Charles University in Prague
Faculty of Mathematics and Physics

MASTER'S THESIS

Ondrej Krč-Jediný

GIMPLE Model Checker

Department of Distributed and Dependable Systems

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

2010

I would like to thank to my supervisor Ondřej Šerý for his valuable advice during the whole process of creating the work. I would also like to thank to my father Ján and friend Andrej for reviewing the text.

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, 8.12.2010 Ondrej Krč-Jediný

Contents

Název práce: GIMPLE Model Checker
Autor: Ondrej Krč-Jediný
Katedra (ústav): Katedra distribuovaných a spolehlivých systémů
Vedoucí diplomové práce: RNDr. Ondřej Šerý Ph.D.
e-mail vedoucího: Ondrej.Sery@mff.cuni.cz

Abstrakt: Cieľom práce je implementácia základných prvkov explicit-state model checkeru pre jazyk C - pokročilého nástroja na hľadanie chýb v programoch. Tento nástroj prehľadáva všetky možné cesty, ktorými môže byť program vykonávaný a zároveň vyskúša všetky možné kombinácie prekladania vlákien. Nástroj je založený na GIMPLE - výstupe front-endu kompilátora GCC, ktorý berie za svoj vstupný jazyk. Práca využíva predchádzajúcu prácu 'Memory representation for GIMPLE Model Checker', ktorá implementuje prácu s pamäťou pre tento nástroj. Tým, že je nástroj vychádza z GIMPLE, umožňuje overovanie systémov priamo v jazyku C, naviac je ľahko rozšíriteľný na iné jazyky podporované GCC.

Kľúčové slová: model checking, GIMPLE, GCC, C

Title: GIMPLE Model Checker
Author: Ondrej Krč-Jediný
Department: Department of Distributed and Dependable Systems
Supervisor: RNDr. Ondřej Šerý Ph.D.
Supervisor's e-mail address: Ondrej.Sery@mff.cuni.cz

The goal of the thesis is a prototype implementation of explicit-state model checker of C - an advanced tool for finding errors in programs. This tool explores all possible paths of program execution as well as all thread interleavings. It is based on GIMPLE - output of front-end of GCC compiler, which is the input language for GMC. The thesis is based on the previous work 'Memory representation for GIMPLE Model Checker', that implements work with memory for this tool. Since it is based on GIMPLE, it makes it possible to verify systems directly in C. In addition, it is easily extensible to other languages supported by GCC.

Keywords: model checking, GIMPLE, GCC, C

Chapter 1

Introduction

Over the time, software complexity has grown to the point that many programs can be hardly understood by a single person. That implies an increased amount of unforeseen errors because the programmer simply isn't able to grasp the whole program in his mind. Software testing is there to help finding such errors, but it is not perfect. Experience shows that many errors pass the process of testing unnoticed. Moreover, there are some errors that cannot be reliably detected by normal testing. These are for example deadlocks and data race problems for programs with multiple processes or threads.

Another fact is that software is nowadays used in many fields where a failure may have catastrophic consequences - loss of a big amount of money or even a human life. There was a need for an improvement of the process of software testing to try to minimize the occurrence of such events. This is where model checking comes to help.

Model checking is a process that given a model of a system, it automatically verifies it against a specification. It decides whether the system satisfies the specification or not. If not, it provides a counter-example or it informs the user how the specification was violated. A more specific type of model checking is code model checking. Rather than verifying the model of a program, it verifies the code of the program itself. Model checking has been successfully applied to verify hardware designs and communication protocols.

This work presents a prototype implementation of a GIMPLE Model Checker (GMC), which is a code model checker for C language. GIMPLE is the output of the front end of GCC[1] [1], which is an input language of GMC. GIMPLE is language independent, so GMC will be suitable for use for any language that is

[1] GCC stands for GNU Compiler Collection

compilable with GCC, however it focuses on C at the moment. These languages are currently C, C++, Objective-C, Fortran, Java and Ada. Moreover, GIMPLE is much simpler to interpret than C itself. GMC is a tool that finds various errors in programs, including deadlocks and data race problems.

This work is based on a thesis that implemented the memory representation for the model checker - the Memory Module [2].

In Chapter 2, overview of GCC and GIMPLE is presented. Chapter 3 describes model checking in more detail and explains important terms. Chapter 4 presents a brief description of the Memory Module work. Chapter 5, the key Chapter of this work, presents the implementation of GMC. Finally Chapter 6 contains a description of how to work with GMC.

Chapter 2

GCC and GIMPLE

This chapter contains the necessary description of GCC and its intermediate language GIMPLE, which is an input language for GMC.

2.1 GCC overview

GCC is a system of compilers supporting various programming languages which are C, C++, Objective-C, Fortran, Java and Ada. As a part of the GNU Project[1], its sources are fully available to anyone. That allowed to base GMC on GIMPLE, which is an intermediate representation of the compiled program that emerges during the compilation process.

2.1.1 Front end vs. back end

As is usual for compilers, GCC can be divided into two separate parts, **front end** and **back end**. A connection between these two parts is an intermediate language, which is an output of the front end but input of the back end. The reason for such splitting is as follows.

Front ends of GCC exist for various languages that are mentioned in the beginning of this section. All of these languages can be converted by the front end to the intermediate language.

On the other side, **back end**s of GCC exist for various architectures. The intermediate language can be converted by a concrete back end of the compiler to the target code for the architecture.

[1] GNU Project is a free collaboration project that allows access to all products and their sources created using GNU.

Rather than having a compiler for each language on each architecture, splitting the compiler like this produces M*N compilers for M languages (front ends for them) and N architectures (back ends for them).

An intermediate representation of GCC is called GIMPLE, which is therefore source language and target architecture independent. [3]

2.1.2 LTO

A new feature of GCC that appeared in version 4.5.0 is called **link time optimization** (**LTO**) [4]. It allows GCC to make inter procedural optimizations. All of the compilation units that make an executable can be included into the optimization process.

Important for GMC is the fact that when using LTO, GCC dumps GIMPLE representation of the program to the disk. After compiling (and dumping) all of the units, LTO reads the GIMPLE representation of the whole program and starts the optimization process. GMC takes advantage of this and reads the GIMPLE representation at the same point as the Link Time Optimizer, obtaining the whole GIMPLE representation of the program from one place. Therefore GMC does not have to worry about programs split to more source files.

2.2 GIMPLE overview

As stated before, GIMPLE is a source language and target architecture independent representation of a program used inside GCC. Statements in GIMPLE are derived from GENERIC[2] by breaking down its expressions into tuples of at most 3 operands (with some exceptions, for example function calls). To achieve this, GIMPLE uses temporary variables.

GIMPLE has a quite limited instruction set, which is one of the reasons why it was chosen as an input language for GMC.

[2] GENERIC is a form of an intermediate representation that is a predecessor of GIMPLE. The process of turning GENERIC into GIMPLE is called *gimplification*.

GIMPLE instruction set		
Function	Brief Description	Supported
GIMPLE_ASM	inline assembly statements	No
GIMPLE_ASSIGN	assignment statement	Yes
GIMPLE_CALL	function calls	Yes
GIMPLE_COND	conditional jump	Yes
GIMPLE_DEBUG	debug statement	No
GIMPLE_EH_FILTER	exception specification	No
GIMPLE_GOTO	unconditional jumps	Yes
GIMPLE_LABEL	label statements (jump targets)	Yes
GIMPLE_NOP	'do nothing' statement	Yes
GIMPLE_PHI	PHI nodes	Yes
GIMPLE_RESX	resumes execution after an exception	No
GIMPLE_RETURN	return from functions	Yes
GIMPLE_SWITCH	the multiway branch (switch statement)	Yes

Table 2.1: GIMPLE instruction set

2.2.1 GIMPLE instruction set

Table 2.1 shows the GIMPLE instruction set[3]. A more detailed description of these instructions can be found in the Doxygen documentation of the classes that represent them in GIMPLE Iterator on the attached CD.

There are also instructions starting with prefix GIMPLE_OMP_, which serve to handle OpenMP[4] threads. These aren't supported, since GMC supports POSIX threads.

Reasons for not supporting some statements are:

- GIMPLE_ASM - inline assembly statements are assembly routines written as inline functions. They are useful in system programming, because of their high speed. Inline assembly routines can handle low-level instructions like manipulating registers, which cannot be implemented in GMC, since it doesn't emulate registers. It is out of the scope of this work.

[3] GIMPLE has two forms, 'high GIMPLE' and 'low GIMPLE'. High GIMPLE is not yet fully lowered, while low GIMPLE already is. This is an instruction set of low GIMPLE, which is the form GMC reads.

[4] OpenMP is an implementation of multithreading, a method of parallelization. To achieve efficient parallelization, it uses declarative directives [5].

- `GIMPLE_DEBUG` - debug statements are used for debugging purposes inside GCC, and are not 'real' instructions.

- `GIMPLE_EH_FILTER`, `GIMPLE_RESX` - instructions handling C++ exceptions are not supported, since GMC focuses on C.

Chapter 3

Model Checking

This chapter contains a more detailed description of model checking, explains important terms and problems that model checking faces and describes existing model checking tools. Solutions chosen for GMC are discussed at the end of this chapter.

3.1 Introduction

3.1.1 Traditional model checking

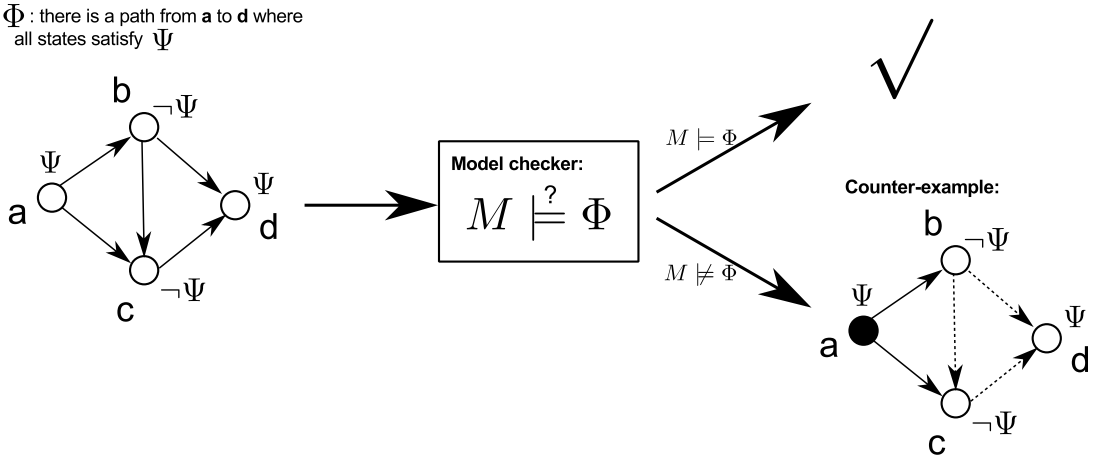
Traditional model checking is an automated technique for verifying formal systems. *Verification* is a process that checks whether particular properties (specification) hold for the system in question. Formally speaking, given a structure M and a logical formula Φ, the model checker decides whether M is a model of Φ, i.e. whether M satisfies Φ ($M \models \Phi$). M is an abstract model of the verified system, usually a *Kripke structure* or a *labeled transition system* (LTS). These are graph structures where nodes of the graph represent states of the system and edges represent transitions between the states. [6]

A **Kripke structure** over a set AP of *atomic propositions* is a 3-tuple (S, R, I) where S is a set of *states*, R is a *transition relation* $R \subseteq S \times S$ and $I : S \to 2^{AP}$ is an *interpretation*. Applied to model checking, S is the finite set of the possible states of the system. R represents all possible transitions between the states of the system and I defines what local properties (atomic propositions) hold for particular states.

A Kripke structure is *total*, if R is a total relation, i.e. $\forall s \in S\ \exists s' \in S$ such that $(s, s') \in R$. In total Kripke structures there is always a path through the structure that is infinite[1]. Kripke structures that are not total are called *partial.* [6]

A **labeled transition system** is a 3-tuple $(S, Act, \rightarrow)$, where S is a set of *states*, A is a set of *actions* and $\rightarrow$ is a set of *transitions* $\rightarrow \subseteq S \times Act \times S$. Therefore a transition $(s, a, s') \in \rightarrow$ defines that the system can evolve from state s to state s' with an action a. [6]

Properties of the system that is being checked are represented by formulas of *temporal logic*, which is a logic that includes a notion of time in its formulas. There are many variants of temporal logic, notably *linear temporal logic* (LTL) and *computational tree logic* (CTL) that are used in model checking. Logics vary in expressive power of the formulas and complexity of model checking. The heart of such model checker is the *decision algorithm*, that decides whether the system satisfies the formula or not. If not, it provides a counter-example showing how were the properties represented by the formula violated. Figure 3.1 illustrates the described process.



Φ : there is a path from **a** to **d** where all states satisfy Ψ
a
b
c
d
Ψ
$\neg\Psi$
Model checker:
$M \overset{?}{\models} \Phi$
$M \models \Phi$
$M \not\models \Phi$
√
Counter-example:

Figure 3.1: Model checking

3.1.2 Code model checking

In **software model checking** the systems being verified are programs. An approach that exhaustively enumerates all the states reachable by the system is

[1] In model checking, usage of total Kripke structures ensures that a deadlock state has a single edge that leads back to itself.

called **explicit-state model checking**.

Model checkers, that accept as input the code of the program rather than a model of the program are **code model checkers**. These eliminate the necessity of creating a model of the program, but there are also some disadvantages to this approach that will be discussed further.

A code model checker usually does not verify program properties defined by formulas of temporal logic. Properties are hard-coded in the implementation of the code model checker and all programs get checked for the same properties. These are various errors that may occur in programs such as deadlocks and data race conditions. [7]

According to the classification presented in this chapter, GIMPLE Model Checker presented in this work is an *explicit-state code model checker* of the C language.

3.2 Advantages and disadvantages

Model checking has several advantages over traditional testing. It is an automated process, therefore there is no extra work that has to be done to begin it (in comparison to writing tests). It explores all possible states of the system, which is hardly achievable by traditional testing. Moreover, it can detect important errors that cannot be detected by traditional testing, like the already mentioned data race problems and deadlocks.

However, there are some problems that model checkers face. Since model checking is in general an undecidable[2] problem, the model checker may fail to finish and run out of time or memory. These problems may also occur for big programs, since the number of states rapidly exceeds the computational limits for complex programs. This problem is named *state explosion.*

Exploring all possible states of the program includes exploring all possible interleavings of threads that may occur during program execution. For a program with threads $N_1, N_2, ...N_n$ where the i-th thread has n_i atomic instructions, the number of possible scheduling sequences of the threads can be computed by the formula [9]:

$$M = \frac{(\sum_{i=1}^{N} n_i)!}{\prod_{i=1}^{N} (n_i)!}$$

[2] Undecidable problem is a decision problem for which there is no algorithm that always leads to a yes or no answer.

For example a program running in 3 threads with 5 atomic instructions in each has 756756 different scheduling sequences. Model checkers thus have to provide techniques for avoiding the state explosion problem.

First of these techniques is called **state matching**. While a model checker explores a program, it maintains information about already visited states and if such state is entered for the second time, the model checker recognizes it and does not continue exploration from this state.

Another technique to reduce the state space is **partial order reduction** (**POR**). It recognizes *independent* instructions which are instructions that result in the same state of the system when executed in any order [10]. The model checker then does not have to try all the possible interleavings of these instructions.

Backtracking refers to a capability of model checkers to restore previous execution states and continue exploration following some yet unexplored branch. Otherwise it would have to execute all the instructions from the beginning, which is very inefficient. This requires the maximum effectiveness of the representation of a state of a program and the state storage.

To minimize the memory used to represent the already explored states, model checkers use state hashing. During the exploration of the program states only the hashes of the states are computed and stored rather than the whole states. When model checker checks if it has seen a state before, it computes its hash first and compares it to the hashes stored in the set of hashes of visited states.

3.3 Existing Tools

Not all important properties of existing model checkers are mentioned in this section. The description is aimed at the manner in which the model checkers explore the state space or verify formulas.

3.3.1 Java PathFinder

Java PathFinder is a tool for verifying Java programs, based on the Java bytecode. Therefore it can verify any language that can be compiled into the Java bytecode. The user can select different algorithms according to which Java PathFinder explores the state space. There is a simple depth-first search and a priority-queue based search that can be configured to select the most interesting successor of a given state. The search algorithm is based on a loop that iterates through the state space until it is completely explored. [8, 9]

3.3.2 MoonWalker

MoonWalker is a model checker that automatically detects errors in CIL[3] bytecode programs - programs written for the .NET platform. It is capable of finding deadlocks and assertion violations. MoonWalker is heavily inspired by Java PathFinder. [11]

3.3.3 SPIN

SPIN[4] (**S**imple **P**romela **In**terpreter) is a software tool for verifying distributed software systems. The input language of SPIN is **PROMELA** (**Pro**cess **Me**ta **La**nguage), that supports modeling of asynchronous distributed algorithms. The properties to be verified about the system are specified as LTL formulas. SPIN, rather than performing model checking itself, generates a problem-specific model checker in C. That improves the performance and saves memory. SPIN supports various modes of program simulation - random, interactive and guided simulation. Proof techniques are either partial or exhaustive, based on either the depth-first search or the breadth-first search algorithm. [12, 13]

3.3.4 ZING

ZING is a model checker aimed at concurrent software. It uses its own modeling language for expressing models of concurrent systems. However, a component of ZING is capable of translating programs written in common programming languages into ZING models. Exploration of the state space in ZING is done in depth-first search manner. [14, 15]

3.3.5 CMC

CMC is a code model checker that works on unmodified C or C++ programs. It focuses on network implementations. The model checking algorithm of CMC is based on breadth-first search. [16]

[3] CIL - Common Intermediate Language is the CLI-defined language used by .NET Framework and Mono. CLI - Common Language Infrastructure is a specification allowing programs written in various high-level languages to be executed on different platforms without being rewritten.

[4] Using a modeling language and formulas for specifying properties, SPIN is the only traditional model checker in this section.

Chapter 4

Memory Module

This chapter describes the work that this thesis is based on, the Memory Module [2]. It is extensively described in the referenced paper, so this is only a brief description, focusing on the features important for GMC. Implementation details and algorithms are not mentioned here. From now on, Memory Module will be sometimes referred to as 'previous work'.

Memory Module is used in GMC as a memory representation of a program. It was designed specifically for it.

4.1 Interface

`MemoryState` is the main class of the Memory Module interface. It is used for the representation of the current state of the program. Apart from this, it also represents the saved states. Therefore there are two types of methods of `MemoryState`. First type of methods manipulate the memory of the simulated program and load and store values into the current state of the memory. Second type of methods is used to compute a hash of the states and save and load the states.

Memory manipulation methods have two different forms, templated and non-templated. The templated methods are used to store and load areas with a fixed layout, the non-templated methods allow to store any value at any place in the memory area.

Before we proceed to explaining how values can be stored in the simulated memory, let's explain what a value has to satisfy to be suitable for storing in the simulated memory.

4.1.1 Content

For a value to be stored in a memory module, the value must implement the `Content` interface, more precisely one of its subinterfaces, either `PtrContent` or `NonPtrContent`. This applies to values stored via non-templated methods. For templated methods, the values get wrapped into the interface automatically. `Content` contains one method to implement, `getUnitSize()` which returns the size of the value in bytes.

PtrContent

`PtrContent` interface has to be implemented for classes representing the value of a pointer. Subclasses of this class have to implement the following methods:

`Hash getTypeHash()`
: computes hash of the type of the represented value.

`getPtr()`
: returns the address that the pointer represented by this object points to.

NonPtrContent

`NonPtrContent` interface is used for classes representing non-pointer values. There is one additional method to implement:

`Hash getHash()`
: computes hash of the represented value.

4.1.2 Pointers

Classes `Ptr` and `Ref` represent pointers. While `Ptr`s are used to work with non-templated methods of `MemoryState`, `Ref`s are used to work with templated methods[1].

Ptr

`Ptr` class is used as a representation of a memory address. It is returned by the non-templated method `MemoryState::alloc(size_t size)`, which allocates a memory area of the desired size and returns an instance of `Ptr` pointing to the

[1] `Ref` is actually a templated class (`Ref<T>`).

allocated memory area. Similarly to pointers in C, instances of Ptr class can be added, subtracted and compared by arithmetic and relational operators.

Ref

Ref is a templated class that represents a reference to an allocated memory area. It is returned by the templated form of method MemoryState::alloc<T>(). Types and offsets of objects that will be stored in an area are specified by the template argument of this function. This argument must be a POD (Plain Old Data) structure. Fields of this structure determine types and offsets of the values in the area. Each structure stored this way in the simulated memory has to define how to compute its hash. This can be done by extending boost::hash() function for the type [17].

4.1.3 MemoryState

As mentioned before, MemoryState is the main class of Memory Module interface. Following sections describe MemoryState methods according to the classification mentioned at the beginning of this chapter.

Non-templated methods

`Ptr alloc(size_t size)`
Allocates an area with a specified size.

`Ptr allocExpanding()`
Allocates an area with no specified size, which automatically increases its size if needed.

`void free(Ptr ptr)`
frees an allocated memory area; the argument points to the beginning of the area.

`void setRootArea(Ptr ptr)`
sets an area that will be considered as the root for all areas. Therefore all areas should be accessible from the root area[2].

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

[2] In GMC, root area is ProgramInfo

`std::pair<ContentIterator, ContentIterator>`
`loadOverlapping(Ptr ptr, size_t s)`

Returns all the values within the area that begins at `ptr` and ends at `ptr` + `s`.

`void store(Ptr ptr, std::auto_ptr<Content> c)`

Stores a content `c` on the address that `ptr` points to.

`size_t getSize(Ptr ptr)`

returns the size of an area.

Templated methods

`Ref<T> alloc<T>()`

Allocates an area that can hold fields of structure `T`.

`Ref<T> allocExpanding<T>()`

Allocates an area that can hold fields of `T`. This area automatically increases its size if needed.

`void free(Ref<T> r)`

frees an allocated memory area, which is referenced by `r`.

`void setRootArea(Ref<T> r)`

sets specified area as the root for all areas.

`const V& load<T, V>(Ref<T> r, V T::* f)`

Loads the value from `r`, stored at offset equal to the offset of `f` in structure `T`.

`void store<T, V> (Ref<T> r, V T::* f, const V v)`

Stores a value `v` at the offset equal to offset of `f` in structure `T` into `r`.

Memory manipulation methods

`void push()`

Saves the current state of the memory on the stack of saved states.

`void pop()`

Removes the top state of the memory from the stack of saved states.

`void backtrack()`

Restores the current state according to the top state on the stack of saved states.

`Hash hash()`
: Computes hash of the top state on a stack of saved states.

`bool hasSavedState()`
: Checks if there are any states on the stack of saved states.

`size_t savedStatesCount()`
: Returns the number of states on the stack of saved states.

4.1.4 Exceptions

Memory Module checks for undefined operations and throws a corresponding exception if such an event occurs (e.g. accessing a freed memory area). The exceptions thrown by Memory Module are:

`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 has not been overwritten (neither partially). Otherwise the value is undefined.

Chapter 5

GIMPLE Model Checker

This chapter describes the implementation of GIMPLE model checker in detail, its design, architecture and non-trivial data structures and algorithms.

GIMPLE Model Checker (GMC from now on) is a program for finding errors in programs. It is not a model checker in a strict sense. It does not accept formulas of temporal logic, but checks specific properties of the implementation. GMC is based on GIMPLE - the output of the front-end of GCC, which is the input language for GMC. It is an explicit-state model checker, systematically exploring all possible paths of program execution. It also explores all possible thread interleavings, therefore it is capable of finding errors that may remain hidden during normal program execution or testing.

Basing model checker on GIMPLE brings several benefits. First, it makes the implementation of interpreter easier. Interpreting GIMPLE is much simpler than interpreting C. Second, GIMPLE being the output of the front-end of GCC, GMC is easily extensible to check not only C programs, but programs in all input languages supported by GCC, which are currently C, C++, Objective C, Fortran, Java and Ada. Third, it promises compatibility with the majority of open source projects.

GMC consists of three main parts as shown in figure 5.1. The first one is the Memory Module, designed specifically for it. It allows GMC to simulate the memory representation of the program, as well as saving, storing and comparing program execution states. It is thoroughly described in the previous chapter. The second one is the GIMPLE iterator, a module for converting a GIMPLE representation of the program to a read-only representation, suitable for the use in GMC. This read-only representation will be called GIMPLE++. The third part is the GIMPLE++ interpreter, which takes a GIMPLE++ representation

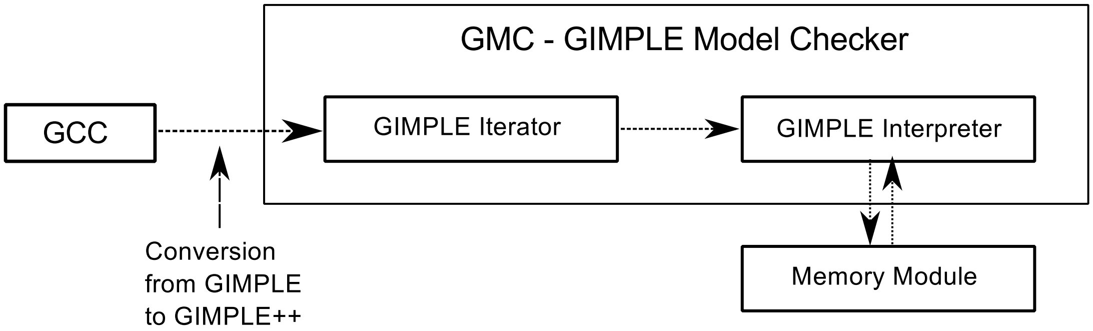

GMC - GIMPLE Model Checker
GCC
GIMPLE Iterator
GIMPLE Interpreter
Memory Module
Conversion from GIMPLE to GIMPLE++

Figure 5.1: scheme of GMC

of the program as an input. Depending on the mode in which it is run, it either interprets the simulated program (interpreter mode) or model-checks it (model checker mode). Model checking in this case means exploration of the whole state space of the program. It reports all kind of errors supported by the Memory Module - detection of memory leaks, access of freed memory areas, null pointer access, undefined pointer operations, undefined memory loads, storage/loading of values that don't lie within areas and detection of pointers that do not point to the beginning of the area, but they should. It also reports deadlocks and data concurrency errors that occurred while interleaving threads.

This is a prototype implementation of GIMPLE Model Checker, therefore it is not an all-covering model checker of C programs. It has several parts that will have to be extended and completed in future, but it is designed to make this process as easy as possible. It is capable of interpreting all C syntax constructs, however support for built-in functions and threads is just to ensure basic functionality and demonstrate correctness of implementation. The model checking parts of the program are also subject to extension. All incomplete/extensible parts will be explicitly mentioned and discussed in this chapter.

5.1 GIMPLE Iterator

GIMPLE Iterator is a module for converting the GIMPLE representation of the program to a read-only representation called GIMPLE++, suitable for use in the model checker. It is necessary to do a conversion like this, since GMC is implemented in C++, while GCC is in C. It is also much clearer to use a representation designed for purposes of GMC, rather than GCC's internal tree structures.

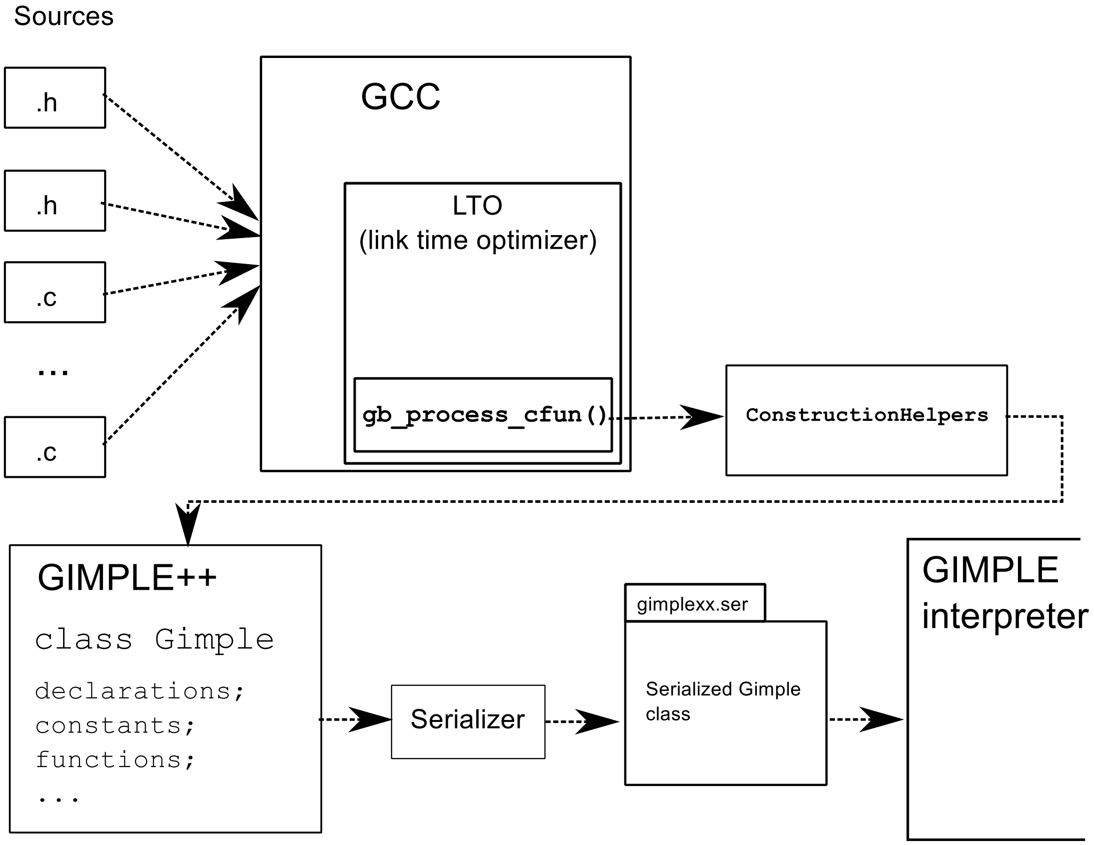


Figure 5.2: scheme of Gimple Iterator

5.1.1 Interaction with GCC

As shown in figure 5.2, GMC takes advantage of GCC's Link Time Optimizer (LTO), described in Section 2.1.2. Doing inter-procedural optimizations, LTO holds representations of all the functions defined in the program at one place. Therefore GMC does not have to worry about functions being defined in different files, since processing of GIMPLE definitions of functions is called from inside of LTO.

Main interaction of GMC with GCC is done via functions in the file **gimplexx-bridge.c**. Function `gb_process_cfun()` is called for every function definition in LTO's `materialize_cgraph()` function that processes all functions in LTO. Before the first call to `gb_process_cfun()`, global declarations get processed. `gb_process_cfun()` converts the representation of every function to a GIMPLE++ representation, extracting the necessary information directly from GCC's internal structures. This is not easy because it requires knowledge of GCC's internal representation. Other problem is that some necessary information like integer or real numbers already have a different representation for purposes of the back end of the compiler. This is not suitable for use in GMC, so the representation from earlier stages of the compiler has to be retrieved.

5.1.2 Important classes

Gimple class

The main class of GIMPLE Iterator is `Gimple`, which stores and holds the resulting GIMPLE++ representation of the program. Every time a type, declaration, constant, statement, operation or operand occurs while processing the GIMPLE representation, the appropriate subclass of class `Type`, `Decl`, `Constant`, `Stmt`, `Operation` or `Operand` gets created, holding its GIMPLE++ representation. Some of the classes were already implemented as a part of the previous work. This work extends and completes these classes, to cover representation of most of the standard use of C language. Classes inherited from the previous work are mentioned in this text for its consistency, to serve as a complete description of the interpreter. Everywhere where these classes are mentioned, it is explicitly stated that they are inherited from the previous work.

After the processing of all the functions of the simulated program is done, an instance of `Gimple` class holding the result gets *serialized*[1]. This process is described in more detail in Section 5.1.4. **gimplexx-bridge.c** communicates with `Gimple` class through functions defined in **ConstructionHelpers.cpp**. These functions have C-like declarations, so they can be called from C code.

[1] *Serialization* is a process that flattens a data structure into a sequence of characters or bits and stores them in a file that can be later turned back to the original object (*deserialized*).

However, they use C++ code which allows construction of Gimple class.

Function class

Function class represents a function definition retrieved from the GIMPLE function structure. It stores the body of the function as blocks of statements. Class Block serves this purpose. It is an equivalent to GCC's basic block[2]. Each Block contains a sequence of statements - instances of Stmt and Edges to other blocks, defining all possible ways to continue execution.

Type class

Abstract class Type defines a common interface for all classes representing GIMPLE types. Subclasses of Type class represent concrete GIMPLE types. They must implement the following methods:

Type methods	
Function	Description
getBitSize()	Number of bits required to represent the type.
getUnitSize()	Number of bytes required to represent the type.
getHash()	Hash of the type.
getName()	Name of the type.
isSame(Type&)	Type comparison.

The number of bits required to represent the type gets retrieved from GIMPLE TYPE_SIZE that is defined for every GIMPLE tree node representing type. getUnitSize() is then computed using the retrieved information. getHash() serves for memory state equivalency purposes in GMC. It returns different hashes for different types.

The name of the type is important, since the current implementation uses the name to compute the hash of the type. Therefore it is necessary to ensure that equal types have equal names, while not equal types have different names. Type name is therefore constructed from all the information stored in the corresponding subclass of Type class. For example unsigned integer type, that has size and precision of 32 bits, will have name 0int3232 (0 is for unsigned).

Possible subclasses of Type class are shown in following table:

[2] A basic block is a linear sequence of code with only one entry point and only one exit. No call graph edge leads into the block.

Type subclasses		
Class	Represented GIMPLE type	Part of prev. work
`ArrayType`	`ARRAY_TYPE`	Yes
`BoolType`	`BOOLEAN_TYPE`	Yes
`EnumType`	`ENUMERAL_TYPE`	No
`FunctionType`	`FUNCTION_TYPE`	Yes
`IntegerType`	`INTEGER_TYPE`	Yes
`PtrType`	`POINTER_TYPE`	Yes
`RealType`	`REAL_TYPE`	No
`VoidType`	`VOID_TYPE`	Yes
`RecordType`	`RECORD_TYPE`, `UNION_TYPE`	No

Decl class

`Decl` is an abstract class that defines a common interface for all classes representing GIMPLE declarations. Subclasses of `Decl` class represent concrete GIMPLE declarations. They must implement the following methods:

Decl methods	
Function	Description
`getUid()`	Unique ID of the declaration.
`getName()`	Declaration name.
`getType()`	Type of declaration, instance of some `Type` subclass.
`isSame(Decl&)`	Declaration comparison.

Unique ID for a declaration gets retrieved from `DECL_UID` that is defined for every GIMPLE `tree` node representing a declaration. Possible subclasses of `Decl` class are shown in the following table:

Decl subclasses		
Class	Represented GIMPLE declaration	Part of prev. work
`FunctionDecl`	`FUNCTION_DECL`	Yes
`VarDecl`	`VAR_DECL`	Yes
`RecordDecl`	`VAR_DECL`	No
`ParamDecl`	`PARM_DECL`	Yes
`LabelDecl`	`LABEL_DECL`	No
`ResultDecl`	`RESULT_DECL`	No
`TypeDecl`	`TYPE_DECL`	No
`FieldDecl`	`FIELD_DECL`	No
`ConstDecl`	`CONST_DECL`	No

Constant class

Abstract class Constant defines a common interface for all classes representing constants. Subclasses of Constant represent concrete GIMPLE constants. They must implement the following methods:

Constant methods	
Function	Description
getType()	Type of the constant, instance of some Type subclass.

Possible subclasses of Constant class are shown in the following table:

Constant subclasses		
Class	Represented GIMPLE constant	Part of prev. work
IntegerConstant	INTEGER_CST	Yes
EnumeralConstant	INTEGER_CST with ENUMERAL_TYPE	No
RealConstant	REAL_CST	No
StringConstant	STRING_CST	Yes

Stmt class

Stmt is an abstract class that defines a common interface for all classes representing GIMPLE statements. Subclasses of Stmt class represent concrete GIMPLE statements. They are listed in following table:

Stmt subclasses		
Class	Represented GIMPLE statement	Part of prev. work
AssignStmt	GIMPLE_ASSIGN	Yes
CondStmt	GIMPLE_COND	Yes
CallStmt	GIMPLE_CALL	Yes
ReturnStmt	GIMPLE_RETURN	Yes
GotoStmt	GIMPLE_GOTO	No
LabelStmt	GIMPLE_LABEL	No
NopStmt	GIMPLE_NOP, GIMPLE_PREDICT	Yes, No
PhiStmt	GIMPLE_PHI	No
SwitchStmt	GIMPLE_SWITCH	No

Each subclass of GIMPLE class stores all the information required to interpret the statements. These are all the operands and operations that occur in the statement. `GIMPLE_PREDICT` is an optimization feature specifying a hint for branch prediction. Ignoring it does not affect program interpretation (apart from a lowered speed). Therefore it is replaced with no operation statement.

Operand class

Abstract class `Operand` defines a common interface for all classes representing GIMPLE operands. Subclasses of `Operand` class represent concrete GIMPLE operands. They are listed in the following table:

Stmt subclasses		
Class	Represented GIMPLE operand	Part of prev. work
`ArrayRefOperand`	`ARRAY_REF`	Yes
`ComponentRefOperand`	`COMPONENT_REF`	No
`LabelExprOperand`	`CASE_LABEL_EXPR`	No
`CstOperand`	GIMPLE constant	Yes
`DeclOperand`	GIMPLE declaration	Yes
`ConstructorOperand`	`CONSTRUCTOR`	No
`AddrOperand`	`ADDR_EXPR`	Yes
`IndirectRefOperand`	`INDIRECT_REF`	Yes
`ArrayConstructorOperand`	`CONSTRUCTOR` with `RECORD_TYPE` or `UNION_TYPE`	No
`RecordConstructorOperand`	`CONSTRUCTOR` with `ARRAY_TYPE`	No

Operation class

Abstract class `Operation` defines a common interface for all classes representing GIMPLE expressions. Subclasses of `Operation` class represent concrete GIMPLE expressions. They are listed in following table:

Operation subclasses		
Class	Repr. GIMPLE expression	Part of prev. work
`NegateOperation`	`NEGATE_EXPR`	No
`AbsOperation`	`ABS_EXPR`	No
`FixTruncOperation`	`FIX_TRUNC_EXPR`	No
`FloatOperation`	`FLOAT_EXPR`	No
`BitNotOperation`	`BIT_NOT_EXPR`	No
`TruthNotOperation`	`TRUTH_NOT_EXPR`	No
`ParenOperation`	`PAREN_EXPR`	No
`NonLvalueOperation`	`NON_LVALUE_EXPR`	No
`NoOperation`	`NOP_EXPR`	Yes
`ConvertOperation`	`CONVERT_EXPR`	Yes
`PlusOperation`	`PLUS_EXPR` and `POINTER_PLUS_EXPR`	Yes
`MinusOperation`	`MINUS_EXPR`	Yes
`MultOperation`	`MULT_EXPR`	Yes
`RdivOperation`	`RDIV_EXPR`	No
`TruncDivOperation`	`TRUNC_DIV_EXPR`	No
`FloorDivOperation`	`FLOOR_DIV_EXPR`	No
`CeilDivOperation`	`CEIL_DIV_EXPR`	No
`RoundDivOperation`	`ROUND_DIV_EXPR`	No
`TruncModOperation`	`TRUNC_MOD_EXPR`	No
`FloorModOperation`	`FLOOR_MOD_EXPR`	No
`CeilModOperation`	`CEIL_MOD_EXPR`	No
`RoundModOperation`	`ROUND_MOD_EXPR`	No
`ExactDivOperation`	`EXACT_DIV_EXPR`	No
`MinOperation`	`MIN_EXPR`	No
`MaxOperation`	`MAX_EXPR`	No

Operation subclasses		
Class	Repr. GIMPLE expression	Part of prev. work
`LshiftOperation`	`LSHIFT_EXPR`	No
`RshiftOperation`	`RSHIFT_EXPR`	No
`LrotateOperation`	`LROTATE_EXPR`	No
`RrotateOperation`	`RROTATE_EXPR`	No
`BitIorOperation`	`BIT_IOR_EXPR`	No
`BitXorOperation`	`BIT_XOR_EXPR`	No
`BitAndOperation`	`BIT_AND_EXPR`	No
`TruthAndOperation`	`TRUTH_AND_EXPR`	No
`TruthOrOperation`	`TRUTH_OR_EXPR`	No
`TruthXorOperation`	`TRUTH_XOR_EXPR`	No
`LtOperation`	`LT_EXPR`	Yes
`LeOperation`	`LE_EXPR`	Yes
`GtOperation`	`GT_EXPR`	Yes
`GeOperation`	`GE_EXPR`	Yes
`EqOperation`	`EQ_EXPR`	Yes
`NeOperation`	`NE_EXPR`	Yes
`OrderedOperation`	`ORDERED_EXPR`	No
`UnorderedOperation`	`UNORDERED_EXPR`	No
`UnLtOperation`	`UNLT_EXPR`	No
`UnLeOperation`	`UNLE_EXPR`	No
`UnGtOperation`	`UNGT_EXPR`	No
`UnGeOperation`	`UNGE_EXPR`	No
`UnEqOperation`	`UNEQ_EXPR`	No
`LtGtOperation`	`LTGT_EXPR`	No
`WidenSumOperation`	`WIDEN_SUM_EXPR`	No
`WidenMultOperation`	`WIDEN_MULT_EXPR`	No

A more detailed description of the classes described in this section can be found in the Doxygen documentation on the attached CD.

5.1.3 Unsupported GIMPLE constructs

Not all of the GIMPLE constructs are supported by GMC. This section describes which the unsupported constructs are and why they are not supported.

In general, most of the unsupported constructs work with complex numbers and vectors in GIMPLE. Constructs working with complex numbers (`COMPLEX_CST`, `COMPLEX_TYPE`, `COMPLEX_EXPR`, ...) are used to represent implementations of complex arithmetic from the `complex.h` library. This was not considered as a part of the basic functionality of C and it is unimportant from the model checking point of view. To ensure basic functionality associated with `_Complex` complex type, many built-in functions handling various operations on the operands of this type would have to be implemented as well.

GIMPLE vectors (`VECTOR_CST`, `VECTOR_TYPE`, ..) are not implemented for similar reasons, since they are not needed for the interpretation of most C programs.

Other unsupported constructs include those that are used for a specific language other than C, for example `NAMESPACE_DECL` for namespace declarations or `REFERENCE_TYPE` for references, both in C++.

This section described only types of unsupported constructs. The complete list can be found on the attached CD.

5.1.4 Serialization

To split the GIMPLE Iterator part of GMC (which is coded into GCC) from the GIMPLE Interpreter (the outside-gcc part), serialization support was added to all classes representing GIMPLE in Iterator. After processing the whole GIMPLE input and creating a GIMPLE++ representation, serialization routine is executed and the whole GIMPLE++ representation gets effectively dumped to a file. When running the Interpreter part, all dumped data structures get retrieved (deserialized) and Interpreter can start. This allows to separate the Iterator and Interpreter parts and run the Interpreter independently. The second benefit is that once a GIMPLE++ representation of the program is obtained, Interpreter can be run without running GCC, which is faster.

Boost C++ Libraries serialization implementation is used to create GIMPLE++ file.

5.2 GIMPLE Interpreter

GIMPLE Interpreter is the most important part of GMC. It takes a GIMPLE++ representation created by GIMPLE Iterator and, using the Memory Module, it simulates the program execution and backtracks all reachable states and thread interleavings. During this process, it keeps track of visited states, and if it detects a previously visited state, this execution path is cut. If a Memory Module exception is thrown, a deadlock is detected, or an assertion is violated, the execution stops and the whole execution path that caused the error is displayed to the user. GMC tries to minimize the state explosion problem and running out of memory by effective storage of visited states.

Interpreting only GIMPLE instructions wouldn't be of much practical use. GIMPLE Interpreter therefore also implements some of the standard C language functions, to allow user to model check programs that use them. It also focuses on the usage of threads, since finding deadlocks and data race problems is one of the main purposes of model checking. It focuses on POSIX threads from `<pthread.h>` library, however these are not hard-coded in GMC and the use of a different thread implementation is possible.

For program exploration, GMC uses the exploration algorithm inspired by the MoonWalker model checker [18, 19]. It is a depth-first search based algorithm that explores the whole state space of the program. A mild form of partial order reduction is also implemented by classifying instructions as safe and unsafe instructions. The model checker stores its state and backtracks only if it executes an unsafe instruction. What are safe and what are unsafe instructions is further explained in Section 5.3.2.

The raw design of this module is inherited from the previous work. It has, however, served only as a simple interpreter for exemplar use, that counted only with one thread of execution. Therefore it was widely extended and adapted to meet the requirements of an advanced interpreter.

5.2.1 Interaction with GIMPLE Iterator

During the creation of a GIMPLE++ representation of the program, ergo during the deserialization of a file previously created by Gimple Iterator, `GimpleListener` class from the Interpreter is passed to a constructor of `Gimple` class. `Gimple` calls the appropriate `GimpleListener` methods when anything interesting for the Interpreter happens. Interaction is illustrated in figure 5.3. For each constant and function definition in a GIMPLE++ representation of the program, `addCst()`

or addFunc() functions get called on ConstantsManager, which saves the constant or function definition into simulated memory. They can be later retrieved during the program execution simulation by calling getCst() and getFunc() functions. The same principle is applied for global variables, that are maintained by GlobalsManager(). Function definition holds all the statements and local declarations inside the function. After the deserialization, GimpleListener ensures that the simulated memory is properly initialized with all required data needed to start the model checking of the program.

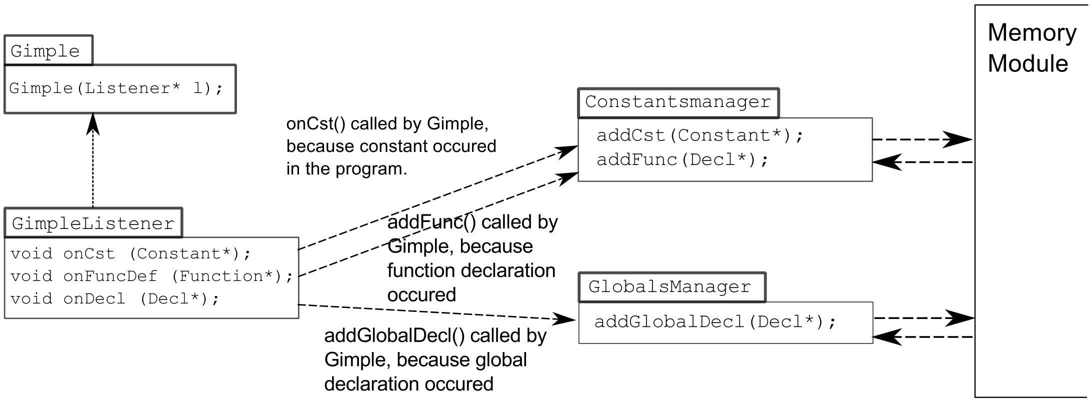

Gimple
Gimple(Listener* l);
GimpleListener
void onCst (Constant*);
void onFuncDef (Function*);
void onDecl (Decl*);
onCst() called by Gimple, because constant occured in the program.
addFunc() called by Gimple, because function declaration occured
addGlobalDecl() called by Gimple, because global declaration occured
Constantsmanager
addCst(Constant*);
addFunc(Decl*);
GlobalsManager
addGlobalDecl(Decl*);
Memory Module

Figure 5.3: Interaction of iterator and interpreter

5.2.2 Thread introduction

Before we proceed to the description of the implementation of GMC, we will define some terms related to threads that need to be explained because GMC works with threads extensively.

- **Thread** of execution can be defined as a unit of processing that can be scheduled by an operating system (by GMC in this case). It generally results from a fork of a program into two or more concurrently running tasks. Threads share some resources of the program (its instructions and global memory).
- **Mutex** is designed to be used in concurrent programming to avoid simultaneous use of a common resource, such as a global variable, by two or more

critical sections. A critical section is a piece of code in which a thread accesses a common resource. Mutex, as a shortcut for **mut**ual **ex**clusion is a program object that negotiates mutual exclusion among threads. Threads can **lock**, **unlock**, and **try to lock** a mutex (described in the next section). It can be locked only by one thread at a time, other thread trying to lock the same mutex have to wait until the owner of the mutex unlocks it. Using mutexes may result in a deadlock (explained further in Section 5.3.2). [20]

- **Thread condition** can be used as another way to synchronize threads. While mutexes implement synchronization by controlling thread access to data, condition variables allow threads to synchronize based upon the actual value of data. It is always used in conjunction with a mutex variable. Condition operations include **wait** and **signal** (described in the next section). [20]

5.2.3 Implementation

Relation between the most important classes of GIMPLE Interpreter can be seen in figure 5.4:

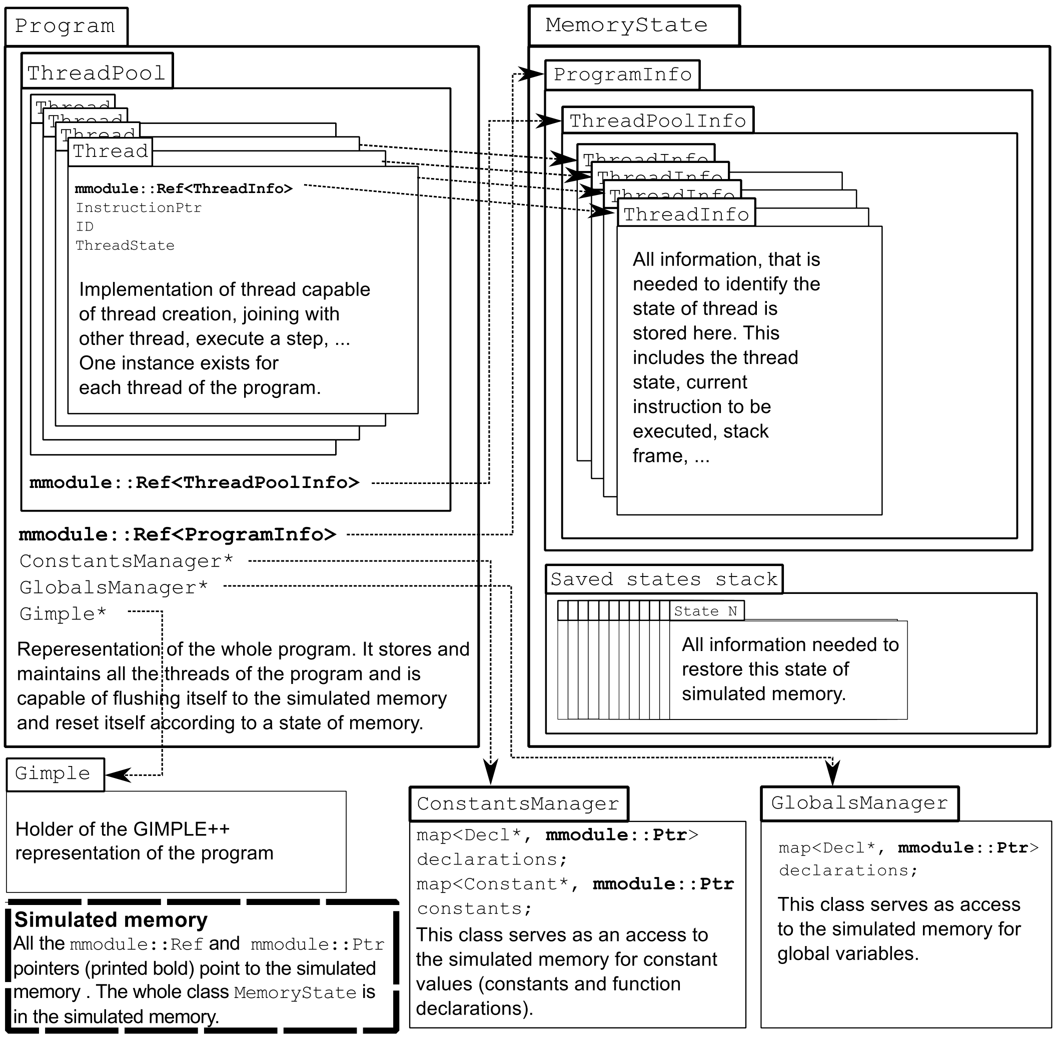

Program
ThreadPool
Thread
Thread
Thread
Thread
mmodule::Ref<ThreadInfo>
InstructionPtr
ID
ThreadState
Implementation of thread capable of thread creation, joining with other thread, execute a step, ... One instance exists for each thread of the program.
mmodule::Ref<ThreadPoolInfo>
mmodule::Ref<ProgramInfo>
ConstantsManager*
GlobalsManager*
Gimple*
Reperesentation of the whole program. It stores and maintains all the threads of the program and is capable of flushing itself to the simulated memory and reset itself according to a state of memory.
MemoryState
ProgramInfo
ThreadPoolInfo
ThreadInfo
ThreadInfo
ThreadInfo
ThreadInfo
All information, that is needed to identify the state of thread is stored here. This includes the thread state, current instruction to be executed, stack frame, ...
Saved states stack
State N
All information needed to restore this state of simulated memory.
Gimple
Holder of the GIMPLE++ representation of the program
ConstantsManager
map<Decl*, mmodule::Ptr> declarations;
map<Constant*, mmodule::Ptr constants;
This class serves as an access to the simulated memory for constant values (constants and function declarations).
GlobalsManager
map<Decl*, mmodule::Ptr> declarations;
This class serves as access to the simulated memory for global variables.
Simulated memory
All the mmodule::Ref and mmodule::Ptr pointers (printed bold) point to the simulated memory . The whole class MemoryState is in the simulated memory.

Figure 5.4: relation between classes in GIMPLE Interpreter

Before the start of the model checking, an instance of Program class gets created according to the data retrieved from the GIMPLE++ representation of the program. Afterwards, the program's ThreadPool contains one thread, the main thread of the program. Then the model checking may begin.

Program class

Class Program serves for interpreting the simulated program. It stores information about all threads in the simulated program. Calling method step() results in the interpretation of a single GIMPLE instruction, which is done by calling step() on a thread currently selected to run (i.e. the main thread at the beginning). Afterwards, by calling method flush(), the current state of the memory can be stored into simulated memory. This state can later be restored when needed during model checking by calling reset(). Section 5.2.4 describes how the active state of program is represented. Threads of the program are managed by an instance of ThreadPool class.

Important Program methods	
Function	Description
init()	Initialize program memory structures in the simulated memory.
step()	Interpret single GIMPLE instruction in the active thread.
flush()	Update state of the simulated memory according to the current state of the program.
reset()	Reset the program to a state according to the current state of the simulated memory. To be used after memory state backtracking.

ThreadPool class

Class ThreadPool manages all the threads of the simulated program. It is capable of creating a thread, destroying it and joining two threads. Apart from this, ThreadPool also manages mutexes and thread conditions of the program. What these are is described in Section 5.2.2. ThreadPool thus manages creation, deletion, locking and unlocking of mutexes as well as creation, deletion, signaling and waiting for thread conditions. The mechanism by which the mutexes and conditions work with the simulated memory is the same as for Thread class. Each Mutex and Condition maintains its own reference to the info structure stored in the simulated memory, as shown for each Thread in figure 5.4. Same as Program, ThreadPool is also able to flush() itself to simulated memory or reset() according to the current state of memory.

Thread class

Class Thread represents a single thread of the program. Like class Program, it has methods flush(), reset() and step(), the last one to interpret GIMPLE instructions. This is done via an instance of GimpleStmtInterpreter class, which is capable of executing GIMPLE++ instructions. During the program execution, there are different states that thread may be in. These are listed in the following table:

possible states of Thread	
State	Description
RUNNING	Thread is currently selected for running.
STOPPED	Thread has been terminated.
WAITLOCK	Thread is waiting to acquire a lock.
WAITCOND	Thread is waiting on a thread condition.
SLEEP	Thread is sleeping (only during interpreting, not model checking).
WAITJOIN	Thread is waiting for another thread to terminate (to join with this thread).

There are methods of class Thread that are used to get information about the current state of a thread. These are:

Thread state methods	
Function	Description
isFinished()	Returns true when there are no more instructions to execute.
isAlive()	Returns true when the thread is not stopped (its state is not THREAD_STOPPED).
isRunning()	Returns true when the thread is currently selected to run (its state is THREAD_RUNNING).
isRunnable()	Returns true when the thread is running (see above) and there are more instructions to execute.
isTerminated()	Returns true when the thread is stopped (its state is THREAD_STOPPED).

GimpleStmtInterpreter class

`GimpleStmtInterpreter` is capable of executing GIMPLE++ statements that are representations of GIMPLE statements. For each GIMPLE++ statement, it implements a visitor[3] method that interprets this statement. It is instantiated inside `Thread::step()` to interpret the statement that the current thread's instruction pointer points to.

Value class

Abstract class `Value` defines an interface for all classes representing values. It implements Memory Module's interface `Content`, and it is used to store all kinds of values into the simulated memory. Possible values are shown in the following table:

Value subclasses	
Class	Description
`BoolValue`	Value of a boolean.
`PtrValue`	Value of a pointer.
`IntegerValue`	Value of an integer.
`EnumeralValue`	Value of an enumeration constant.
`RealValue`	Value of a real number (float or double).
`StringValue`	Value of a string.
`FunctionValue`	Value of an user defined function.
`BuiltinFunctionValue`	Value of a supported built-in function or a supported function from a library.
`ThreadFunctionValue`	Value of a supported thread function.
`RecordValue`	Value of a record (structure or union).
`ArrayValue`	Value of an array.
`ArrayConstructorValue`	Value of an array constructor.
`RecordConstructorValue`	Value of a record constructor.

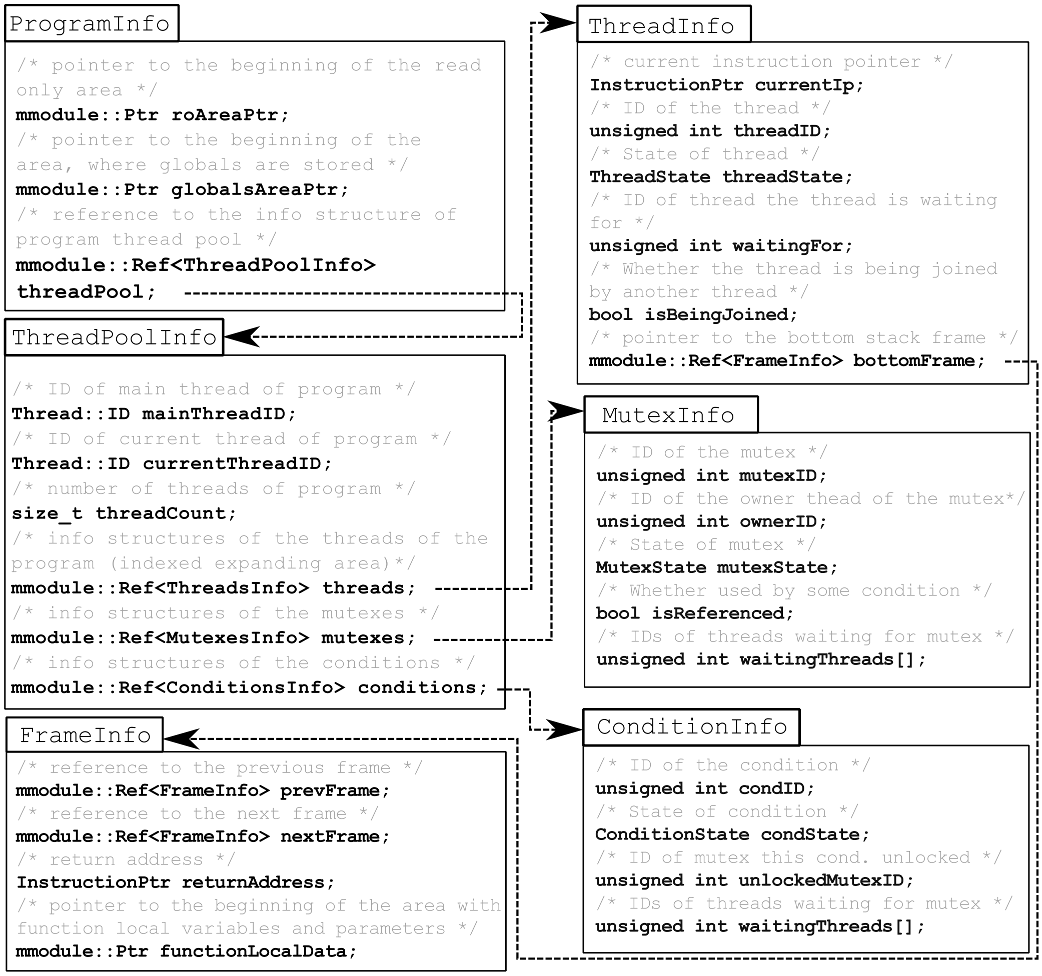
A more detailed description of the described classes can be found in the Doxygen documentation on the attached CD.

Now that these classes were described, we may proceed to the description of how the program is represented in the simulated memory.

[3] *Visitor* is a design pattern that is used to separate an algorithm from an object it operates on.

5.2.4 Program in simulated memory

The representation of a program state in the simulated memory contains just enough information needed to compare two states of the program for equivalency. Keeping the amount of information representing the state to a minimum is crucial for the model checker to avoid running out of memory. The state of a program is implemented in GMC by the `ProgramInfo` structure. A reference to this structure is stored as an instance variable of `Program` class. The contents of this structure as well as its collaboration with other `Info` structures is shown in figure 5.5.



```
ProgramInfo
/* pointer to the beginning of the read
only area */
mmodule::Ptr roAreaPtr;
/* pointer to the beginning of the
area, where globals are stored */
mmodule::Ptr globalsAreaPtr;
/* reference to the info structure of
program thread pool */
mmodule::Ref<ThreadPoolInfo>
threadPool;
```

```
ThreadPoolInfo
/* ID of main thread of program */
Thread::ID mainThreadID;
/* ID of current thread of program */
Thread::ID currentThreadID;
/* number of threads of program */
size_t threadCount;
/* info structures of the threads of the
program (indexed expanding area)*/
mmodule::Ref<ThreadsInfo> threads;
/* info structures of the mutexes */
mmodule::Ref<MutexesInfo> mutexes;
/* info structures of the conditions */
mmodule::Ref<ConditionsInfo> conditions;
```

```
FrameInfo
/* reference to the previous frame */
mmodule::Ref<FrameInfo> prevFrame;
/* reference to the next frame */
mmodule::Ref<FrameInfo> nextFrame;
/* return address */
InstructionPtr returnAddress;
/* pointer to the beginning of the area with
function local variables and parameters */
mmodule::Ptr functionLocalData;
```

```
ThreadInfo
/* current instruction pointer */
InstructionPtr currentIp;
/* ID of the thread */
unsigned int threadID;
/* State of thread */
ThreadState threadState;
/* ID of thread the thread is waiting
for */
unsigned int waitingFor;
/* Whether the thread is being joined
by another thread */
bool isBeingJoined;
/* pointer to the bottom stack frame */
mmodule::Ref<FrameInfo> bottomFrame;
```

```
MutexInfo
/* ID of the mutex */
unsigned int mutexID;
/* ID of the owner thead of the mutex*/
unsigned int ownerID;
/* State of mutex */
MutexState mutexState;
/* Whether used by some condition */
bool isReferenced;
/* IDs of threads waiting for mutex */
unsigned int waitingThreads[];
```

```
ConditionInfo
/* ID of the condition */
unsigned int condID;
/* State of condition */
ConditionState condState;
/* ID of mutex this cond. unlocked */
unsigned int unlockedMutexID;
/* IDs of threads waiting for mutex */
unsigned int waitingThreads[];
```

Figure 5.5: Collaboration diagram of *Info classes

As can be seen in figure 5.5, `ProgramInfo` contains `ThreadPoolInfo`, which contains `ThreadInfo` for each thread of the program, `MutexInfo` for each mutex and `ConditionInfo` for each thread condition. Further, it contains a pointer to the read-only area with the defined functions and constants as well as a pointer to the area with the global variables. The `ThreadInfo` contains the current instruction pointer, `FrameInfo` pointer to the bottom stack frame, the running state of the thread and a pointer to the thread's local data. `FrameInfo` representing stack frame, contains references to the previous and next stack frames, a pointer to the area where the local variables and parameters of the function are stored as well as the return address of the function.

5.3 Interpreter vs. Model Checker

GMC works in two possible modes, as an interpreter and as a model checker.

5.3.1 Interpreter

If GMC is run as an interpreter, the program gets interpreted step by step and the thread scheduling policy is determined by `ThreadPool` class. The policy is that a thread is selected to run after a previously selected thread is finished or its state is not running (e.g. it is waiting for another thread to join, waiting for a mutex or for a thread condition). It is non-preemptive scheduling policy since the interpreter is not capable of forcibly selecting a thread to run.

During interpretation, visited states are not stored, therefore no state equivalency matching is done. Following pseudocode shows how the interpreter algorithm works:

Algorithm 1 The interpreter

```
program.init()
while not program.isFinished() do
  program.step()
  if not program.currentThread().runnable() then
    program.selectRunnableThread()
  end if
end while
```

Even though less errors can be detected in this mode than in model checking mode, it can be used for programs that are too complex to be model checked.

Interpreter exceptions

Errors detected by interpreter are the Memory Module exceptions described in Section 4.1.4. Apart from these, there are other exceptions that can be thrown:

Interpreter exceptions	
Class	Thrown when
`AbortException`	Function `abort()` was called.
`ExitException`	Function `exit()` was called[4].
`AssertException`	Assertion failed (`assert()` from `assert.h`).
`InvalidPtrConvertException`	Invalid pointer conversion occurred (e.g. assigning nonzero integer to pointer).
`OverlappingAreasException`	Overlapping memory areas were passed as the arguments to the function `memcpy()`, `strcpy()` or `strcat()`.

[4] Apart from the other exceptions, this one is not considered to be an error, since `exit()` is a function for normal program termination (compared to `abort()`, which signals an abnormal program termination).

5.3.2 Model checker

If GMC is run as a model checker, an algorithm that explores all possible ways of program execution is run. This algorithm is inspired by the MoonWalker model checker (which is inspired by Java PathFinder) and it was selected for its relative simplicity and easy extensibility. The algorithm in pseudocode is shown on the next page.

It is a depth-first search based algorithm, that runs until all possible program execution paths have been explored, a deadlock was encountered or a Memory Module exception was thrown. Each iteration in the main loop corresponds to a step forward or a sequence of steps backward. While stepping forward means execution of instruction(s) and eventual rescheduling, stepping backward means restoring a previous memory state and rescheduling (i.e. backtracking in the depth-first search).

The rescheduler (called by `reschedule()`) compares the current memory state with the previously stored ones, and if it finds that it is in a previously visited state, it backtracks to the previous state. Additionally, it may select a different thread as the running thread of the program for the the next iteration.

Taking the pseudocode step-by-step, lines 1-3 initialize the model checked program and store its initial state into the simulated memory. Line 5 initializes the structure that will be returned from rescheduler. This structure contains the ID of next thread that rescheduler selected to run (`resch_ret.next`), number of steps the rescheduler backtracked (`resch_ret.backtrackCount`) and whether there are any more non-executed possible paths of the program to explore (`resch_ret.continue`). Variables on lines 6 and 7 are boolean flags that control the execution and the rescheduler respectively.

The first `if` in the repeat cycle starting on line 12 checks if the current thread has any more instructions to be executed and whether the thread is not waiting for an event that is not available at the moment (e.g. waiting for another thread to join). If this is the case, step of the program is executed (line 29) and it is checked whether the rescheduler should be called. Rescheduler is called (line 24) if the executed instruction is unsafe. This process is repeated until the whole state space of the program has been explored or a deadlock has occurred.

A deadlock occurs when none of the threads are runnable but not all of the threads are terminated. This is an undesirable situation, usually the cause of incorrect locking or synchronization and is often hard to detect by classical testing. Checking for deadlocks is implemented inside the rescheduler.

Function `isCurrentInstructionUnsafe()` on line 14 marks an instruction as **safe**, when its result cannot depend on the different scheduling policies of

Algorithm 2 The model checker

```
$program.init()$
$memoryState.init()$
$memoryState.store(program)$

$resch\_ret \leftarrow empty$
$execute \leftarrow false$
$reschedule \leftarrow true$

**repeat**
    $currentThread \leftarrow program.currentThread()$

    **if** $currentThread.runnable()$ **then**
        $execute \leftarrow true$
        $reschedule \leftarrow currentThread.isCurrentInstructionUnsafe()$
    **else**
        $execute \leftarrow false$
        $reschedule \leftarrow true$
        **if** $currentThread.running()$ **then**
            $currentThread.terminate()$
        **end if**
    **end if**

    **if** $reschedule$ **then**
        $resch\_ret \leftarrow reschedule(program, memoryState)$
        $execute \leftarrow execute$ **and** $resch\_ret.backtrackCount \geq 0$ **and** $resch\_ret.continue$
    **end if**

    **if** $execute$ **then**
        $program.step()$
    **end if**

    **if** $reschedule$ **then**
        $program.currentThread \leftarrow resch\_ret.next$
    **end if**
**until** $resch\_ret.continue \neq false$
```

threads. That means all instructions that manipulate with local variables are marked as safe. On the other hand, **unsafe** instructions can be affected by different scheduling policies. These are the instructions that manipulate global variables.

There are some cases when the rescheduler is not called during the algorithm and is skipped, but these are not mentioned for the pseudocode would become unreadable. These are cases when the state of the memory does not change between two calls of the rescheduler and it can therefore conclude that the state was already visited.

Model checker exceptions

While model checking a program, (apart from Memory Module exceptions), a `DeadlockException` can be thrown when a deadlock occurs.[5]

[5] Data race problems can be detected by user-defined assertions.

5.3.3 Implemented functions

GMC implements some standard C functions and some library functions that can be used in the simulated programs. Behavior of some of them differs from the original behavior. These differences include turning unsafe functions into safe, for example `memcpy` throws an exception when the destination memory space overlaps with the source memory space. All supported functions are listed in this chapter including their descriptions that may differ from original ones. Apart from exceptions that these functions throw and are listed in this chapter, these function may throw more exceptions resulting from use of Memory Module - the exceptions described in Section 4.1.4.

GMC supports these functions:[6]

- `malloc()` (N)[7] - allocates bytes of dynamic memory and returns a pointer to the allocated space.

- `free()` (N) - frees the memory space, which was allocated by a call to `malloc()` before.

- `printf()` (Y) - writes output to standard output according to a specified format. Types of arguments specified in the format string are ignored, the arguments are printed according to their real type.

- `puts()` (N) - writes a string and a trailing newline to the standard output.

- `putchar()` (N) - writes a character to the standard output.

- `scanf()` (Y) - scans input from the standard input stream. Format string is ignored, `scanf` is capable of reading only integer numbers.

- `assert()` (N) - (actually a macro) checks if an assertion was false. If it was, `AssertException` is thrown and the execution stops.

- `sleep()` (Y) - has different behavior in interpreter and model checker. In the interpreter, it makes the calling thread sleep until a specified number of seconds have elapsed. Since the interpreter is slower than normal program execution, the effect of sleep will be different from the original, so this

[6] Functions malloc, free, printf, puts, and scanf were implemented as a part of the previous work.

[7] N means that behavior of the function does not differ from the original behavior, Y means that it does and it is stated how.

should be taken into account. In the model checker, sleeps work only as a backtracking spot (an equivalent for an unsafe instruction), where the thread only resigns to the position of the currently running thread.

- `abort()` (N) - causes abnormal program termination, `AbortException` is thrown and execution stops.
- `exit()` (N) - causes normal program termination, `ExitException` is throw but is not considered as an error.
- `abs()`, `labs()`, `llabs()` (N) - computes the absolute value of an integer.
- `atoi()`, `atol()`, `atoll()`, `strtol()` (Y) - converts a string to an integer. The first three functions are implemented like the latter, so they also detect errors.
- `atof()`, `strtod()` (Y) - converts a string to a double. `atof()` is implemented the same as `strtod()`, therefore `atof()` also detect errors. The use of the second parameter of `strtod()` is not implemented. It is ignored, if it is not null.
- `rand()` (N) - returns a pseudo-random integer.
- `srand()` (N) - sets a seed for generating pseudo-random numbers.
- `memcpy()` (Y) - copies bytes from one memory area to another. If the areas overlap, an `OverlappingAreasException` is thrown.
- `memmove()` (N) - copies memory from one place to another.
- `strcat()` (Y) - concatenates two strings. If the strings overlap, an `OverlappingAreasException` is thrown.
- `strcpy()` (Y) - copies a string from one location to another. Again, if the strings overlap, an `OverlappingAreasException` is thrown.
- `strcmp()` (N) - compares two strings.
- `strlen()` (N) - calculates the length of a string.

Definitions of all of these functions, as well as functions for manipulating threads described in the next section are kept at one place. They are all in

BuiltinFunctions.cpp for easy extensibility. Adding a new function implementation means adding its implementation into this source (and its header to **BuiltinFunctions.h**) plus ensuring its proper recognition in `GimpleListener`'s `onDecl()` method's visitor for function declarations.

5.3.4 Thread support

Support for threads in GMC is aimed at POSIX threads from the `<pthread.h>` library. However, there is a way to use different thread implementations described at the end of this section. In the following text, POSIX thread names for functions and types will be used. Since thread attributes (`pthread_attr_t` structure and its use) are not implemented in GMC, threads work only in their default mode. That means that the behavior of all of these functions differ from the original. The common sign for all these functions is that they try to detect as many errors as possible. On success, all the functions return 0. Starting with `pthread_mutex_create`, all the functions return `EINVAL` when trying to work with uninitialized mutexes or thread conditions.

Current support for threads consists of:

- `pthread_create()` - starts a new thread in the program.
- `pthread_join()` - blocks execution of the running thread until the joined thread is finished. If the joined thread has already terminated, the function returns immediately with an error code. A deadlock may occur (for example when three threads are waiting in a loop to join), which is detected and `DeadlockException` is thrown.
- `pthread_exit()` - terminates the calling thread. It is not capable of returning a value via a parameter that is available to another thread in the same process that calls `pthread_join()`.
- `pthread_t` - type holding thread identifiers to access the threads.
- `pthread_mutex_init()` - initialize a mutex variable.
- `pthread_mutex_destroy()` - set a mutex state from uninitialized to unlocked. Error code `EBUSY` is returned when trying to destroy a locked mutex or a mutex used by a thread condition.

- `pthread_mutex_lock()` - lock a mutex or block the calling thread if the mutex is already locked. Error code `EDEADLK` is returned when trying to lock a mutex locked by the same thread.
- `pthread_mutex_trylock()` - lock a mutex or return immediately if the mutex is already locked. Error code `EBUSY` is returned if the mutex is locked.
- `pthread_mutex_unlock()` - release a mutex variable. Error code `EPERM` is called when trying to unlock a mutex that is not owned by the current thread.
- `pthread_mutex_t` - type holding mutex identifiers to access mutexes.
- `pthread_cond_init()` - initialize a thread condition.
- `pthread_cond_destroy()` - uninitialize a thread condition. Error code `EBUSY` is returned when trying to destroy a condition that is in use (by some condition wait function).
- `pthread_cond_signal()` - unblock a thread waiting on a condition variable.
- `pthread_cond_wait()` - Blocks a thread until the specified condition is signaled. Should be called while a mutex is locked by the calling thread. This function releases the locked mutex while waiting and after the signal is received from `pthread_cond_signal()`, the mutex will be automatically locked for use by the thread. Error code `EINVAL` is also returned when different mutexes were supplied for concurrent condition wait operations on the same condition variable, or `EPERM` when the mutex is not locked by the current thread (or not locked at all).
- `pthread_cond_t` - type holding identifiers to access the thread conditions.

Thread configuration

As mentioned before, there is some functionality available for the case when other threads implementation than POSIX threads is needed to be used in GMC. POSIX threads are not hard-coded in GMC, but their names and the order of their arguments are in the configuration file that is show in Appendix A.

For example, let's define the function for creating threads to be POSIX function:

```
int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
                   void *(*start_routine) (void *), void *arg);
```

The first argument specifies where to store the the ID of the new thread. The second argument determines attributes for the new thread. The third argument defines the function where the execution of the new thread starts. Finally, the fourth argument specifies the data to be passed to the new thread.

In the thread configuration file, the definition for the described function would be:

```
CREATE=pthread_create 1 3 4
```

It defines the name of the function for creating threads and positions of ID storage, start function and data arguments.

So there is variability in the syntax for threads, but the behavior is the same as for POSIX threads. Threads with different behavior would have to be implemented into the built-in functions (see previous section).

Chapter 6

Usage and Examples

In this chapter, instructions for the usage of GMC are presented, including examples.

6.1 Usage

Installation and configuration of GMC is described in detail in README files on the attached CD. To run GMC, there is prepared script called GMC that calls gcc to create a GIMPLE++ representation of the program and then runs the model checker itself, using this representation as an input. The scheme for running GMC is:

```
GMC -i|-m [--dump] source1 [source2 ...]
```

Therefore GMC has to be run with one of these options:

- **-i** Interpretation mode. GMC only interprets the input program, does not model check it. Thread policy coded in the interpreter is the following: a thread is selected to run after the previously run thread has finished or it is not runnable (e.g. waiting for another thread to join, waiting for a locked mutex or waiting for a thread condition). This policy is called sequential. In this mode, the model checker reports only Memory Module errors (exceptions) described in Section 4.1.4, plus interpreter errors (exceptions) described at the end of Section 5.3.1.

- **-m** Model checking mode. GMC runs the model checking algorithm on the input program, explores all possible paths of program execution and backtracks on unsafe instructions. Apart from Memory Module errors (exceptions), it reports interpreter and model checker exceptions described at the end of Section 5.3.2, most notably deadlocks and assertion violations. Assertions may be used to check for data race problems (see example in next section). If any of these occurred, the sequence of steps of the program that has led to the error is reported to the user.

An optional option –**dump** can be used to print the GIMPLE representation of the program to a file named **GIMPLE.dump**.

6.2 Examples

6.2.1 Data race example

Consider the following C code (let's call it **datarace.c**):

```
#include <pthread.h>
#include <stdio.h>
#include <assert.h>

// Global variable accessed by both main thread and thread t1.
int i = 0;

void* thread1(void* data)
{
    i += 1;
    // This thread requires to be
    // the second to access the variable.
    assert(i == 2);
}
int main()
{
    pthread_t t1;
    pthread_create(&t1, NULL, thread1, NULL);
    i += 1;
    // This thread requires to be the first
    // to access the variable.
    assert(i == 1);
    pthread_join(t1, NULL);
    printf("%s\n", "OK!");
    return 0;
}
```

This is a simple test that may cause a data concurrency error. Both threads access the same global variable, but they count on a specific order of the execution, as described in comments. When running this code in interpretation mode, we get:

```
~]$ ./GMC -i datarace.c
OK!
```

The sequential scheduling policy of the threads caused the correct order of accesses to the global variable `i`, so the data concurrency was not recognized. This may be the case also during a normal execution of the program. But when we run it in model checker mode, we get:

```
~]$ ./GMC -m datarace.c
OK!
*********
Exception encountered:
Assertion failed. The assertion was:
datarace.c: main: Assertion ‘i == 1’ failed.
*********
Logger: Instructions executed:
t1: i.4 = i
t1: i.5 = i.4 + 1
t1: i = i.5
main: i.2 = i
main: if (i.2 != 1)
main: &__assert_fail (&"i == 1"[0], &"datarace.c"[0],
22, &__PRETTY_FUNCTION__[0])
```

While backtracking the second possible order of the threads, the assertion in thread `t1` failed, therefore an exception was thrown and the order of the executed GIMPLE++ statements that have led to the error were reported to the user. The model checker has therefore successfully recognized the possible data race problem. The GIMPLE++ representation of the program from this example can be found in appendix B.

6.2.2 Deadlock example

Consider the following C code (let's call it deadlock.c):

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

pthread_mutex_t m1, m2;

void* thread1(void* data)
{
    // lock the mutexes in reverse order than the main thread
    // order: m2 first, then m1
    pthread_mutex_lock(&m2);
    pthread_mutex_lock(&m1);
    pthread_mutex_unlock(&m1);
    pthread_mutex_unlock(&m2);
}
int main()
{
    pthread_t t1;
    pthread_mutex_init(&m1, NULL);
    pthread_mutex_init(&m2, NULL);
    pthread_create(&t1, NULL, thread1, NULL);
    // lock the mutexes in order: m1 first, then m2
    pthread_mutex_lock(&m1);
    pthread_mutex_lock(&m2);
    pthread_mutex_unlock(&m2);
    pthread_mutex_unlock(&m1);
    pthread_join(t1, NULL);
    printf("%s\n", "OK!");
    return 0;
}
```

This is a test for detecting a deadlock. The main thread locks mutexes m1 and m2, the second thread locks them in the reverse order from the main thread. This may or may not result in a deadlock, when the main thread has locked mutex m1 and is waiting for m2, while the thread t1 has locked mutex m2 and is waiting for m1. Running this program in GMC in interpretation mode, we get:

```
~]$ ./GMC -i deadlock.c
OK!
```

Again, sequential scheduling policy has caused that at first the main thread locked an unlocked both mutexes and then the second thread did the same, but in the reverse order. But since both mutexes were already unlocked, no deadlock occurred. Running the same program in the model checking mode results in:

```
~]$ ./GMC -m deadlock.c
OK!
OK!
OK!
*********
Exception encountered:
Deadlock occurred.
*********
Logger: Instructions executed:
t1: &pthread_mutex_lock (&m2)
t1: &pthread_mutex_lock (&m1)
main: &pthread_mutex_lock (&m2)
```

We can see that while backtracking all possible thread interleavings, GMC also detected the deadlock in one of them.

Chapter 7

Conclusion

GIMPLE Model Checker is an explicit-state code model checker of C. It can detect many errors caused by incorrect work with memory, which is a common problem when writing C programs. It can perform backtracking on the program, explore the whole state space of the program and therefore detect deadlocks and data concurrency errors via assertions.

To avoid the state explosion problem, GMC uses an effective storage for the states of the program and while backtracking, it maintains a list of states it was in before. When any of these states is revisited, the exploration does not continue, therefore duplicate work is avoided. A simple form of partial order reduction is implemented as well, by classifying instructions as safe and unsafe. The state of the memory is saved only when executing unsafe instructions.

GMC takes GIMPLE as its input which is simpler to interpret than C and it is language independent representation. GMC is therefore not limited to use only with C programs. It can be easily extended to support all languages that use GIMPLE as an intermediate representation. Such an extension would mean completing the support for all the unsupported GIMPLE constructs (or at least basic language-specific constructs) and implementing the support for the built-in and library functions for the languages.

Function support can be extended also for C programs, since in its present form, it covers only some of the basic C functions. Extensibility for functions was considered during the design of GMC to make this process as easy as possible. Another subject to extension is the GIMPLE Iterator, which does not cover the whole GIMPLE. However, in its present form, it covers advanced use of C.

Since GMC is a prototype implementation of the model checker, it uses only a very simple exploration algorithm. It may become insufficient if GMC was heavily extended.

Memory Module, which was not part of this work, can be improved by implementing advanced techniques of incremental hashing or support for symbolic values.

Bibliography

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

[2] Kouba J. (2009): *Memory Representation for Model Checker of C/C++*, Master's thesis, Charles University in Prague.

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

[4] http://gcc.gnu.org/wiki/LinkTimeOptimization/

[5] http://www.openmp.org/

[6] Müller-Olm M., Schmidt D.A., Steffen B. (1999): *Model checking: a tutorial introduction.* Proc. 6th Static Analysis Symposium, G. File and A. Cortesi, eds., Springer LNCS 1694, pp. 330-354.

[7] Jhala R., Majumdar R. (2009): *Software model checking.* ACM Computing Surveys (CSUR), Volume 41 Issue 4.

[8] 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.

[9] http://babelfish.arc.nasa.gov/trac/jpf

[10] Clark E. M., Grumberg O., Peled D. (2000): *Model Checking*, MIT Press, Cambridge, MA.

[11] http://wwwhome.ewi.utwente.nl/ ruys/moonwalker/

[12] Holzmann G. J. (1997): *The model checker Spin*, IEEE T/SE, Vol. 23, No. 5, pp. 279-295.

[13] http://spinroot.com/spin/whatispin.html/

[14] 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.

[15] http://research.microsoft.com/en-us/projects/zing/

[16] 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.

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

[18] de Brugh V. Y. (2006): *Software Model Checking for Mono*, Master's thesis, University of Twente.

[19] de Brugh V. Y. (2007): *Optimizing Techniques for Model Checkers*, Master's thesis, University of Twente.

[20] https://computing.llnl.gov/tutorials/pthreads/

Appendix A

Thread configuration file

This is an example thread configuration file, defining POSIX threads from the `pthread.h` library to be used as threads recognized by GMC. A copy of it can be found on the attached CD.

```
# Defines a function for thread creation.
# - First number specifies which argument
# specifies where to store the created thread.
# - Second number specifies which argument specifies
# the name of the function to be started after the
# thread is created.
# - Third number specifies which argument
# specifies the data to be passed to the new thread.
CREATE=pthread_create 1 3 4

# Defines a function for thread join
# (wait for the specified thread to terminate).
# - The number specifies which argument
# specifies the thread to be joined.
JOIN=pthread_join 1

# Defines a function for thread exit.
# - The number specifies which argument
# specifies the data to be returned.
EXIT=pthread_exit 1

# Defines a type holding thread identifiers
# to access the threads
TYPE=pthread_t

# The same pattern as mentioned in the comment before
# applies for the following functions.
# -First number specifies which argument specifies
# where to store/obtain mutex/condition
MUTEX_INIT=pthread_mutex_init 1
MUTEX_DESTROY=pthread_mutex_destroy 1
MUTEX_LOCK=pthread_mutex_lock 1
MUTEX_TRY_LOCK=pthread_mutex_trylock 1
MUTEX_UNLOCK=pthread_mutex_unlock 1
MUTEX_TYPE=pthread_mutex_t
COND_INIT=pthread_cond_init 1
COND_DESTROY=pthread_cond_destroy 1
COND_SIGNAL=pthread_cond_signal 1
COND_WAIT=pthread_cond_wait 1
COND_TYPE=pthread_cond_t
```

Appendix B

Examples in GIMPLE

B.1 Data race

This is the GIMPLE representation of the example file **datarace.c** from Section 6.2.1:

```
mmain ()
gimple_bind <
  int i.0;
  int i.1;
  int i.2;
  pthread_t t1.3;
  int D.3296;
  pthread_t t1;
  static const char __PRETTY_FUNCTION__[5] = "main";

  gimple_call <pthread_create, NULL, &t1, 0B, thread1, 0B>
  gimple_assign <var_decl, i.0, i, NULL>
  gimple_assign <plus_expr, i.1, i.0, 1>
  gimple_assign <var_decl, i, i.1, NULL>
  gimple_assign <var_decl, i.2, i, NULL>
  gimple_cond <ne_expr, i.2, 1, <D.3293>, <D.3294>>
  gimple_label <<D.3293>>
  gimple_call <__assert_fail, NULL, &"i == 1"[0],
  &"datarace.c"[0], 22, &__PRETTY_FUNCTION__[0]>
  gimple_label <<D.3294>>
  gimple_assign <var_decl, t1.3, t1, NULL>
  gimple_call <pthread_join, NULL, t1.3, 0B>
  gimple_call <__builtin_puts, NULL, &"OK!"[0]>
  gimple_assign <integer_cst, D.3296, 0, NULL>
  gimple_return <D.3296>
>


thread1 (void * data)
gimple_bind <
  int i.4;
  int i.5;
  int i.6;
  static const char __PRETTY_FUNCTION__[8] = "thread1";

  gimple_assign <var_decl, i.4, i, NULL>
  gimple_assign <plus_expr, i.5, i.4, 1>
  gimple_assign <var_decl, i, i.5, NULL>
  gimple_assign <var_decl, i.6, i, NULL>
  gimple_cond <ne_expr, i.6, 2, <D.3301>, <D.3302>>
  gimple_label <<D.3301>>
  gimple_call <__assert_fail, NULL, &"i == 2"[0],
  &"datarace.c"[0], 13, &__PRETTY_FUNCTION__[0]>
  gimple_label <<D.3302>>
>
```

B.2 Deadlock

This is the GIMPLE representation of the example file **deadlock.c** from Section 6.2.2:

```
main ()
gimple_bind <
  pthread_t t1.0;
  int D.3274;
  pthread_t t1;

  gimple_call <pthread_mutex_init, NULL, &m1, 0B>
  gimple_call <pthread_mutex_init, NULL, &m2, 0B>
  gimple_call <pthread_create, NULL, &t1, 0B, thread1, 0B>
  gimple_call <pthread_mutex_lock, NULL, &m1>
  gimple_call <pthread_mutex_lock, NULL, &m2>
  gimple_call <pthread_mutex_unlock, NULL, &m2>
  gimple_call <pthread_mutex_unlock, NULL, &m1>
  gimple_assign <var_decl, t1.0, t1, NULL>
  gimple_call <pthread_join, NULL, t1.0, 0B>
  gimple_call <__builtin_puts, NULL, &"OK!"[0]>
  gimple_assign <integer_cst, D.3274, 0, NULL>
  gimple_return <D.3274>
>


thread1 (void * data)
gimple_bind <
  gimple_call <pthread_mutex_lock, NULL, &m2>
  gimple_call <pthread_mutex_lock, NULL, &m1>
  gimple_call <pthread_mutex_unlock, NULL, &m1>
  gimple_call <pthread_mutex_unlock, NULL, &m2>
>
```

Appendix C

The attached CD

The sources and the documentation of the GIMPLE Model Checker can be found on the attached CD. Most important content of the CD is listed in this section, however a more detailed description can be found in `README` files on the CD.

`dist/`
: Contains the script for running GMC as well as the scripts for running the automated tests of GMC.

`gcc/`
: Contains version of GCC that was used to implement GIMPLE Iterator part of GMC (version 4.5.0).

`gmc/`
: Contains the sources and the Doxygen documentation of GMC.

`thesis/`
: Contains the text of this thesis.