of the current state, it must save the state first by calling `push()` and then call `hash()`, which will return the state hash. If GMC does not want to backtrack to the current state later on, it can call `pop()` right away.

Exceptions

The memory manipulation methods of the class `MemoryState` do not prevent GMC from doing an undefined operation, like freeing an already freed area or saving a value outside the bounds of an area. However `MemoryState`

Function	Description
`void push()`	Saves the current state on the top of the stack of saved states.
`void pop()`	Removes the top state from the stack of saved states.
`void backtrack()`	Restores the current state to the top state on the stack of saved states.
`Hash hash()`	Returns the hash of the top state on the stack of saved states.
`bool hasSavedState()`	Checks if there are any states on the stack of saved states.
`size_t savedStateCount()`	Returns the number of the states on the stack of saved states.

Table 3.3: Saved states management methods.

detects whenever GMC tries to do such undefined operation and throws an appropriate exception. The exceptions are described in Table 3.4.

MemoryAccessListener interface

The class `MemoryState` needs a way how to inform GMC about memory leaks. The `MemoryAccessListener` interface exists for this purpose. An object implementing this interface is passed to the constructor of the class `MemoryState` and the created object than uses the listener to inform GMC about memory leaks. Whenever an unreachable area that was not freed is detected, the method `unreachableArea()` is called on the listener. The unreachable areas are detected and reported during the execution of the method `MemoryState::push()`.

3.3 Implementation

An overview of the implementation of MM is in this chapter. On a simple example, we demonstrate how the MM represents the current state and the deltas. Then the main classes and some of their methods are described. This chapter is important for anyone who wants to extend or modify MM. For a detailed description of all the types and methods see the Doxygen documentation on the attached CD.

Exception	Description
`NullPointerException`	Thrown when trying to dereference the `null` pointer or the `null` reference.
`DeletedAreaAccessException`	Thrown when trying to access an already freed area.
`NotAreaBeginException`	Thrown when a pointer does not point to the beginning of a memory area, but it should.
`OutOfBoundsException`	Thrown when trying to store or load a value that does not lie within the bounds of an area.
`UndefinedMemoryLoadException`	Thrown when trying to load an undefined value. The value on the address a is *defined*, if `MemoryState::store()` was called before to store the value at the address a, and the value was not overwritten (even partially). Otherwise the value is *undefined*.

Table 3.4: Exceptions thrown by `MemoryState` methods.

```
struct S {
    int i;
    Ref<S> ref;
};

void test (MemoryAccessListener* l)
{
    MemoryState ms (l);
    Ref<S> a1 = ms.alloc<S> ();
    Ref<S> a2 = ms.alloc<S> ();
    ms.setRootArea (a1);
    ms.store (a1, &S::i, 1);
    ms.store (a1, &S::ref, a2);
    ms.push ();

    Ref<S> a3 = ms.alloc<S> ();
    ms.store (a1, &S::ref, a3);
    ms.free (a2);
    ms.push ();
}
```

Figure 3.3: MM usage example.

3.3.1 Example

We start with a simple example of the usage of MM. The code of the example is shown in Figure 3.3. It simply allocates two areas, sets the first area as the root area, writes the integer and the pointer (that is pointing to the second are) into the first area and saves the current state using the method `MemoryState::push()`. In the second part, it allocates the third area, overwrites the pointer in the first area with the pointer to the third area, frees the second area and saves the current state.

Figure 3.4 shows the objects used by MM that are used to represent the current state and the delta. The black objects and associations show the objects right before the first call to `MemoryState::push()`, while the gray objects and associations show the objects that were created by the first call to `MemoryState::push()`.

Figure 3.5 shows the objects used to represent the current state and the delta after the execution of the second part of the example (the delta created in the first part is omitted). The black objects and associations show the objects right before the second call to `MemoryState::push()`, while the

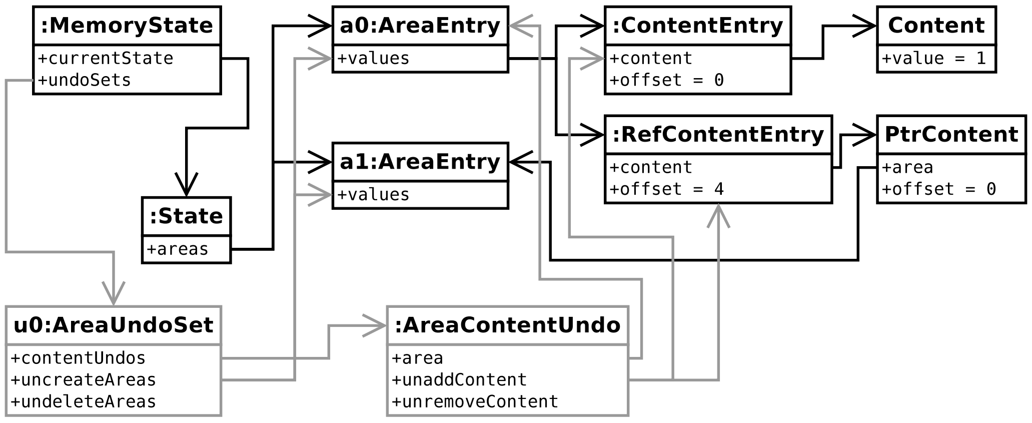

Figure 3.4: The objects after the first part of the example in Figure 3.3.

gray objects and associations show the objects that were created by the second call to `MemoryState::push()`.

The diagrams showing the objects are very simplified. They do not contain many associations that exist between the objects and attributes. Also there should be the objects implementing the interfaces `PtrContent` and `NonPtrContent` and not the interfaces themselves. Despite the simplifications and inaccuracies, the diagrams give a good overview of the internal classes used to store the current state and the deltas.

As can be seen, the current state is represented by the class `State`. It contains the list of areas that are reachable from the root area (including the freed areas). The area is represented by the class `AreaEntry`, which contains the values stored in it. The user supplied objects representing the values (implementing the interface `Content` or `PtrContent`) are wrapped into the `ContentEntry` or `RefContentEntry` objects.

The delta is represented by the class `AreaUndoSet`, which contains the list of areas that were added (uncreateArea) and removed (undeleteAreas) during the modifications represented by the delta and also the list of the `AreaContentUndo` objects. The object of the class `AreaContentUndo` represents the modifications made on an area. It contains the list of the values that were added (unaddContent) and removed (unremoveContent) together with the pointer to the area the object belongs to.

Note that whenever a value is overwritten or an area becomes unreachable, the objects representing them are not deleted. They are kept in the

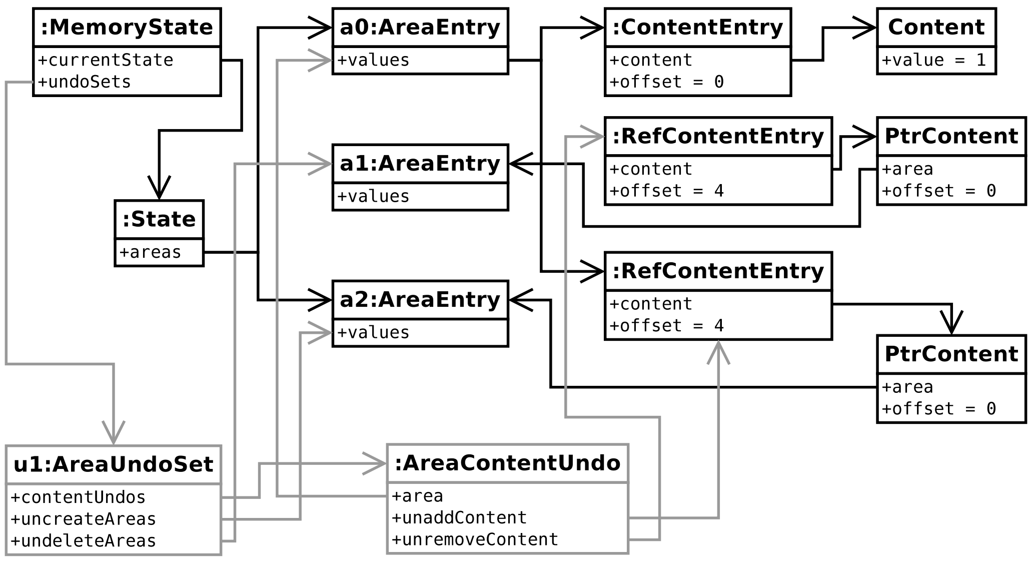

Figure 3.5: The objects after the second part of the example in Figure 3.3.

delta, so they can be later added back into the current state, when the modifications stored in the delta are reverted.

3.3.2 ContentEntry class

An object of the class `ContentEntry` holds a non-pointer value stored in the simulated memory. It contains the pointer to the GMC supplied object representing the value and the offset within the area at which the value is saved.

3.3.3 RefContentEntry class

An object of the class `RefContentEntry` holds a pointer value stored in the simulated memory. The class is derived from `ContentEntry`. It contains pointer to the area in which the value is saved.

3.3.4 AreaEntry class

An object of the class `AreaEntry` represents an area. It contains the set of the `ContentEntry` and `RefContentEntry` objects holding the values stored in the area. Next it contains the in-link list and out-link set. The in-link list is the list of the `RefContentEntry` objects representing pointer values that are pointing to the area. The out-link set is the set of the `RefContentEntry` objects representing all the pointer values stored in the area that are not `null` and point to an other area. When computing canonical addresses, only the elements of this set are explored during the BFS traversal, which speeds up the computation. The out-link set is sorted according to the offset within the area. Finally this class contains the list of modified values, whose partial hashes may need to be updated. These are the values that were added into or removed from the area, or pointer values if the canonical address to which they are pointing has changed.

The class also contains the canonical address of the beginning of the area it represents. When an area is moved during the canonicalization, no bytes are really copied, only the canonical address is updated. Also the area partial hash of the moved area is updated and the value partial hashes of the values the moved area contains and of the pointer values pointing to the moved area are updated.

3.3.5 State class

The class `State` represents the current state of the memory. It contains the list of all the areas in the state, including freed areas, but excluding unreachable areas. The class maintains the list of modified areas. An area is modified if it was allocated or freed or if any of its values is modified. The class also maintains the pointer to the root area.

3.3.6 Heap graph

In order to implement the incremental heap canonicalization algorithm, MM must be able to search the heap graph. The vertices of the heap graph are the `AreaEntry` objects, while the edges are the `RefContentEntry` objects. Because `AreaEntry` maintains the in-link list and out-link set, the incoming and outgoing edges can be found efficiently. To ease the access to the heap graph, the class `State` is a model of the *Vertex List Graph* and *Incidence Graph* concepts defined in the `boost::graph` library[17]. This allows MM to use the breadth first search algorithm from the library to calculate the canonical addresses.

3.3.7 AreaContentUndo class

This class represents the modifications made to an area. It has two lists containing `ContentEntry` and `RefContentEntry` objects. The first list contains the objects representing the values that have been added into the area since the last state was saved, while the second list contains the objects representing the values that have been removed from the area since the last state was saved.

3.3.8 AreaUndoSet class

This class contains the delta with differences between two saved states. For every modified area it contains an `AreaContentUndo` object. It also contains the list of the areas that have been allocated or freed since the last state was saved.

The constructor of this class takes the pointer to a `State` object as a parameter. It captures all the modifications made on the supplied object, so they can be undone later by calling `AreaContentUndo::undo(State*)`.

3.3.9 MemoryState

The class `MemoryState` contains the `State` object that represents the current state, the *undo stack* with the deltas (the `AreaUndoSet` objects) and the *saved states stack* with pointers to the elements of the undo stack. The undo stack contains the deltas describing differences between the saved states, while the saved states stack represents the saved states.

Methods

The `alloc()`, `free()`, `store()` and `setRootArea()` methods modify only the current state object.

The method `push()` works in the following way. First it updates the in-link lists and out-link sets of all the areas. This can be done efficiently, because the current state object tracks all the modified values. As the in-link lists and out-link sets are updated, it is also checked if the links were *significantly changed*. The links are significantly changed, when the changes in the links modify the breadth first search (BFS) tree[11]. If the links were significantly changed, the canonical addresses of all the areas are updated. The canonicalization step walks the whole area graph.

Next, the `AreaUndoSet` object is constructed, using the current state object, and it is pushed onto the undo stack. Also the pointer that is pointing to the `AreaUndoSet` object is pushed onto the saved states stack.

Next, the state hash is updated and stored in the top `AreaUndoSet` object on the undo stack. It is necessary to update only the partial hashes of the modified values and areas.

As the last step, the `ContentEntry`, `RefContentEntry` and `AreaEntry` objects that are not needed are disposed. The `AreaEntry` object is not needed if the area it represents was allocated and then became unreachable before the state was saved. The object of the class `ContentEntry` or `RefContentEntry` is not needed, if the value it represents was stored and then overwritten before the state was saved.

The `pop()` method is simple, it only removes the top element from the saved states stack. It does not remove any elements from the undo stack.

The `backtrack()` method finds the delta `u` to which the top element on the saved states stack is pointing, and then applies all the deltas that are above `u` on the undo stack to the current state, taking the top-most delta first. After that, all used deltas are disposed. This restores the current state to the state it was in when the top element of the saved states stack was pushed onto the stack.

The `hash()` method simply returns the state hash that is stored in the `AreaUndoSet` to which the top element of the saved states stack is pointing.

3.3.10 Entries life-cycle

A `ContentEntry`, `RefContentEntry` or `AreaEntry` object is created, when a value is stored into the memory or an area is allocated respectively. If the object is not needed, after the delta is created, it is disposed. If it is needed, the created delta takes the ownership of the object. Later, if the value is

[11]The BFS tree is a subtree of heap graph. It contains only the edges that, when explored during the BFS traversal, lead to an undiscovered vertex.

overwritten or the area becomes unreachable, the corresponding object is not disposed, but it is only removed from the current state. The object is disposed once the delta that owns it is disposed. The delta is disposed after it is applied to the current state object in the method `backtrack()`.

3.3.11 Containers

MM uses intrusive containers from the `boost::intrusive` library [17] everywhere, except for the stack of saved states. The intrusive containers are used to save some memory, but more importantly to be able to transform a pointer to an element into an iterator in constant time.

3.4 Space and time complexities

The space and time complexities of MM are described in this section.

3.4.1 Time complexity

The time complexities of the memory manipulation methods are stated in Table 3.5. Areas are stored in a linked list, so the addition of an area to or the removal of an area from the state can be done in constant time. Note that `free()` depends on the number of values stored in the area, because when an area is freed, all its values must be removed from the state. Since the objects holding the values are kept in sorted set (sorted by the offset within the area), whenever a value is added into or removed from the area, the time needed for this is in $O(log(n))$, where n is the number of values in the area. When storing a value, all values it overwrites must be removed from the area, hence the $O((1 + r)log(v))$ complexity in case of the method `store()`.

Time complexity of state manipulation methods is stated in Table 3.6. It is important to note, that time complexity of the method `push()` – if the links were not significantly changed – depends only on the number of modified values and areas since the last call to `push` or `backtrack()`. This is caused by the fact, that MM tracks the modifications with value granularity. Similarly the method `backtrack()` depends on the number of modified values and areas since the state the method will backtrack to has been saved.[12]

[12]Time complexity of `backtrack()` method is in fact better than is stated in the Table

Method	Time complexity	Description
`alloc()`	$O(1)$	
`alloc<T>()`	$O(1)$	
`free()`	$O(v)$	v is the number of values in the area
`free<T>()`	$O(v)$	
`setRootArea()`	$O(1)$	
`setRootArea<T>()`	$O(1)$	
`store()`	$O((1 + r)log(v))$	r is the number of values overwritten (even partially) by the inserted value
`store<T>()`	$O(log(v) + t_c)$	t_c is the time needed to create the copy of the value being stored
`load()`	$O(log(v))$	
`load<T>()`	$O(log(v))$	
`loadOverlapping()`	$O(log(v))$	

Table 3.5: Time complexity of `MemoryState` value manipulation methods.

Method	Time complexity	Description
`push()`	Average $O(t_m)$ if links were not significantly changed, average $O(t_m + a + p + m)$ otherwise	t_m is the time consumed by calls to `alloc()`, `free()` and `store()` methods (and their templated forms) between this call and previous call to `backtrack()` or `push()`, a is the number of areas in the current state (including freed, excluding unreachable), p is the number of all non-null pointer values pointing to an area other than the one they are stored in, m is the number of values in all areas, whose canonical address has changed
`pop()`	$O(1)$	
`backtrack()`	$O(t_m + t_p)$	t_m is the time consumed by calls to `alloc()`, `free()` and `store()` methods (and their templated forms) between this call and previous call to `backtrack()` or `push()`, t_p is the time consumed by calls to `push()` which created the deltas that will be applied to the current state by this call
`hash()`	$O(1)$	
`hasSavedState()`	$O(1)$	
`savedStateCount()`	$O(1)$	

Table 3.6: Time complexity of `MemoryState` methods (2).

The only method which time complexity does not depend on the number of modifications is the method `push()`. When calling this method after the links were significantly changed, whole area graph must be traversed, no matter how many values or areas were actually modified. This can be improved by implementing an incremental shortest path algorithm, such as [18].

The time complexity for the method `push()` is only for an average case, because the method updates the state hash and the canonical addresses. When the partial hashes of the modified values and areas are computed, the results of the functions R_a and R_v must be obtained (see Section 3.1.12).. Similarly when the canonical addresses are updated, the results of function C must be obtained (see Section 2.4.5). Because these functions are constructed incrementally, MM must remember the arguments that have been already used and the corresponding returned values. The functions are implemented using hash tables in whose the find and add operations take a constant time on average, so the time complexity of the method `push()` is also expressed for an average case only.

3.4.2 Space complexity

The overall space consumed by the MM depends on the space needed to represent the current state and all the deltas and on the space that is needed to represent the random functions R_a and R_v used in the computation of the value and area partial hashes (see Section 3.1.12) and the function C used in the canonicalization algorithm (see Section 2.4.5).

Since MM uses intrusive containers almost everywhere, the space needed to represent the current state and all the deltas can be easily calculated from the number and sizes of the objects that represent them. The numbers of these objects are as follows. Every pointer or non-pointer value in the current state or in any of the deltas consumes one `RefContentEntry` object or one `ContentEntry` object respectively. Every area that is reachable in the current state or in any of the saved states consumes one `AreaEntry` object. Every delta (entry on the undo stack) consumes one `AreaUndoSet` object. Finally whenever an area is modified (canonical address is modified or value is added into or removed from it) between two consecutive saved states one

3.6. Simply put, it depends only on the number of modifications stored in the deltas, that will be applied to the current state. But to express this precisely, would be quite complicated, so a higher but simpler estimation is used.

Class	Size in bytes
`ContentEntry`	48
`RefContentEntry`	76
`AreaEntry`	104
`AreaUndoSet`	32
`AreaContentUndo`	24

Table 3.7: Sizes of MM classes on 32-bit x86 architecture.

`AreaContentUndo` object is consumed. The sizes of these classes on 32-bit x86 architecture can be seen in Table 3.7.

For every value in the current state or in any of the deltas there is one user supplied object representing it. It can be seen that MM maintains only one copy of every stored value contained in current state or any of the saved states. Even if the value is contained in multiple saved states, only one copy of it (and the `ContentEntry` and `RefContentEntry` object holding it) exists. However, if two same values are stored by two different calls to the method `MemoryState::store()`, two copies of the same value exist. Very likely the memory of the running program contains many duplicities in values. Implementing a cache of values to remove the duplicities is a possible future work.

MM must represent also the random functions R_a and R_v and the function C. Because these functions are constructed incrementally, MM must remember the arguments that have been already used and the corresponding returned values. The functions are implemented using hash tables, so the space occupied to hold n argument to return value mappings is in $O(n)$. However, the number of the different arguments the functions have been called with can not be easily expressed, so the consumed space is added to the overall space complexity under a single variable.

To sum up, the space occupied by MM to represent the current state and the saved states can be expressed as:

$$\begin{aligned} T &= N_{CE} \cdot S_{CE} + N_{RCE} \cdot S_{RCE} + N_{AE} \cdot S_{AE} + N_{AUS} \cdot S_{AUS} \\ &\quad + N_{ACU} \cdot S_{ACU} + V + S_{SSS} + F + S_C \end{aligned} \tag{3.5}$$

where N_{CE}, N_{RCE}, N_{AE}, N_{AUS} and N_{ACU} are the number of objects of the classes `ContentEntry`, `RefContentEntry`, `AreaEntry`, `AreaUndoSet` and `AreaContentUndo`, the S_* are the sizes of the classes, V is the size of the memory occupied by the objects representing the values in the current state and in the saved states, S_{SSS} is the size of the saved states stack, F is the space consumed to represent the functions R_a, R_v and C and S_C is a constant.

3.5 Automated tests

MM is tested by running GIMPLE Interpreter (see Chapter 4) on various programs and checking, if it behaves correctly. For each test there are specified command line parameters to the interpreter, that are used to run the test, the input for the test and expected output. A script that runs all the tests exists to automate the testing process. The tests the script and their description can be found on the attached CD.

3.6 Benchmark

The GIMPLE Interpreter is also used in the benchmark of MM. The benchmark compares the execution times of the native run of a program and the interpreted run of the same program. The GIMPLE Interpreter is very simple. There are many places where it can be improved to run faster[13], therefore the benchmark gives only a very rough estimation of the slowdown caused by the interpretation and the slowdown of the interpreter when saving states or backtracking to previous states.

The test program is very simple and can be seen in Appendix B.2. At the beginning it reads an unsigned integer `m` and then, in a loop, it allocates an area of size `m*sizeof(unsigned int)`, reads and unsigned integer `n`, fills the area with values `n` and frees the area at the end of the loop.

Five tests were run, each with different input and interpreter parameters. Every test was executed 100 times and the final execution time was calculated as the mean of all the executions. In every test, the value of `m` was set to 1000, which means that in every loop an area for 1000 unsigned integers was allocated and filled with 1000 values. The loop was run 128 times in every interpreted test. In the native test (test 0) the loop was run 128000 times, so that the test execution time was of the same order of magnitude as the interpreted tests.

[13]E.g., it should try to store fewer values into the memory, by not storing the temporary variables.

Test	Mean	Variance
0	1.036	0.0000036
1	1.582	0.0000901
2	1.672	0.0001165
3	1.631	0.0000824
4	1.654	0.0002181

Table 3.8: Benchmark results.

The four interpreted tests differ in the way, how they save states and backtrack to previous states. Test 1 does not save states at all, while the other interpreted tests save state every time a value is read from the input (so they save them in every loop). Test 2 saves states, but never backtracks, so it behaves like a code model checker that searches a state space in a form of path. Test 3 in every loop (except the first one) backtracks to the previous saved state, so it behaves like a code model checker that searches a state space in a form of star. Finally test 4 backtracks like a code model checker exploring a state space in a form of binary tree.

The results are in Table 3.8 and Figure 3.6 shows the box-plots of the measured values. It can be seen, that the interpretation (without saving states) is 1527 times slower than a native run. On the other hand, when backtracking is enabled, execution time increases only by 5.7%. Also the differences between execution times of the variations in backtracking are small (tests 2, 3, 4); they differ in less than 2.5%.

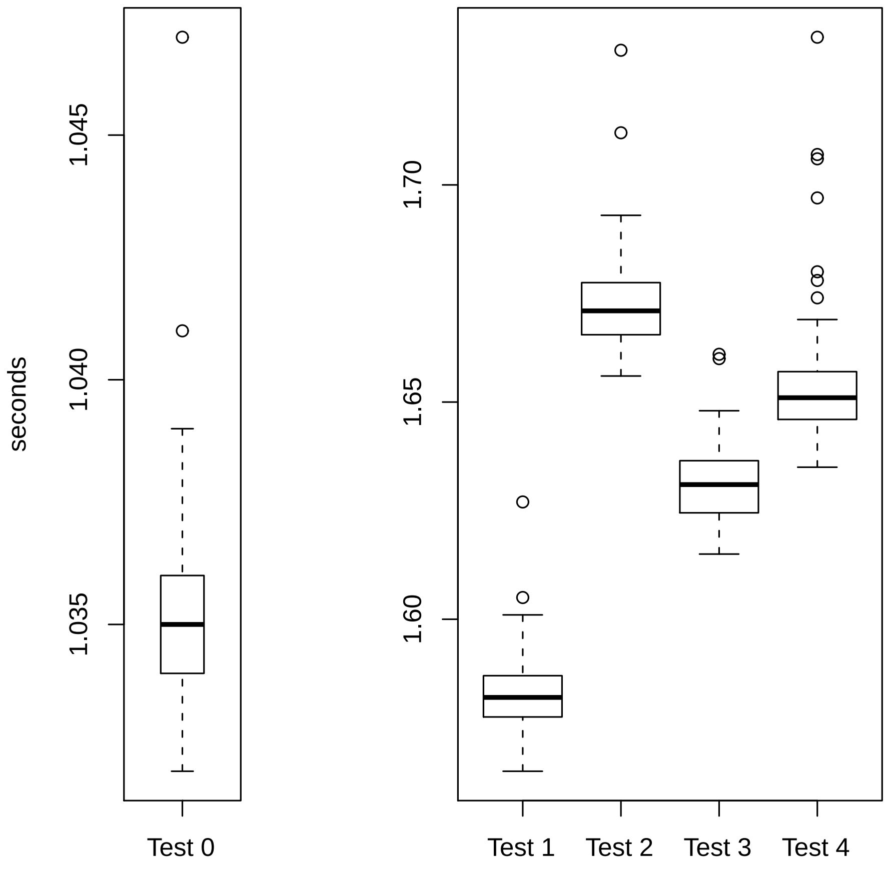

Figure 3.6: Box-plots for the measured values.

Chapter 4

GIMPLE Interpreter

To test the implementation of MM and to verify, that it provides enough functionality to fulfil all the needs of GMC, a simple interpreter of GIMPLE (*GIMPLE Interpreter*) has been created. It also serves as an example of how to use MM. Source codes of GIMPLE Interpreter and its Doxygen documentation can be found on the attached CD.

The interpreter is implemented as a patch for GCC that adds interpreting functionality to it. GIMPLE Interpreter extends GCC with five additional command line options which are described below.

`--interpret` enables interpretation. When this option is specified, GCC directly interprets the input source code instead of compiling it.

`--Ibacktrack` enables backtracking. Before every call to `scanf()`, one integer is read, that specifies how many user inputs to backtrack. For example if 1 is specified, the interpreter does not read current input, but it rather backtracks to the state, when last input was read, and re-reads this last input.

`--Ibacktrack-fine` behaves in exactly the same way as previous option. It differs only in the frequency of saving the states. When this option is specified, the state is saved also before every `gimple_cond` statement.

`--Istate-reentry-check` enables state re-entry checking. Before every call to `scanf()`, interpreter acquires the hash of the state and checks if a state with the same has been visited before. If so, the interpreter reports this fact and exits.

Statement	Description
`gimple_assign`	Assignment operation.
`gimple_cond`	Conditional jump.
`gimple_call`	Function call.
`gimple_return`	Return from function.

Table 4.1: Supported GIMPLE statements.

`--Istate-reentry-check-fine` behaves like the previous option, but it checks for the state re-entry also before every call to `gimple_cond`.

To see the GIMPLE representation of the program that will be interpreted, the option `--dump-tree-interpret-raw` can be used together with `--interpret` option. This prints the GIMPLE representation of the program to the file named like the input source file but with the `.127t.interpret` suffix added. For example the command

```
gcc --interpret --dump-tree-interpret-raw fib.c
```

will interpret `fib.c` file, but it will also print the GIMPLE representation of the program into the file named `fib.c.127t.interpret`. The source file `fib.c` that prints out Fibonacci numbers and its GIMPLE representation can be found in Appendix B.1. If only the option `--dump-tree-interpret` is specified (without `-raw` suffix), the GIMPLE representation of the program is printed using a C like syntax. However, to see the GIMPLE statements used in the program, the option with `-raw` suffix must be used.

Supported program constructs

As said before, GIMPLE Interpreter is quite simple, so not all programs can be interpreted. In the following text is described, what program constructs are supported.

The set of data types supported by the interpreter is very limited. Supported are only all sorts of signed and unsigned integers, arrays of supported types and pointers to supported types.

The interpreter also supports only four GIMPLE statements listed in Table 4.1. However, these statements are sufficient to interpret simple C programs.

The interpreter assumes that the input contains the program entry point (the function `main()` in C and C++ programs) and the definitions of all the used functions. There are however a few standard C functions that can be used: `malloc()`, `free()`, `printf()`, `puts()` and `scanf()`. The behavior of the functions `malloc()`, `free` and `puts()` is the same as it is defined by the C standard, but the behavior of the other two functions differs. The function `printf()` completely ignores the type specification of the parameters in the format string (the first parameter of `printf()`). It prints the parameters at the right positions, but the format of their output depends only on their real type and not on the type and format specified in the format string. The function `scanf()` ignores the format string completely. It reads $n - 1$ integer values, where n is the number of the parameters the function was called with, and stores them in the locations to whose the parameters that follow the first parameter are pointing. If any of the second and later parameters is not a pointer to an integer value, the interpreter aborts when the `scanf()` call is interpreted. Whenever the interpreter detects an unsupported program construct, an error message is printed on the error output and the interpreter is aborted.

Output

GIMPLE Interpreter writes on its standard output the standard output of the interpreted program and also few error and information messages. The messages are enclosed between `<` and `>` characters. The list of the message strings and their descriptions can be found in Table 4.2. The other error and information messages produced by the interpreter are printed on the error output, since the interpreting program can't write to error output.

4.1 Implementation

GIMPLE Interpreter does not use the GIMPLE representation used in GCC directly, rather it uses GIMPLE++ representation created by the *GIMPLE Iterator* module (see Chapter 5). GIMPLE++ is a read-only representation of the program that has C++ interface and contains the information extracted from GIMPLE that is relevant to GIMPLE Interpreter.

`Printed string`	Description	Aborts
`<unreachable area detected>`	When an non-freed unreachable area is detected for the first time.	No
`<state re-entered>`	When a state was re-entered.	Yes
`<freed area access>`	`FreedAreaAccessException` has been thrown.	Yes
`<null pointer>`	`NullPointerException` has been thrown.	Yes
`<undefined pointer operation>`	`UndefinedPtrOperationException` has been thrown.	Yes
`<not area begin>`	`NotAreaBeginException` has been thrown.	Yes
`<undefined memory load>`	`UndefinedMemoryLoadException` has been thrown.	Yes
`<out of bounds>`	`OutOfBoundsException` has been thrown.	Yes

Table 4.2: Message strings.

4.1.1 Values

The class `Value` is the common predecessor of all classes representing values stored in the simulated memory. The sub-classes `BoolValue`, `IntegerValue`, `StringValue` and `PtrValue` exist to represent the values of the `bool`, integer, string and pointer types.

4.1.2 Memory structures emulation

GIMPLE Interpreter emulates program memory structures on the heap in the following way. The root is a `ProgramInfo` area (an area that was allocated by a call to the templated form of `MemoryState::alloc()` method with `ProgramInfo` as the template parameter) that contains pointer to the read-only area with the constants and functions used in the simulated program and to a `ThreadInfo` area for the main and only thread of the program. The `ThreadInfo` area contains the current instruction pointer and the pointer to the bottom stack frame. The stack frame is represented by a `FrameInfo` area, that contains the references to the previous and next stack frames, the return address and pointer to the area with function local variables and parameters. The areas that can be accessed by the simulated program, are manipulated using non-templated form of `MemoryState` methods, while the templated methods are used to manipulate the areas backed by `*Info` classes.

4.1.3 Interpreting

The interpreter sequentially interprets the GIMPLE statements. First, it finds the program entry point (main function), takes the first block and starts interpreting the statements in the block using the object of the class `GimpleStmtInterpreter`. The interpreter stops after the program exits the main function or if an error occurs.

Chapter 5

GIMPLE Iterator module

This module transforms GIMPLE, the internal representation of the program used in GCC, into a representation that is easier to work with from within C++ (GIMPLE++). GIMPLE++ is a read only representation of the program. It is meant to be used only for the queries, all the optimizations and transformations must be done in GCC or during the initialization. Once the GIMPLE++ representation is constructed, it can not be modified.

5.1 Gimple class

The class `Gimple` is the main class of the module, that holds the GIMPLE++ representation of program source code. To initialize a newly constructed object of this class, the member function `processCfun()` must be run for each function being compiled. This call transforms the currently compiled function into the GIMPLE++ representation and saves it into the `Gimple` object.

The constructor of the class `Gimple` takes one parameter, a listener used to report encountered declarations, constants and function definitions. During the execution of `processCfun()` method, whenever a declaration, constant or function definition is encountered the method `onDecl()`, `onCst()` or `onFuncDef()` is called on the listener. The listener can be used to initialize the read only memory of the simulated program with constants and functions.

After the initialization, the method `entryPoint()` can be used to get the declaration of the entry point function.

Function	Description
`getFunctionType()`	Returns the type of the function.
`getParamDeclarations()`	Returns the declarations of the function parameters.
`getLocalDeclarations()`	Returns function local declarations (local variables, local functions, ...).
`getFirstBlock()`	Returns the first block of the function.

Table 5.1: Methods of the class `Function`.

5.2 Function class

The class `Function` represents a function and it is the place where the function body is stored. The most important methods are described in Table 5.1. The body of the function is composed of blocks of statements.

5.3 Block class

The class `Block` represents an uninterruptible block of statements. When any of the statements in a block is executed, they are all executed. This means that the `GIMPLE_COND` or `GIMPLE_RETURN` statements are always the last statement in the block. Every block contains pointers (edges) to all the blocks that can follow it, to know where to continue with the execution. For the list of most important methods see Table 5.2.

5.4 Other classes

Every class representing a statement, operation, operand, type, declaration or constant is a sub-type of the class `Stmt`, `Operation`, `Operand`, `Type`, `Declaration` or `Constant` respectively. The visitor pattern can be used to work with the sub-types. When the user holding a pointer or a reference to the super-class wants to perform an action on the sub-class, he can call `accept(Visitor&)` method on the super-class and pass an object of a class implementing the interface `Visitor` defined in the super-class. The method

`Function`	Description
`stmts()`	Returns the range of all the statements in the block.
`succEdges()`	Returns the range of all the out edges.
`predEdges()`	Returns the range of all the in edges.
`getTrueBranchEdge()`	Returns the edge that is taken, when the last `GIMPLE_COND` statement evaluates to `true`.
`getFalseBranchEdge()`	Returns the edge that is taken, when the last `GIMPLE_COND` statement evaluates to `false`.
`getFallthruEdge()`	Returns the edge that is taken, when the last statement is neither a conditional jump nor a return statement.

Table 5.2: Methods of `Block` class.

`accept(Visitor&)` will call the method `visit(SubType&)` on the visitor, where `SubType` will be the actual type of the object on which `accept()` was called.

Chapter 6

Conclusion

The newly implemented memory module (MM) for GIMPLE Model Checker (GMC) was presented in this thesis. MM is used to represent the memory of a model checked program. It has the interface that allows low-level memory access. It can backtrack to previously saved memory states and compute the hash of the current memory state in order to allow exploration of the program state space.

MM differs from other code model checkers in the fact, that it allows to store any object implementing the given interface. It is possible to store, e.g., the data of a value together with its type, a symbolic value used in a symbolic execution or a predicate over a stored value used in predicate abstraction, not only the byte representation of a value.

MM uses various techniques to lower the time and memory consumption. It tracks all modifications to the memory state with value granularity, which allows to store only the deltas capturing the differences between saved states and efficiently compute the hash of a memory state. Unlike Java PathFinder or MoonWalker, MM does not use collapse compression, therefore it does not hold in the memory the values or areas that do not reside in the current state or any of the saved states. This means, that the memory consumption of MM is proportional to the number of values and areas in the current state and in all the deltas.

MM uses incremental hashing to compute the hash of a memory state. Before every state hash computation, the incremental heap canonicalization algorithm is used which reorganizes the areas in the memory to form a canonicalized heap, so the returned state hash is the same for all the equivalent heaps.

MM detects attempts to do an invalid memory operation, like access a value in non-allocated memory, read a value from uninitialized memory or double free an area. It also allows to detect area placement dependent operations like moving pointers between areas, computing the difference between pointers belonging to different areas and comparing pointers belonging to different areas.

To test MM and to verify, that it provides enough functionality to fulfil all the needs of GMC, a simple interpreter of GIMPLE (*GIMPLE Interpreter*) was created. As a part of GIMPLE Interpreter the GIMPLE Iterator module was created, to ease the access to the GIMPLE representation of a program from within C++.

6.1 Future work

There are few possibilities of how to further lower the memory consumption of MM and how to increase its speed. Currently, when the heap is canonicalized, the whole heap graph is examined by the breadth first search algorithm. Before the canonicalization step, it is checked, whether the canonical addresses really need to be updated (the links were significantly changed). This check takes the time proportional only to the number of the modifications made since the last saved state. In many cases the check succeeds and allows to skip the canonicalization step, but if the check fails, the whole heap graph still needs to be examined. The slowest operation in the canonicalization algorithm is finding the areas, whose shortest path to the root area has changed. Currently MM uses the breadth first search algorithm to find these areas, but a more effective *incremental shortest path*[18] algorithm can be used.

Whenever a value is stored into the simulated memory, MM saves the object that represents the value, even if there already is the same object present. It is very likely, that there will be many duplicated values stored in the memory, so a *value caching* can be implemented to lower the memory consumption.

Multiprocessor and multi-core systems are common at the present time, therefore MM should be extended to support multi-threading or even be distributable across multiple hosts.

Bibliography

[1] Havelund K., Pressburger T. (2000): Model Checking Java Programs Using Java Pathfinder. *International Journal on Software Tools for Technology Transfer 2* (4), pp. 366–381

[2] Demartini C., Iosif R., Sisto R. (1999, June): A Deadlock Detection Tool for Concurrent Java Programs. *Software: Practice and Experience 29* (7), pp. 577–603.

[3] Holzmann G. J. (2000): Logic Verification of ANSI-C code with SPIN. *In Proc. of the 7th International SPIN Workshop*, Volume 1885. Springer-Verlag.

[4] Lerda F., Visser W. (2001): Addressing Dynamic Issues of Program Model Checking. *Model Checking Software*, pp. 80-102.

[5] http://gcc.gnu.org/

[6] http://gcc.gnu.org/onlinedocs/gccint/

[7] Musuvathi M., Park D. Y. W., Chou A., Engler D. R., Dill D. L. (2002): CMC: A Pragmatic Approach to Model Checking Real Code. *Proceedings of the Fifth Symposium on Operating System Design and Implementation*

[8] Holzmann G. J. (1997): The Model Checker SPIN. *IEEE Transactions on Software Engineering 23*, pp. 279–295.

[9] Demartini C., Iosif R., Sisto R. (1999): dSPIN: A Dynamic Extension of SPIN. In *SPIN*, pp. pp. 261–276.

[10] Iosif R. (2001): Exploiting Heap Symmetries in Explicit-State Model Checking of Software. *In Proc 16th IEEE Conference on Automated Software Engineering*, pp. 254–261

[11] Andrews T.,Qadeer S.,Rajamani S. K., Rehof J., Xie Y. (2004): Zing: Exploiting Program Structure for Model Checking Concurrent Software. *CONCUR 2004*

[12] Andrews T., Qadeer S., Rajamani S. K., Rehof J., Xie Y. (2004): Zing: A Model Checker for Concurrent Software. *MSR Technical Report: MSR-TR-2004-10.*

[13] Ruys T. C., Aan de Brugh N. H. (2007): MMC: the Mono Model Checker. *Electron. Notes Theor. Comput. Sci.* 190, 1 (Jul. 2007), pp. 149–160.

[14] Aan de Brugh N. H., Nguyen V. Y., Ruys T. C. (2009): MoonWalker: Verification of .NET Programs. *Proceedings of the 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*: pp. 170–173.

[15] Musuvathi M., Dill D. L. (2005): An Incremental Heap Canonicalization Algorithm. *SPIN 05: SPIN Workshop*, Springer Verlag

[16] ISO (1999): The ANSI C Standard (C99). *Technical Report WG14 N1256*, ISO/IEC.

[17] http://www.boost.org/doc/

[18] G. Ramalingam, T. W. Reps (1996): An incremental algorithm for a generalization of the shortest-path problem. *J. Algorithms 21*, pp. 267–305

[19] Holzmann G. J. (1997): State compression in spin: Recursive indexing and compression training runs. *In Proc. of the 3th International SPIN Workshop*

Appendix A

Examples

A.1 State manipulation

A.1.1 Non-templated methods

This example demonstrates the usage of the non-templated memory manipulation methods of the class `MemoryState`.

```
using namespace mmodule;

class IntValue : public NonPtrContent {
  private:
    // The hash of this type
    static const size_t typeHash = 0xc6685ae6;

  public:
    IntValue (int iv) : i (iv)  {}
    size_t getUnitSize () const { return 4; }
    Hash getHash () const
    { return ((size_t)i)+typeHash; }

    int i;
};

static void example1 (MemoryState* ms)
{
    Ptr area1 = ms->alloc (2*4);

    std::auto_ptr<Content> contAP (new IntValue (11));
    ms->store (area1, contAP);
    contAP.reset (new IntValue (12));
    ms->store (area1+4, contAP);

    const Content* cont1 = ms->load (area1);
    assert (typeid (*cont1) == typeid (IntValue));
    assert (dynamic_cast<const IntValue*> (cont1)->i == 11);
    const Content* cont2 = ms->load (area1+4);
    assert (typeid (*cont2) == typeid (IntValue));
    assert (dynamic_cast<const IntValue*> (cont2)->i == 12);

    std::pair<MemoryState::ContentIterator,
              MemoryState::ContentIterator> range;
    // get range of all values in the area
    range = ms->loadOverlapping (area1, ms->getSize (area1));
    assert (range.first->first == 0);
    assert (range.first->second == cont1);
    range.first++;
    assert (range.first->first == 4);
    assert (range.first->second == cont2);
    range.first++;
    assert (range.first == range.second);

    contAP.reset (new IntValue (13));
    const Content* cont3 = contAP.get ();
    ms->store (area1 + 2, contAP);  // overwrite values
    range = ms->loadOverlapping (area1, ms->getSize (area1));
    assert (range.first->first == 2);
    assert (range.first->second == cont3);
    range.first++;
    assert (range.first == range.second);

    const Content* cont4 = ms->load (area1+2);
    assert (typeid (*cont4) == typeid (IntValue));
    assert (dynamic_cast<const IntValue*> (cont4)->i == 13);

    ms->free (area1);
}
```

A.1.2 Templated methods

This example demonstrates the usage of the templated memory manipulation methods of the class `MemoryState`.

```
struct SizeWrapper {
    size_t s;
};
```

```
size_t hash_value (const SizeWrapper& sw)
{ return sw.s; }

struct Struct {
    int i;
    SizeWrapper sw;
};

static void example2 (MemoryState* ms)
{
    SizeWrapper sizeWrapper = { 11 };

    Ref<Struct> ref = ms->alloc<Struct> ();

    ms->store (ref, &Struct::i, 1);
    ms->store (ref, &Struct::sw, sizeWrapper);
    assert (ms->load (ref, &Struct::i) == 1);
    assert (ms->load (ref, &Struct::sw).s == 11);

    sizeWrapper.s = 12;
    ms->store (ref, &Struct::sw, sizeWrapper);
    ms->store (ref, &Struct::i, 2);
    assert (ms->load (ref, &Struct::i) == 2);
    assert (ms->load (ref, &Struct::sw).s == 12);

    ms->free (ref);
}
```

A.2 Saved states management

This example demonstrates the usage of state manipulation methods of the class `MemoryState`.

```
struct Struct2 {
    int i1;
    int i2;
};

static void example3 (MemoryAccessListener* listener)
{
    MemoryState ms (listener);
    Ref<Struct2> ref = ms.alloc<Struct2> ();
    ms.setRootArea (ref);
    ms.store (ref, &Struct2::i1, 1);
    ms.store (ref, &Struct2::i2, 2);
    ms.push ();
    Hash h1 = ms.hash ();

    ms.store (ref, &Struct2::i1, 11);
    ms.push ();
    Hash h2 = ms.hash ();
    assert (h1 != h2);

    ms.store (ref, &Struct2::i2, 12);
    ms.push ();
    Hash h3 = ms.hash ();
    assert (h1 != h2 && h2 != h3);
    assert (ms.savedStateCount () == 3);

    ms.pop ();
    ms.pop ();
    ms.backtrack ();
    assert (ms.load (ref, &Struct2::i1) == 1);
    assert (ms.load (ref, &Struct2::i2) == 2);
    assert (ms.savedStateCount () == 1);

    ms.store (ref, &Struct2::i1, 11);
    ms.push ();
    Hash h4 = ms.hash ();
    assert (h4 == h2);
}
```

Appendix B

Program source codes

B.1 Fibonacci numbers

This is an example of C program and its GIMPLE representation. The file containing the GIMPLE representation is created by the GCC when it is run with `--interpret` and `--dump-tree-interpret-raw` options.

B.1.1 `fib.c`

```
#include <stdio.h>

int main()
{
  int n = 0;
  int f_2 = 0, f_1 = 0, f;
  int i;

  printf ("Enter how many numbers of Fibonacci sequence to show: ");
  scanf ("%d", &n);

  for (i = 0; i <= n; i++)
    {
      if (i <= 1)
        f = 1;
      else
        f = f_1 + f_2;

      f_2 = f_1;
      f_1 = f;
      printf ("fib (%d) = %d\n", i, f);
    }

  return 0;
}
```

B.1.2 fib.c.127t.interpret

```
;; Function main (main)

main ()
{
  int i;
  int f;
  int f_1;
  int f_2;
  int n;
  int D.1635;
  int n.0;

<bb 2>:
  gimple_assign <integer_cst, n, 0, NULL>
  gimple_assign <integer_cst, f_2, 0, NULL>
  gimple_assign <integer_cst, f_1, 0, NULL>
  gimple_call <printf, NULL, \\
&"Enter how many numbers of Fibonacci sequence to show: "[0]>
  gimple_call <scanf, NULL, &"%d"[0], &n>
  gimple_assign <integer_cst, i, 0, NULL>
  goto <bb 7>;

<bb 3>:
  gimple_cond <le_expr, i, 1, NULL, NULL>
    goto <bb 4>;
  else
    goto <bb 5>;

<bb 4>:
  gimple_assign <integer_cst, f, 1, NULL>
  goto <bb 6>;

<bb 5>:
  gimple_assign <plus_expr, f, f_1, f_2>

<bb 6>:
  gimple_assign <var_decl, f_2, f_1, NULL>
  gimple_assign <var_decl, f_1, f, NULL>
  gimple_call <printf, NULL, &"fib (%d) = %d\n"[0], i, f>
  gimple_assign <plus_expr, i, i, 1>

<bb 7>:
  gimple_assign <var_decl, n.0, n, NULL>
  gimple_cond <le_expr, i, n.0, NULL, NULL>
    goto <bb 3>;
  else
    goto <bb 8>;

<bb 8>:
  gimple_assign <integer_cst, D.1635, 0, NULL>
  gimple_return <D.1635>

}
```

B.2 Benchmark test program

This is the program used in the benchmark of MM.

```
#include <stdio.h>
#include <stdlib.h>

int
main ()
{
  unsigned n, i, m;
  unsigned* a;

  scanf ("%d", &m);
  n = 1;
  while (n > 0)
  {
    a = (unsigned*) malloc (m*sizeof(unsigned));
    scanf ("%d", &n);

    for (i = 0; i < m; i++)
      a[i] = n;

    printf ("%d\n", a[0]);
    free (a);
  }
  return 0;
}
```

Appendix C

The attached CD

The sources and the documentation of the implemented modules can be found on the attached CD. The most important files and directories are stated in the following list, but a more detailed description can be found in the `README` files on the CD.

`ginterpreter/`	Contains the sources and the Doxygen documentation for MM, GIMPLE Interpreter and GIMPLE Iterator.
`benchmark/`	Contains the inputs and the scripts that were used to run the benchmark of MM and the results of the benchmark.
`tests/`	Contains the inputs and the scripts that were used to test MM.
`scripts/`	Contains various helper scripts.
`Jan_Kouba-thesis.pdf`	Text of this thesis.