Aalto University, School of Science
ITMO University, Faculty of Information Security and Computer Technologies

Master's Programme in Computer, Communication and Information Sciences
International double degree programme

Artem YUSHKOVSKIY

Automated Analysis of Weak Memory Models

Master's Thesis
Espoo, Finland & Saint Petersburg, Russia, ?.?.2018 **TODO**

Supervisors: Assoc. Prof. Keijo Heljanko
Docent Igor I. Komarov

Aalto University
School of Electrical
Engineering

Aalto University, School of Science
ITMO University, Faculty of Information Security and Computer Technologies

Master's Programme in Computer, Communication and Information Sciences
International double degree programme ABSTRACT

Author: Artem Yushkovskiy

Title:
Automated Analysis of Weak Memory Models

Date: ?.?.2018 **TODO** **Pages:** v + 48

Supervisors: Assoc. Prof. Keijo Heljanko
Docent Igor I. Komarov

In id fringilla velit. Maecenas sed ante sit amet nisi iaculis bibendum sed vel elit. Quisque eleifend lacus nec ipsum lobortis ornare. Nam lectus diam, facilisis eget porttitor ac, fringilla quis massa. Phasellus ac dolor sem, eget varius lacus. Sed sit amet ipsum eget arcu tristique aliquam. Integer aliquam velit sit amet odio tempus commodo. Quisque commodo lacus in leo sagittis vel dignissim quam vestibulum. Cras fringilla velit et diam dictum faucibus. Pellentesque at eros non mauris auctor euismod. Nullam convallis arcu vel lectus sollicitudin rutrum. Praesent consequat, nisl at pretium posuere, neque arcu dapibus lacus, ut sollicitudin elit velit ultricies libero. Vestibulum ante ipsum primis in faucibus orci luctus et ultrices posuere cubilia Curae;

Nulla semper hendrerit molestie. Pellentesque blandit velit sit amet est vestibulum faucibus. Nullam massa turpis, venenatis non mollis fringilla, mattis et diam. Fusce molestie convallis elementum. Morbi nec lacus dapibus arcu mollis gravida. Aliquam erat volutpat. Nam vitae magna nunc. Nunc ut ipsum at massa porttitor vestibulum. Praesent diam lorem, ultrices nec vestibulum id, volutpat nec lacus.

Keywords: **TODO:** Thesis template, master's thesis

Language: English

Acknowledgements

In id fringilla velit. Maecenas sed ante sit amet nisi iaculis bibendum sed vel elit. Quisque eleifend lacus nec ipsum lobortis ornare. Nam lectus diam, facilisis eget porttitor ac, fringilla quis massa. Phasellus ac dolor sem, eget varius lacus. Sed sit amet ipsum eget arcu tristique aliquam. Integer aliquam velit sit amet odio tempus commodo. Quisque commodo lacus in leo sagittis vel dignissim quam vestibulum. Cras fringilla velit et diam dictum faucibus. Pellentesque at eros non mauris auctor euismod. Nullam convallis arcu vel lectus sollicitudin rutrum. Praesent consequat, nisl at pretium posuere, neque arcu dapibus lacus, ut sollicitudin elit velit ultricies libero. Vestibulum ante ipsum primis in faucibus orci luctus et ultrices posuere cubilia Curae;

Saint Petersburg & Espoo
?.?.2018 **TODO**

Artem Yushkovskiy

Abbreviations

AST	Abstract Syntax Tree
CF	Control-Flow
CPU	Central Processor Unit
DF	Data-Flow
DTO	Data-Transfer Object
OOP	Object-Oriented Programming
SMT	Satisfiability Modulo Theories
UML	Unified Modeling Language
WMM	Weak Memory Model
TODO	**MORE**

Contents

Chapter 1

Introduction

1.1 Problem statement

Most modern computer systems contain large parts that operate concurrently. Though parallelisation of the system can improve its performance drastically, it opens numerous of problems connected to correctness, robustness and reliability, which makes the concurrent program design one of the most difficult problems of programming [McK].

Traditionally, studies related to concurrent programming focus more on fundamental theoretical questions of designing race-free and lock-free parallel algorithms, asynchronous data structures and synchronisation primitives of a programming language [Ben06]. Unfortunately, when it comes to real-world concurrent programs, the algorithmic level of abstraction is not enough for guaranteeing their correctness. The reasons of this fact lie in the code optimisations that both compiler and hardware perform in order to increase performance as much as possible [AG96]. For instance, Figure 1.1 provides simple example of reachability of the state '`(0:EAX=0 /\ 1:EAX=0)`' on the x86 architecture (such little examples that illustrate specific behaviour of a WMM are called *litmus tests*). This state is allowed because in the x86 architecture each processor may cache the write to shared memory variable into its local write buffer, so that they do not immediately become visible by processes running on other cores.

In the example, the write '`MOV [x],1`' performed by process `P0` stores value `1` to the shared variable `[x]` into the write buffer of process `P0`. Meanwhile, the write cache of the process `P1` may not have an updated version of the variable `[x]`, neither the main memory, so that the read '`MOV EBX,[x]`' performed in the process `P1` may read the initial value `0` even if this variable

{ x=0; y=0; }	
P0	P1
MOV [x],1 MOV EAX,[y]	MOV [y],1 MOV EAX,[x]
exists (0:EAX=0 /\ 1:EAX=0)	
x86-TSO: allow	

Figure 1.1: *Store buffering (SB): A litmus test on write-read reordering allowed under the x86-TSO and forbidden under the SC memory model*

has been already updated in the other process. The problems enumerated above have led to the need for formalisation of semantics of memory operations within different concurrent architectures defined by *memory models*.

The first memory model for concurrent systems was formulated by Leslie Lamport back in 1979 [Lam79]. This memory model, called the *sequential consistency (SC)*, allows only those executions that produce the same result as if the operations had been executed in an interleaved fashion in a single process[1]. This means that the order of operations executed by a process is strictly defined by the program (the code) it executes. The SC model does require the write to a shared variable performed in one process to become visible by all other processes *instantly*. This means that each process writes directly to the shared memory, without local buffering. Another important requirement of the SC memory model is that it forbids memory operations reordering within a single process (the order is strictly defined by the program).

The SC model is considered to be a *strong memory model* in the sense that it provides firm guarantees regarding the ordering and caused effect of memory operations. Different relaxations of this model lead to the class of *weak memory models (WMMs)* that specify how processes interact through shared memory, when a write becomes visible to other processes and what value a read can return. Thus, WMMs serve as set of guarantees made by designers of execution environment (hardware, programming language, compiler, database, operation system, etc.) to programmers on which behaviours of their concurrent code they may rely on.

[1] In order to prescind from the implementation details while discussing the memory models theory, we avoid the use of the software-specific term *thread* and the hardware-specific term *processor*. Instead, we adhere the terminology of the theory of concurrency employed by Ben-Ari [Ben06] by naming a concurrent piece of code the *process*.

1.2 Related work

Research on weak memory models firstly aims to *formalise* a formal approach of understanding programs with respect to weak memory models which is *systematic*, *sound* and *complete*. Perhaps the most well-known framework for weak memory model-aware analysis was formalised by J. Alglave in 2010 [Alg10]. In addition to developing rather theoretical basis, researchers work on extracting the WMMs for hardware architectures from existing implementations of from their specifications, which are written in natural language and thus suffer from ambiguities and incompleteness. Over the last decade the memory models have been defined for most mainstream multiprocessor architectures, such as x86-TSO and Sparc-TSO (for *Total Store Order*) model for x86 and Sparc architecture formalised in 2009 [OSS09], much more relaxed memory model for Power and ARM architectures [AFI+09; SSA+11; AMT14], and others. There are projects for validating hardware architectures wrt. a memory model, e.g. [LPM14; LSM+16].

Most modern high-level programming languages rely on relaxed memory model as well. Thus, the memory model for Java is based on the *happens-before* principle [Lam78], it was introduced in J2SE 5.0 in 2004 [MPA05]; the C++11 standard [ISO12] has introduced the set of hardware-independent synchronisation fences and atomic operations. The C++17 memory model [BOS+11] is based on the relation *strongly happens-before*. Weak memory are being formalised for even more abstract software environments, the notable project in this area is the project on formalising the Linux kernel memory model, which is being actively developing these days [AMM+18; MAM+17].

Furthermore, there exists a wide range of tools that perform memory model-aware analysis.

- A state-of-the-art tool is `diy` (*do it yourself*), developed by researchers from INRIA institute, France and University of Cambridge, UK. The `diy`[2] is a software suite for designing and testing weak memory models. It is firstly released back in 2010, and since that time it remained to be the only tool for testing weak memory models. The `diy` consists of several modules: the litmus tests generators `diy`, `diycross` and `diyone`, the litmus test concrete executor `litmus` that runs tests on a physical machine and collects its behaviours, and the weak memory model

[2]The `diy` project web site: `http://diy.inria.fr/`

simulator herd that implements reachability analysis for exploring states reachable under the specified WMM.

- There exist tools that perform the weak memory model-aware program verification and model checking. The notable examples are the stateless model checkers RCMC [KLS+17], CHESS [MQ08] and Nidhugg [AAA+17], the tool Trencher for checking programs against the TSO memory model [BDM13], the tool Porthos for portability analysis [PFH+17a],
- Some tools tackle the problem of automated synthesis of the synchronisation primitives, such as the automatic fence insertion tool musketeer [AKN+14], and the automatic verification and fence inference tool blender [KVY11].
- Some other tools perform static instrumentation of concurrent C programs and encode the WMM into the program representation so that it can be model-checked by standard tools. The examples are the instrumenting compiler goto-cc which is a part of CBMC model checker [KT14], the program transformation tool [AKN+13], the tool Weak2SC for producing program descriptions which can be fed into standard model checking tools (such as SPIN [Hol97] or NuSMV [CCG+00]) for performing memory model-aware analysis [TW16].

All the tools listed above consider only a single memory model, however, in real life we face serious engineering problems involving necessity to model more than one execution environment. One of these problems is the *portability* of the program from one hardware architecture to another. A program written in a high-level language is then compiled for different hardware. Even if all the compiler optimisations were disabled (which is rare case nowadays), the behaviour of two compiled versions of the same program may differ due to differences between hardware memory models. As the result, a program compiled under the platforms T can reach states that are unreachable on the platform S, which is a *portability bug* from the source platform S to the target platform T [PFH+17a].

The very first tool that performs the WMM-aware portability analysis is Porthos[3] introduced in April 2017 [PFH+17b]. This tool reduces described problem to a bounded reachability problem, which can be solved with help of an SMT-solver. This approach allows to capture symbolically the semantics of analysing program and both weak memory models into a single

[3]The Porthos project web site: http://github.com/hernanponcedeleon/PORTHOS

SMT-formula, augmented by the reachability assertion. As most modern SMT-solvers are efficient enough to be able to operate the state space of size millions of variables bounded by millions of constraints ([MZ09]), the used method can be applicable in solving the real-world problems.

1.3 Task statement

Current work aims to rework the proof-of-concept tool `Porthos` by extending the input language, which currently represents the minimum subset of C, and revising the general architecture of the tool in order to enhance performance, extensibility, reliability and maintainability. As the general architecture and almost all components of `Porthos` have been redesigned, the tool received a new name – `PorthosC`[4] . Considering the enhancements of the architecture, `PorthosC` represents a generalised framework for SMT-based memory model-aware analysis, which can not only perform the portability analysis, but can serve as a basis for other kinds of static code analysis.

1.4 Thesis structure

The thesis is organised as following. Chapter 2 gives a general view on the weak memory model-aware analysis. Chapter ...

[4]Hereinafter with the name '`Porthos`' we refer to the tool `Porthos` version 1, whereas the new version of `Porthos` is called `PorthosC`.

Chapter 2

Memory model-aware analysis

In general, analysis of concurrent programs with respect to axiomatic memory models is performed in several stages. Firstly, the control-flow and data-flow of a program is encoded as the set of possible *candidate executions*. Obtained model of the program describes the anarchic semantics, which is a truly parallel semantics with no global time that describes all possible computations with all possible communications [ACM16]. Thereafter, the anarchic semantics is constrained by the *weak memory model* specification which is a set of axiomatic constraints for filtering out executions inconsistent in particular architecture.

2.1 The event-based program representation

The classical approach for modeling concurrent programs is to use the *global time*, a single order of interleavings among all events happened in different threads. Although these models are easy to understand, it may be impossible to treat *all* possible states, number of which is exponentially large. However, there exist equivalence classes such that the result of execution different interleavings from single equivalence class is the same (for instance, computations performed by a processor locally do not affect the global state). One such model is the *event-based* representation of a program, which models the program as a directed graph of events (the *event-flow graph*). The event-based model is *non-deterministic*, which The vertices of such a graph represent *events* (see Section 2.1.1), and edges represent *basic relations* (see Section 2.1.2).

2.1.1 Events

An event is a fact of executing the low-level primitive operation such as memory access, threads synchronisation, computation, etc.

A *memory event* $e_m \in \mathbb{E}$ represents the fact of access to the memory. Only memory events change the state of an abstract machine executing the code, since it is completely determined by values stored in its memory. Since memory is the crucial low-level resource shared by multiple processes, most relations are defined over memory events. The processes can access a shared memory location (denoted by l_i, for *location*), or a local one (denoted by r_i, for *register*). A memory event can access at most one shared memory location, high-level instructions that address more than one shared variable must be transformed into a sequence of events. A memory event is specified by its direction with respect to the shared variable, its location $\mathtt{loc}(e_m)$, its processor label $\mathtt{proc}(e_m)$, and a unique event label $\mathtt{id}(e_m)$ [Alg10].

The set of memory events $\mathbb{M}$ is divided into write events $\mathbb{W}$ (that write values to shared-memory locations) and read events $\mathbb{R}$ (that read values stored in shared-memory locations). We add a restriction that each memory event uses at most one shared location, so that the write instruction $i = write(l_1, l_2)$, that encodes the write from the shared location l_2 to the shared location l_1, is represented as two consequent events $e_1 = \mathtt{load}(r_1 \leftarrow l_2)$; $e_2 = \mathtt{store}(l_1 \leftarrow r_1)$. Also, it is important to separate the set of initial write events $\mathbb{IW} \subset \mathbb{W}$ that perform initialisation of program variables.

A *computation event* $e_c \in \mathbb{C} \subseteq \mathbb{E}$, represents a low-level assembly computation operation performed solely on local-memory arguments. An example of computation event may be the event $e_c = r_1 \leftarrow add(r_2, 1)$ that writes the sum of values stored in register r_2 and constant 1 (which is modelled as a register as well) to the register r_1. For modelling branching statements, we distinguish the set $\mathbb{C}_! \subseteq \mathbb{C}$ of *predicative* computation events (also called as *branching events*), that are evaluated as a boolean value.

The synchronisation instructions (fences) cause the *barrier events*, that do not perform any computation or memory value transfer, instead, they add new relations to the program model that restrict the set of allowed behaviours. Functionally, a fence may be a synchronisation barrier or a instruction of flushing the local memory caches, etc.

2.1.2 Relations

The relation $r \subseteq \mathbb{E} \times \mathbb{E}$ is a set of pairs of events (a subset of Cartesian product of two sets of events). There are two kinds of relations between events: *basic relations* that capture semantics of the program, and *derived relations* that are defined from the basic relations and events in the weak memory model specification. Constraints over relations that are specified by weak memory models are defined as requirements of acyclicity, irreflexivity or emptiness of specific relations [ACM16].

The basic relations are the following [Alg10]:

- The *control-flow* of a program is defined by the *program-order* relation $\mathtt{po} \subset \mathbb{E} \times \mathbb{E}$, which represents the total order of events of same process. For instance, if the instruction i_1 generates the event e_1 and the instruction i_2 follows i_1 and generates the event e_2, then $e_1 \overset{\mathtt{po}}{\rightarrow} e_2$.
- The *data-flow* of a program is defined by *communication relations*:
  - the *read-from* relation $\mathtt{rf} \subset \mathbb{W} \times \mathbb{R}$ that maps each write event to the read event that reads the value written by write event;
  - the *coherence order* relation $\mathtt{co} \subset \mathbb{W} \times \mathbb{W}$ that defines the total order on writes to the same location across all processes (also called the *write serialisation*, ws-relation);
- Events from the same process are related by the *scope relation* $\mathtt{sr} \subset \mathbb{E} \times \mathbb{E}$. In contrast to the herd tool, the PorthosC does not use hierarchy of scopes (depicted as the scope tree); instead, it uses simple lables that indicate which process has produced certain event.

Below we enumerate some derived relations [Alg10]:

- the *from-read* relation $\mathtt{fr} \subset \mathbb{R} \times \mathbb{W}$ that maps a read event to all write events preceding the write event from which the read event gets its value:
  $r \overset{\mathtt{fr}}{\rightarrow} w \triangleq (\exists w'.w' \overset{\mathtt{rf}}{\rightarrow} r \wedge w' \overset{\mathtt{co}}{\rightarrow} w)$
- the *communication* relation po over memory events, that fully describes the data-flow of a program:
  $m_1 \overset{\mathtt{com}}{\rightarrow} m_2 \triangleq ((m_1 \overset{\mathtt{rf}}{\rightarrow} m_2) \vee (m_1 \overset{\mathtt{co}}{\rightarrow} m_2) \vee (m_1 \overset{\mathtt{fr}}{\rightarrow} m_2))$

- the *external* (and *internal*) *read-from* relations that restrict the `rf`-relation to the different (respectively, same) processes:
  $w \xrightarrow{\mathtt{fre}} r \triangleq (w \xrightarrow{\mathtt{fr}} r \wedge \mathtt{proc}(w) = \mathtt{proc}(w))$
  $w \xrightarrow{\mathtt{fri}} r \triangleq (w \xrightarrow{\mathtt{fr}} r \wedge \mathtt{proc}(w) \neq \mathtt{proc}(w))$

- the `po-loc` relation that is the `po`-relation over events that access to the same shared variable:
  $m_1 \xrightarrow{\mathtt{po-loc}} m_2 \triangleq (m_1 \xrightarrow{\mathtt{po}} m_2 \wedge \mathtt{loc}(m_1) = \mathtt{loc}(m_2))$

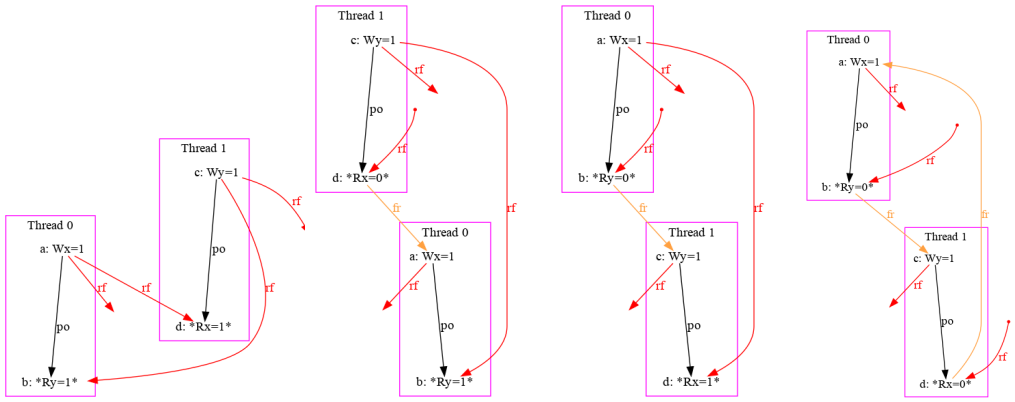
- the semantics of *fences* (memory barriers) specific for different architectures may be defined as derived relations.

2.1.3 Executions

The semantics of a concurrent program is represented as the set of allowed executions. The The *execution* is a path in the event-flow graph defined by `po`- and `rf`-relations and set of final writes to a given memory location that is valid under certain memory model [AMT14]. It can be interpreted as a sequence of guesses which event is to be executed next. The *candidate execution* is an execution that is not yet constrained by a memory model.

Figure 2.1 illustrates four possible candidate executions for the litmus test Example 1.1 (the pictures are generated by the `herd7` tool, version 7.47). Since there are no conditional jumps, the `po`-relation is defined and we do not need to guess it. Since each thread performs single write followed by a single read, the `co`-relation is also defined (it relates the initial write event with the write event to the same location).

Thus, there are only four possible executions defined by the choice of `rf`-relation. The candidate executions pictured in Figures 2.1a–2.1c are consistent both under strong memory model SC and under relaxed memory models x86-TSO, Power, ARM, and some others. However, the execution shown in Figure 2.1c is still consistent under relaxed-memory architectures, but it becomes inconsistent under SC architecture as it forbids cycles over $\mathtt{fr} \cup \mathtt{po}$.



(a) *Final state:* `(0:EAX=1 /\ 1:EAX=1)` **(b)** *Final state:* `(0:EAX=1 /\ 1:EAX=0)` **(c)** *Final state:* `(0:EAX=1 /\ 1:EAX=1)` **(d)** *Final state:* `(0:EAX=0 /\ 1:EAX=0)`

Figure 2.1: *Possible candidate executions for the litmus test Example 1.1*

2.2 The cat language

Weak memory models are defined via the `cat` language [ACM16]. This is a domain specific language for describing consistency properties of concurrent programs. The `cat` language combines expressive power of a functional language (it is inspired by OCaml and adopts its types, first-class functions, pattern matching and other features) with types, operations and assertions that are specific for operating with relations and executions. In `cat`, new relations can be defined via the keyword `let` and the following operators over relations [ACM16].

Below we enumerate pre-defined operators over relations and sets of events:

1. *Unary relations*:
   - *the complement* of a relation `r` is `~r`
   - *the transitive closure* of a relation `r` is `r+`
   - *the reflexive closure* of a relation `r` is `r?`
   - *the reflexive-transitive closure* of a relation `r` is `r*`
   - *the inverse* of a relation `r` is `r^-1`
2. *Binary relations:*
   - *the union* of two relations `r1` and `r2` is `r1 | r2`,

- *the intersection* of two relations `r1` and `r2` is `r1 & r2`,
- *the difference* of two relations `r1` and `r2` is `r1 \ r2`,
- *the sequence* of two relations `r1` and `r2` is `r1;r2`, which is defined as the set of pairs `(x,y)` such that there exists an intervening `z`, such that `(x,z)` $\in$ `r1` and `(z,y)` $\in$ `r2`.

For instance, the `fr`-relation is defined as a sequence of inverted `rf`-relation and `co`-relation: $\texttt{fr} = (\texttt{rf}^{-1}; \texttt{co})$. As an example memory model definition in `cat` language, the the x86-TSO can be found in Appendix A.3. This memory model asserts acyclicity of communication relation, `po-loc` relation, `mfence` relation and some other derived relations [OSS09].

Chapter 3

Portability analysis as an SMT problem

As it has been discussed in Chapter 1, the program may behave differently when compiled for different parallel hardware architectures. This may cause the portability bugs, the behaviour that is allowed under one architecture and forbidden under another. In this Chapter, we describe the general task of analysing the concurrent software portability as a *bounded reachability* problem, which in turn can be reduced to a SAT problem [PFH+17a] (more precisely, to an SMT-problem).

3.1 Model checking and reachability analysis

The model checking is the problem of verifying the system (the model) against the set of constraints (the specification). As the state machine model is the most widespread mathematical model of computation, most classical model checking algorithms explore the state space of a system in order to find states that violate the specification. The general schema of model checking is the following: firstly, the analysing system is being represented as a transition system, a finite directed graph with labeled nodes representing states of the system such that each state corresponds to the unique subset of atomic propositions, that characterise the behavioral properties of each state. Then, the system constraints are being defined in terms of a modal temporal logic with respect to the atomic propositions. Commonly, the Linear Temporal Logic (LTL) or Computational Tree Logic (CTL), along with their extensions, are used as a specification language due to the expressiveness and verifiability of their statements. In the described schema,

the model checking problem is reducible to the reachability analysis, an iterative process of a systematic exhaustive search in the state space. This approach is called *unbounded model checking (UMC)*.

However, all model checking techniques are exposed to the *state explosion problem* as the size of the state space grows exponentially with respect to the number of state variables used by the system (its size). In case of modeling concurrent systems, this problem becomes much more considerable due to exponential number of possible interleavings of states. Therefore, the research in model checking over past 40 years was aimed at tackling the state explosion problem, mostly by optimising search space, search strategy or basic data structures of existing algorithms.

One of the first technique that optimises the search space considerably major was the symbolic model checking with binary decision diagrams (BDDs). Instead of by processing each state individually, in this approach the set of states is represented by the BDD, efficient data structure for performing operations on large boolean formulas [CKN+12]. The BDD representation can be linear of size of variables it encodes if the ordering of variables is optimal, otherwise the size of BDD is exponential. The problem of finding such an optimal ordering is known as NP-complete problem, which makes this approach inapplicable in some cases.

The other idea is to use satisfiability solvers for symbolic exploration of state space [CBR+01]. In this approach, the state space exploration consists of sequence of queries to the SAT-solver, represented as boolean formulas that encode the constraints of the model and the finite path to a state in the corresponding transition system. Due to the SAT-solver. This technique is called *bounded model checking (BMC)*, because the search process is being repeated up to user-defined bound k, which may result to incomplete analysis in general case. However, there exist numerous techniques for making BMC complete for finite-state systems (e.g., [Sht00]).

3.2 Portability analysis as a bounded reachability problem

In general, a BMC problem aims to examine the reachability of the "undesirable" states of a finite-state system. Let $\vec{x} = (x_1, x_2, ..., x_n)$ be a vector of n variables that uniquely distinguishes states of the system; let $Init(\vec{x})$ be an *initial-state predicate* that defines the set of initial states of the system; let $Trans(\vec{x}, \vec{x}')$ be a *transition predicate* that signifies whether there the

transition from state $\vec{x}$ to state $\vec{x}'$ is valid; let $Bad(\vec{x})$ be a *bad-state predicate* that defines the set of undesirable states. Then, the BMC problem, stated as the reachability of the undesirable state withing k steps is formulated as following: $\mathsf{SAT}(Init(\vec{x_0}) \wedge Trans(\vec{x}_0, \vec{x}_1) \wedge ... \wedge Trans(\vec{x}_{k-1}, \vec{x}_k) \wedge Bad(\vec{x}_k))$.

Portability analysis problem may also be stated as a reachability problem, where the undesirable state is the state reachable under the target $\mathcal{M}_\mathcal{T}$ memory model and unreachable under the source memory model $\mathcal{M}_\mathcal{S}$. However, unlikely the BMC problem, the portability analysis does not require to call the SMT-solver repeatedly, since (imperative) programs may be converted as acyclic state graph (by reducing the loops, see Section **??**) and the $Trans$ predicate may be stated only for the final state of a program.

Consider the function $cons_\mathcal{M}(P)$ calculates the set of executions of program P consistent under the memory model $\mathcal{M}$. Then, the program P is called portable from the source architecture (memory model) $\mathcal{M}_\mathcal{S}$ to the target architecture $\mathcal{M}_\mathcal{T}$ if all executions consistent under $\mathcal{M}_\mathcal{T}$ are consistent under $\mathcal{M}_\mathcal{S}$ [PFH+17a]:

Definition 3.2.1 (Portability)**.** Let $\mathcal{M}_\mathcal{S}$, $\mathcal{M}_\mathcal{T}$ be two weak memory models. A program P is portable from $\mathcal{M}_\mathcal{S}$ to $\mathcal{M}_\mathcal{T}$ if $cons_{\mathcal{M}_\mathcal{T}}(P) \subseteq cons_{\mathcal{M}_\mathcal{S}}(P)$

Note that the definition of portability requirements against *executions* is strong enough, as it implies the portability against *states* (the *state-portability*) [PFH+17b]. The result SMT-formula ϕ that encodes the portability problem should contain both encodings of control-flow ϕ_{CF} and data-flow ϕ_{DF} of the program, and assertions of both memory models: $\phi = \phi_{CF} \wedge \phi_{DF} \wedge \phi_{\mathcal{M}_\mathcal{T}} \wedge \phi_{\neg\mathcal{M}_\mathcal{S}}$. If the formula is satisfiable, there exist a portability bug.

3.2.1 Encoding for the control-flow

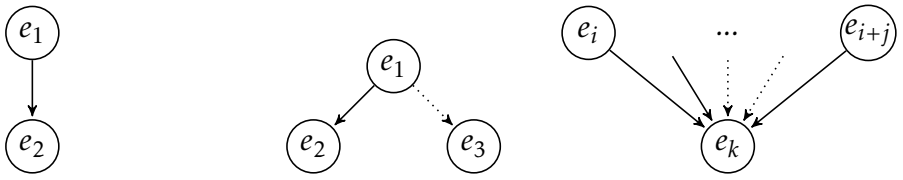
The control-flow of a program is represented in the *control-flow graph*, a directed acyclic connected graph with single source and multiple sink nodes, obtained by the *loop unrolling* (see Section **??**).In control-flow graph, there are two types of transitions (edges): *primary transitions* that denote unconditional jumps or if-true-transitions (pictured with solid lines), and *alternative transitions* that denote if-false-transitions (pictured with dotted lines). Each node on graph can have either one successor (primary) or two successors (both primary and alternative); only computation events can serve as a branching point). However, each merge node can have any

positive number of predecessors, where each edge may be either primary or alternative.

While working on the PorthosC, we applied some modifications of the encoding scheme for the control-flow. The changes are conditioned by the need to be able to process an arbitrary control-flow produced by conditional and unconditional jumps of C language. For that, we compile the recursive abstract syntax tree (AST) of the parsed C-code to the plain (non-recursive) event-flow graph. We show that the new encoding is smaller than the old one used in Porthos since it does not produces new variables for each high-level statement of the input language. For instance, Porthos uses the encoding scheme where the control-flow of the sequential instruction $i_1 = i_2; i_3$ was encoded as $\phi_{CF}(i_2; i_3) = (cf_{i_1} \Leftrightarrow (cf_{i_2} \wedge cf_{i_3})) \wedge \phi_{CF}(i_2) \wedge \phi_{CF}(i_3)$, and control-flow of the branching instruction $i_1 = (c\ ?\ i_2\ :\ i_3)$ was encoded as $\phi_{CF}(\ c\ ?\ i_2\ :\ i_3) = (cf_{i_1} \Leftrightarrow (cf_{i_2} \vee cf_{i_3})) \wedge \phi_{CF}(i_2) \wedge \phi_{CF}(i_3)$ (here we used the notation of C-like ternary operator '`x?y:z`' for defining the conditional expression '`if` x `then` y `else` z'). In contrast, the new scheme implemented in PorthosC firstly compiles the recursive high-level code into the linear low-level event-based representation, that is then encoded into an SMT-formula. The encoding of branching nodes depends on the *guards*, the value of conditional variable on the branching state, which in turn is encoded as data-flow constraint (see Section 3.2.2).

Let $\mathbf{x} : \mathbb{E} \to \{0, 1\}$ be the predicate that signifies the fact that the event has been e**x**ecuted (and, consequently, has changed the state of the system). Let $\mathbf{v} : \mathbb{C} \to \mathbb{R}$ be the function that returns the value of the computation event (evaluates it) that will be computed once the event is executed (strictly speaking, it retuns the *set* of values determined by the `rf`-relation (see Section 2.1.2 for details on relations). We distinguish the function $\mathbf{v}_p : \mathbb{C}_{\mid} \to \{0, 1\}$ that evaluates the predicative computation event. In the result formula, all symbols $\mathbf{x}(e_i)$ and $\mathbf{v}(e_i)$ are encoded as boolean variables.

Consider the following possible mutual arrangement of nodes in a control-flow graph:



(a) *Sequential events* (`seq`) **(b)** *Conditional branching* (`br`) **(c)** *Branch merging* (`mer`)

Figure 3.1: *Linear and non-linear cases of control-flow graph*

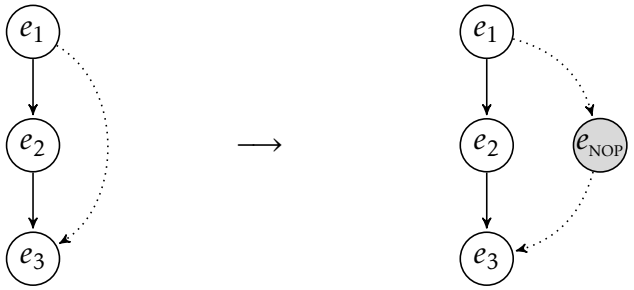


Figure 3.2: *Transformation of the empty-branch nonlinear control-flow*

For listed cases, below we propose the encoding scheme that uniquely encodes each node of graph and allows to encode partially executed program. Equation 3.1 encodes the sequential control-flow represented in Figure 3.1a and reflects the fact that the event e_2 can be executed iff the event e_1 has been executed. Equation 3.2 encodes the branching control-flow depicted in Figure 3.1b by allowing only following executions: $\{\emptyset, (e_1), (e_1 \rightarrow e_2), (e_1 \rightarrow e_3)\}$. In encoding 3.3 of the merge-point represented in Figure 3.1c, the event e_k is executed if either of its predecessors was executed, regardless of type of the transition.

$$\phi_{CF_{seq}} = \mathbf{x}(e_2) \rightarrow \mathbf{x}(e_1) \tag{3.1}$$

$$\begin{aligned}\phi_{CF_{br}} = {} & [\mathbf{x}(e_2) \rightarrow \mathbf{x}(e_1)] \wedge [\mathbf{x}(e_3) \rightarrow \mathbf{x}(e_1)] \wedge \\ & [\mathbf{x}(e_2) \rightarrow \mathbf{v}(e_1)] \wedge [\mathbf{x}(e_3) \rightarrow \neg\mathbf{v}(e_1)] \wedge \\ & \neg[\mathbf{x}(e_2) \wedge \mathbf{x}(e_3)]\end{aligned} \tag{3.2}$$

$$\phi_{CF_{mer}} = \mathbf{x}(e_k) \rightarrow \Big(\bigvee_{e_p \in \mathtt{pred}(e_k)} \mathbf{x}(e_p) \Big) \tag{3.3}$$

For sake of encoding correctness, we require all branches to have at least one event. Thus, for branching statements that do not have any events in one of the branches (such a branch represents a conditional jump forward), we add the synthetic nop-event as it is shown in Figure 3.2:

3.2.2 Encoding for the data-flow

To encode the data-flow constraints, we use the *static single-assignment (SSA) form* in order to be able to capture an arbitrary data-flow into a single SMT-formula. The SSA form requires each variable to be assigned

only once within entire program. In contrast, `Porthos` used the dynamic single-assignment (DSA) form, that requires indices to be unique within a branch. Although the number of variable references (each of which is encoded as unique SMT-variable) on average is logarithmically less in case of the DSA form than the SSA form, the result SMT-formula still needs to be complemented by same number of equality assertions when encoding the data-flow in merge points [PFH+17a].

Following [PFH+17b], the indexed references of variables are computed in accordance with the following rules: (1) any access to a shared variable (both read and write) increments its SSA-index; (2) only writes to a local variable increment its SSA-index (reads preserve indices); (3) no access to a constant variable or computed (evaluated) expression changes their SSA-index. These rules determine the following encoding of load, store and computation events within single thread:

$$\phi_{DF_{e=\mathtt{load}(r\leftarrow l)}} = \mathbf{x}(e) \rightarrow (r_{i+1} = l_{i+1}) \tag{3.4}$$

$$\phi_{DF_{e=\mathtt{store}(l\leftarrow r)}} = \mathbf{x}(e) \rightarrow (l_{i+1} = r_i) \tag{3.5}$$

$$\phi_{DF_{e=\mathtt{eval}(\ldots)}} = \mathbf{x}(e) \rightarrow \mathbf{v}(e) \tag{3.6}$$

To convert the program into SSA form, for each event each variable that is declared so far (either local or shared) is mapped to its indexed reference; this information is stored in the SSA-map "event *to* variable *to* SSA-index". The SSA-map is computed iteratively while traversing the event-flow graph in topological order as it is described in Algorithm 1.

Algorithm 1 Algorithm for computing the SSA-indices

Input: The event-flow graph $G = \langle N, E \rangle$ where V is the set of nodes (events), E is the set of control-flow transitions, e_0 is the entry node
Output: The SSA-map of the form "{ event : { variable : index }}"

```
1: function COMPUTE-SSA-MAP(G)
2:     S ← empty map; S[e_0] ← empty map
3:     for each event e_i ∈ G.N in topological order do
4:         for each predecessor e_j ∈ pred(e_i) do
5:             S[e_i] ← copy(S[e_j])
6:             for each variable v_k ∈ set of variables accessed by e_i do
7:                 S[e_i][v_k] ← max(S[e_i][v_k], S[e_j][v_k])
8:                 if need to update the index of v_k then        ▷ cases (1)-(2)
9:                     S[e_i][v_k] ← S[e_i][v_k] + 1
```

The time of described algorithm is linear of the size of event-flow graph since it performs only single traverse of the graph.

As it has been described before, the `rf`-relation links data-flow between events of data-flow stored in equivalence assertions over the SSA-variables. The encoding of this linkage left untouched as it is implemented in `Porthos`: for each pair of events e_1 and e_2 linked by the `rf`-relation, we add the following constraint:

$$\phi_{DF_{mem}}(e_1, e_2) = \mathtt{rf}(e_1, e_2) \rightarrow (l_i = l_j) \tag{3.7}$$

where the variable of location l is mapped to the SSA-variable l_i for event e_1, and to the SSA-variable l_j for event e_2; and the predicate $\mathtt{rf}(e_1, e_2)$ is encoded as a boolean variable, which itself equals $true$ if e_2 reads the shared variable that was written in e_1.

3.2.3 Encoding for the memory model

todo

Chapter 4

The PorthosC: implementation

The main call for commencing the work on PorthosC was the need for processing real-world C programs, which, at first, requires the input language to be extended. This implies the support not only for new syntactic structures of C language (such as the `switch` statement or the postfix increment operator `i++`), but also for its fundamental concepts and features (such as types, pointer arithmetic or first-order functions), which requires revision of the whole architecture of the tool. Yet far from all of the C language has been supported (that, considering its complexity and numerous pitfalls, goes far beyond current thesis[1]), we consider the accomplished work as a step towards this.

4.1 General principles

The existing implementation of Porthos does not distinguish the event-based program model from the high-level AST, they both are encoded into single SMT-formula (see classes of package '`dartagnan.program`' of Porthostool). Moreover, the syntax tree was implemented as a mutable data structure, which is being modified at all stages of the program (for instance, see the methods '`dartagnan.program.Program.compile(...)`' of Porthos that recursively compute some properties of the AST and change its state). We are inclined to consider this architecture as one that is fast to develop, but hard to maintain (since it is difficult to guarantee the correctness of the program) and extend (since adding the support for a new high-level

[1] To ensure that, we have merely to look at existing C compilers, for instance, the open-source gcc compiler, which uses the C parser written in more than 18.5 thousand lines of code (see `https://github.com/gcc-mirror/gcc/blob/master/gcc/c/c-parser.c`)

instruction requires changing multiple components of the program, from parser to encoder).

Therefore, while working on the new design of PorthosC, we decided to clearly separate the high-level intermediate code representation (implemented as a recursive AST structure) from low-level event-based representation (implemented as an event-flow graph). Such a modular architecture allows to support multiple input languages by parsing them and converting parsed syntax trees to a simplified AST. which, along with all other data-transfer objects (DTO), must be immutable, so that it is possible to guarantee the correctness of the program by controlling its invariants. The immutability in PorthosC is implemented via `final` fields that are assigned by the immutable-object values (either a primitive type, or another immutable object, or an immutable collection provided by the library Guava by Google[2]).

During the development of PorthosC we mainly followed the *KISS principle*, which can be exhaustively described in 17 Unix Rules of Eric Raymond [Ray03].The following list summarises the main rules we followed during the development of PorthosC:

1. *Robustness:*
   1.1. prohibit under-approximating abstractions;
   1.2. preservation the completeness of analysis;
   1.3. modular architecture: each module can be tested independently;
   1.4. usage of software design patterns where necessary;
   1.5. usage of immutable data structures for all DTOs;
2. *Transparency:*
   2.1. following the principles of simplicity and readability;
   2.2. clear and informative program output;
3. *Efficiency:*
   3.1. keeping the trade-off between execution time and memory usage;
4. *Extensibility:*
   4.1. clear modular architecture.

The robustness of analysis is the main criterion of PorthosC as a verification of a bug-hunting tool. This implies modelling the system with only

[2]Guava project repository: `https://github.com/google/guava/`

over-approximating abstractions, which, however makes it more vulnerable to the combinatorial growth of the state space size, preserve the completeness of analysis. As a robust and transparent tool, the PorthosC must adhere to the strategy of aborting its work on any unexpected state (for instance, if a parser failed to parse the string and recovery algorithm is not described).

As its predecessor, PorthosC uses the open-source SMT-solver Z3[3]by Microsoft Research [DB08]. However, unlikely its predecessor, the PorthosC has an additional abstraction level Z-formula (see Section 4.2.2.4) that allow to use any other SMT-solver.

The programming language choice for PorthosC was also made in favour of *java*, firstly, in order to be able to reuse some parts and concepts of Porthos that is written in java, and secondly, because the authors find the object-oriented (OOP) concepts of java suitable for modelling languages. Although java does not show best results in performance benchmarks (for example, comparing to C++ [Hun11; Oak14]), the performance cornerstone of PorthosC (as well as any other SMT-based code analyser) is the phase of solving the SMT-formula, which is left to the third-party SMT-solverinvoked from PorthosC via java API. However, considering the perspective of using PorthosC as a static analyser for real-world programs, the memory optimisation problem must also be taken into account during both encoding and solving stages. It is worth noting that, for the reasons of simplicity, the PorthosC is not a concurrent program, however, we believe that, due to its modular architecture, it can be easily parallelised on the level of program modules.

4.2 Architecture

The general architecture scheme of PorthosC is presented in Figure 4.1. The rectangles in the picture denote processing units (marked with gear sign with a unique number of the component), and labels of the arrows describe the internal representation carried out during the transformation.

The program takes as input the program to be analysed and one (the reachability analysis mode) or two (the portability analysis mode) memory models. The parsed program syntax tree is then converted (Section 4.2.3.3)

[3]The Z3 project repository: https://github.com/Z3Prover/z3

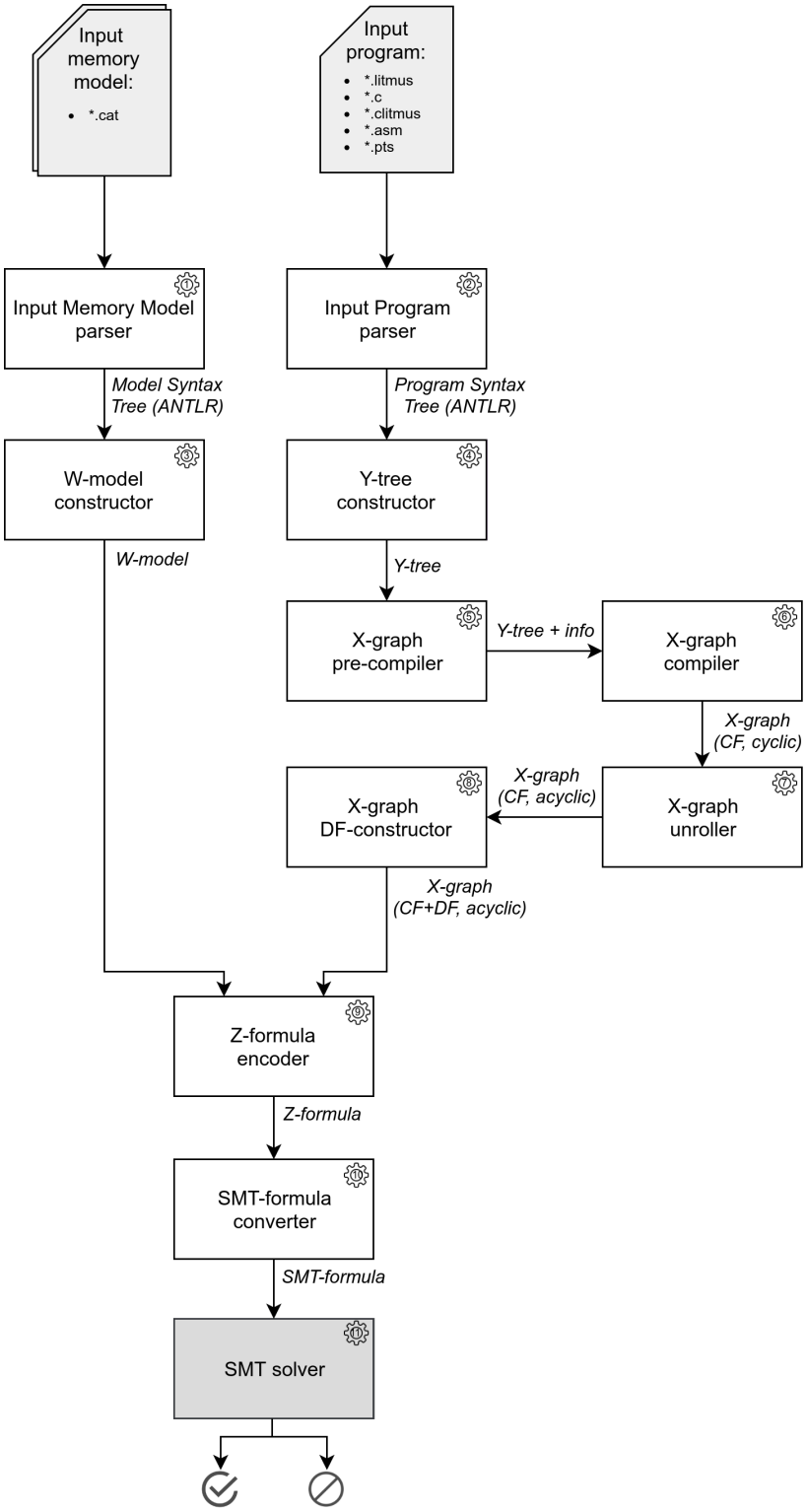


Figure 4.1: *The general architecture of* PorthosC

to a program AST called Y-tree[4](Section 4.2.2.1), which then is being preprocessed at the pre-compilation stage (Section 4.2.3.4) in order to collect information necessary for the compilation. The Y-tree then is being compiled (Section 4.2.3.5) to an X-graph representation (Section 4.2.2.2). The compiled X-graph then is being converted to an acyclic form (Section 4.2.3.6) in order to be encoded into a Z-formula (Section 4.2.2.4). Apart from that, the memory-model constructor (Section 4.2.3.2) constructs the abstract syntax tree of derived relations of the weak memory model W-model (Section 4.2.2.3). Thereafter, W-model and acyclic X-graph are encoded (Section 4.2.3.8) to a Z-formula representation (a wrapper over an SMT-formula), which then is translated to an SMT-formula (Section 4.2.3.9), which then is solved by the SMT-solver (Section 4.2.3.10).

4.2.1 Program input

Both Porthos and PorthosC use the ANTLR parser generator[5] [Par13], a powerful language processing tool. The ANTLR takes as input the user-defined grammar of the target language in a BNF-like form and produces the LL(*)-parser and optionally some auxiliary classes (such as listeners and visitors for the syntax tree). Although this parser may be not as efficient as the hand-written language-optimised parser, it reduces the overhead of implementing the parser significantly. Among other advantages ANTLR, it is worth of noting that it has rather large collection of officially supported grammars. Nonetheless, the intuitive syntax for defining grammars and numerous of tools for debugging grammars make the ANTLR an attractive instrument for solving the parsing problem.

Figure 4.1 represents the simplified grammar of of input language used by previous version of Porthos (the full ANTLR grammar is available at Appendix A.1.

The input language parser used by Porthos suffered from several disadvantages. Firstly, it contained the parser code inlined into directly the grammar, so that the grammar would serve as a template for the parser code

[4]In order to avoid confusion between different internal representations, we prefix the names of elements of each internal representation with a letter. For instance, we picked the letter 'Y' to denote the AST code representation as drawing of this letter resembles the tree branching; with letter 'X' we prefix elements of the event-flow graph as the events are to be executed; and with letter 'W' we prefix elements of the weak memory model AST.

[5]The ANTLR project repository: https://github.com/antlr/antlr4

```
<prog> : <init> <thrd>* <assert>
       ;
<thrd> : thread <tid> <inst>
       ;
<inst> : <atom>
       | <inst> ; <inst>
       | while <pred> <inst>
       | if <pred> { <inst> } <inst>
       ;
<atom> : <reg> <- <expr>
       | <reg> <:- <loc>
       | <loc> := <reg>
       | <reg> = <loc>.load(<atomic>)
       | <loc> = <reg>.store(<atomic>)
       | ('mfence' | 'sync' | 'lwsync' | 'isync')
       ;
<pred> :
       | true
       | false
       | <expr> (and | or) <expr>
       | <expr> ('==' | '!=' | '>' | '>=' | '<' | '>=') <expr>
       ;
<expr> : [0-9]
       | <reg>
       | <expr> ('*' | '+' | '-' | '/' | '%') <expr>
       ;
```

Figure 4.1: *The sketch of the input language grammar used by* Porthos v1

(which is called semantic actions). Such a combining of two rich languages[6] makes the code hardly understandable, and, therefore, poorly maintainable. In PorthosC, we clearly separated the parser (generated from the grammar file '*<grammar>.g4*') from the converting the ANTLR syntax tree to the AST, that is one for all languages of an input program.

Secondly, in Porthos, the semantics of operations was defined syntactically (in ANTLR grammar), whereas this should be performed by a separate module operating on the AST level[7], so that it does not require to change grammar for encoding the semantics of a new function.

As the reader can notice from the grammar sketch in Figure 4.1, the memory operations of different kinds vary syntactically. For example, the assignment of local computation to a register uses the symbol '<-', the atomic non-relaxed load operation denoted as '<:-', non-relaxed store operation denoted as ':=', and the semantics of relaxed `load` and `store` are resolved syntactically by matching the function name. In PorthosC, the semantics of the data-flow operation is determined according to the types of operands, that are determined during the pre-compilation stage (see Section 4.2.3.4). The semantics of the functions also is being resolved

[6] by the term "reach" we mean "expressiveness": at least, the grammar of the language of semantic actions (i.e., java in our case) is Turing-complete.

[7] Most modern languages (including C++) allow the function overloading, thus requiring the type information to be available for the function resolution algorithm (see Section 4.2.3.4.2).

during the pre-compilation stage via the *invocation hooking* mechanism (see Section 4.2.3.5.2).

Thirdly, the grammar used by Porthos had restricted set of allowed operations. For example, it allowed only computations over the local variables, which might lead to inconsistency of the result SMT-formula if the same variable name was used both as a register and as a location, for instance, the in code snippet '`x := reg1; reg2 <- (x + 1);`', the first statement interprets the variable `x` as a location, while the grammar of second statement requires it to be a register. Also, only integers were processed by the input language parser of Porthos. In PorthosC, we extended support for primitive types supported by the Z3 solver (this apply to 32-bit integers encoded as Ints of Z3, floats encoded as Reals, enumerations encoded as Scalars). Although Z3 supports the array theory (characterised by the select-store axioms [MB11]), the complexity of static analysis of pointers and non-constant sized arrays moves the full support of arrays and pointers out of the scope of current thesis.

The minor drawbacks of grammar used by Porthos include lack of operator associativity (expressions of the form `1 + 2 * 3` could not be parsed), incorrectly (in terms of C) implemented grammar rule for the statement (the semicolon punctuator '`;`' was implemented as the separator between two statements, whereas in C it serves as a statement terminator). Also, Porthos supports only the litmus-specific syntax for the variables initialisation, however, it allows only ini tialisation of the shared variable and only by default value `0`. The PorthosC supports arbitrary declaration to be performed in initialisation statement.

The PorthosC uses the C language grammar of proposed in the C11 standard [ISO11], that was extended by litmus test-specific syntax such as initialisation and final-state assertion statements (the original ANTLR grammar can be found in the official repository containing the collection of ANTLR v4 grammars[8]).Current version of PorthosC does not recognise C processor directives (it ignores them), however, in future it can be extended to support them.

Currently, PorthosC can operate only in the intra-procedural analysis mode, assuming that each function defined in the input file is being executed in a separate thread. However, the redesigned architecture of PorthosC allow to easily support the inter-procedural (cross-procedure) analysis by inlining function calls and binding variable contexts. Thus, instead of

[8]Repository path: `https://github.com/antlr/grammars-v4`

analysing a single code file, the user can specify the whole code project. The functions to be executed in parallel can be specified by the user, however, the some concurrent parts of code can be detected automatically at the compilation stage while the semantics of function invocations is being resolved (for instance, the C threads operations from `thread.h` or `pthread.h` may be supported).

4.2.2 The internal representations

For keeping the architecture transparent, we build all abstraction levels with interfaces, even if some of them does not add any new functionality.

4.2.2.1 Y-tree

The first internal representation used by `PorthosC` is the *Y-tree*, which represents a rather high-level AST. The Apendix A.2 represents the file tree of main classes that constitute the Y-tree hierarchy (as the inheritance tree might be obvious for the C-like AST, we confine ourselves to presenting the classes file tree only, which we tend to retain clearly structured).

The abstract syntax tree Y-tree is an abstraction level suitable for compiling the program to a low-level representation (in case of processing low-level assembly code, it may be directly converted to the X-graph representation). As the `ANTLR` syntax tree follows the same structure as the grammar, which is a superset of the real (meaningful) grammar of C language, it lacks multiple concepts of the language (for example, the syntax of indexer access corresponds the grammar rule "`postfixExpression '[' expression ']'`", that is converted to the `YIndexerExpression`). However some details of the syntax might have been abstracted away (for instance, array operations may be emulated by functions invocations, see [Gri12, Chapter 5]), we found this level of abstraction suitable enough for our tasks.

Each Y-tree element implements the interface `YEntity` and carries the `OriginLocation` instance that contains information about the coordinates of the input text that generated the Y-tree element.

Following the C11 standard [ISO12], we distinguish a *statement* (*"an action to be performed"*) from an *expression* (*"a sequence of operators and operands that specifies computation of a value, or that designates an object or a function, or that generates side effects, or that performs a combination thereof"*).

All Y-tree expressions implement the `YExpression` interface. On the Y-tree level, the pointer arithmetic is modelled by the integer number

pointer level of an expression (although in fact this is the property of a type not of an expression, the y-tree is an untyped syntax tree, therefore the elements Y-tree should carry this property). We distinguish the subset of expressions that imply no side-effects, they implement the interface `YAtom` and can be global or local (which is defined also syntactically).

The Y-tree expressions are the following:

- `YBinaryExpression` that model the C binary operator (*relative* operator that compares two expressions of any type, *logical* that processes two boolean expressions, and *numerical* that processes two numerical expressions);
- `YUnaryExpression` that model the C unary expression (logical negation, numeric prefix and postfix increment and decrement, bitwise complement);
- `YMemberAccessExpression` that has an arbitrary expression of type `YExpression` as its base expression (it will be resolved during the compilation stage);
- `YIndexerExpression` and `YInvocationExpression` that as arbitrary expression as its base or arguments (strictly speaking, the indexer expression is an unary-function invocation, but as the SMT-solver we use supports the constant-array theory, we can maintain the array type);
- `YAssignmentExpression` that assigns an `YExpression` to an `YAtom`;
- `YVariableRef` that stores the untyped "reference" to a variable (viz., the name only);
- `YLabeledVariableRef` that represents the litmus-specific local variable reference for a certain the process (e.g., '`P0:x`' which means the local variable `x` of the process `P0`);
- `YParameter` that represents a typed variable (the type was declared, similarly to the variable definition);
- `YConstant` that represents an untyped non-named constant.

Similarly to expressions, all Y-tree statements implement the `YStatement` interface. The statements are the following:

- YBranchingStatement representing the if-then-else statement;
- YLoopStatement representing both while- and for- loops;
- YJumpStatement representing unconditional jump (goto-jump to a label and loop-jumps break and continue);
- YCompoundStatement (block statement) representing sequence of N statements grouped into one syntactic unit;
- YLinearStatement representing a single expression;
- YVariableDeclarationStatement containing the information about the variable type during the variable declaration.

On the Y-level of abstraction, we define the YType as a *reference* for the type (since the Y-tree is not typed, all expressions do not have type, however, the YType is used for storing the information on declaration, including the type itself, type modifiers and qualifiers).

According to the C standard, *"any statement may be preceded by a prefix that declares an identifier as a label name"*. The Y-tree statements of follow this rule, however they these labels are symbolic, and they need to be resolved at the pre-compilation stage. Apart from the set of statements listed before, we define the YFunctionDefinition and its inheritor a litmus-specific declaration YProcessDefinition used in intra-procedural analysis mode. The function definition contains the YCompoundStatement body and the YMethodSignature signature, which is used in the function resolution during the compilation stage. The other litmus-specific statements are YPreludeDefinition that carries the list of YStatement initial writes, and YPostludeDefinition that carries the YExpression binary expression to be asserted by the litmus test.

The syntax tree that contains set of definitions (e.g., litmus-initialisations, function definitions, litmus-asserts) is modelled by the class YSyntaxTree.

4.2.2.2 X-graph

The Y-tree is compiled into the low-level event-based program representation called *X-graph*. The mathematical structure of event-flow graph was discussed in Section 2.1. The nodes of the graph are events, and the edges are basic relations: the control-flow relation po and the data-flow relations co and rf. Hereinafter, we denote the X-graph with only control-flow

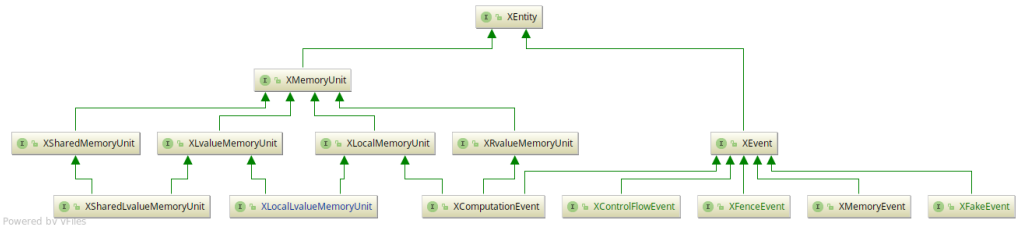


Figure 4.2: *The inheritance tree of interfaces of X-graph*

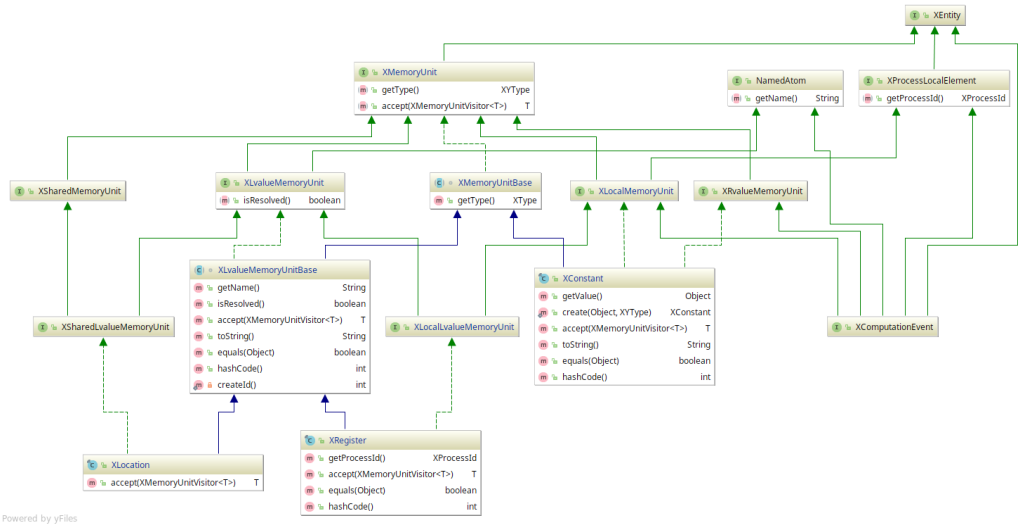


Figure 4.3: *The inheritance tree of X-graph memory units*

edges as X-graph$_{CF}$, the X-graph with only data-flow edges as X-graph$_{CF}$. The complete X-graph is X-graph$_{CF+DF}$ = X-graph$_{CF}$ ∪ X-graph$_{DF}$. Figure 4.2 represents the main hierarchy of the X-abstraction level.

All elements of X-graph implement the interface `XEntity`. There are two main kinds of X-entity: *events* that implement the `XEvent` interface, and *memory-units* that implement the `XMemoryUnit` interface.

4.2.2.2.1 Memory units

The *memory unit* is a memory cell of an abstract machine executing the code. This machine has an infinite number of arbitrary-sized *registers* (local memory units) and *locations* (shared memory units). Local memory units inherit the `XProcessLocalElement` interface, that stores the ID of the owning process. Figure 4.3 represents inheritance hierarchy of memory units.

Following the terminology of the C standard, we distinguish the *r-value* and *l-value* memory units (unlike r-values, the l-values may be assigned a new value). As r-values cannot change their value, they can be seen as the value itself (therefore the `XComputationEvent` is modelled as an local r-value memory unit, see more detailed discussion further in current Section).

Each memory unit has an `XType` associated with it. The X-type is a symbolic representation of the C primitive type[9] that is easily convertible to an SMT-type (modelled as `ZType`).

The memory units are created and stored by the `XMemoryManager`, which provides interface for accessing memory units during compilation stage. For more detailed description of memory management see Section 4.2.3.5.1.

Following the litmus tests format, we distinguish three types of X-graph: one for a process, one for a litmus-initialisation block and one for the assertion statement. However, all three types of X-graph are modelled by the same graph structure `XProcess` with certain restrictions complied by the corresponding type of X-interpreter that constructs the graph. This simplifies drastically the processing of different types of code blocks as they all are modelled by the same data structure. The examples of restrictions on X-graphs are the following: the initialisation block can not have branchings, fence events; the process cannot have assertion events; the assertion block can not have shared-memory events or fence events. We discuss the interpretation of different types of statements in more detail in Section 4.2.3.5.3.

4.2.2.2.2 Events

An event (`XEvent`) represents the fact of executing the primitive operation, which is independent from other events. Each event is specified by the process generated them and a unique event label. This information is stored by events in the immutable structure `XEventInfo`.

The following interfaces model basic kinds of events (see Figure 4.2):

- `XMemoryEvent`
  The memory event defines the transfer of the value from one memory unit to another. There are four types of memory events (the arrow denotes the direction of the data-flow):

[9] Here we should note the `PorthosC` can eventually evolve to be able to analyse programs written in an OOP language (for instance, in C++). In this case, the `XType` will have more complex structure than a simple enumeration, which it has when we need to emulate only primitive types of C language. See more detailed discussion on input language type system in Section 4.2.3.4.2.

  - XRegisterMemoryEvent:
    (XLocalLvalueMemoryUnit) ← (XLocalMemoryUnit),
  - XLoadMemoryEvent:
    (XLocalLvalueMemoryUnit) ← (XSharedMemoryUnit),
  - XStoreMemoryEvent:
    (XSharedLvalueMemoryUnit) ← (XLocalMemoryUnit),
  - XInitialWriteEvent:
    (XLvalueMemoryUnit) ← (XRvalueMemoryUnit).

- XComputationEvent
  We distinguish two types of computation events:
  - XUnaryComputationEvent that encodes bit negation and no-operation,
  - XBinaryComputationEvent that encodes numeric operations (such as addition, multiplication, etc.), bit vector operations (such as bit-and, bit-xor, etc.), relative operations (such as greater-then comparison, equality comparison, etc.), and logical operations (such as conjunction and disjunction).
    The computation event class implements both XEvent and XMemoryUnit. This is a model-level optimisation, which is possible because a computation event performs computation over local-only memory and does not change value of any memory unit. Thus, the *computation* abstraction (as the CPU time spent for the computation itself) can be safely removed from the model, and the computation event can be seen as a zero-time operation that produces the *value*. For analysing large expressions this optimisation is sensible because it considers the whole expression as a single computation event, encoded therefore as a single SMT-variable.
    Note, for the purpose of simplifying the X abstraction level, computation events may have been modelled as invocations of bodiless functions (for instance, the operation ‘x + y’ may be modelled as the invocation of the function ‘+’ with the arguments ‘x’ and ‘y’). However, current version of PorthosC maintains the XComputationEvent as the operators are supported by the SMT-solver.

- `XControlFlowEvent`
  The control-flow event indicates a non-linear jump in the code. We distinguish two kinds of control-flow events:
  - `XJumpEvent` that performs no computation and no data operation, it can be safely removed from the model as an optimisation,
  - `XMethodCallEvent` that models the function call with the `fastcall` calling convention (passing arguments in registers). The function call also implements the `XLocalMemoryUnit` since it represents the computed value returned by the function call. If the function has been resolved, the actual arguments are bind to the formal parameters (treated as temporary local memory units), the called `XMethodCallEvent` is pushed onto the call stack, and the execution jumps to the function body. Each `return` statement creates the assignment of the function call event on the top of call stack.
    If the function has not been resolved, **TODO: think about it. Will the formula be SAT? Or do we need to abort the analysis?**

- `XFenceEvent`
  The fences are implemented as an enumeration `XBarrierEvent`. Current implementation of `PorthosC` supports all fences supported by `Porthos`: `mfence`, `sync`, `optsync`, `lwsync`, `optlwsync`, `ish`, `isb`, and `isync`.

- `XFakeEvent`
  The fake events are auxiliary elements of X-graph.
  - `XEntryEvent`, the per-process unique source event in the event-flow graph,
  - `XExitEvent`, the process sink event. There are two kinds of sink events: *complete* and *incomplete* (see description in Section 4.2.3.6),
  - `XNopEvent`, the no-operation event (a jump to the next event), used for correct encoding in case when the control-flow branch does not have any event (see Figure 3.2).

4.2.2.2.3 Edges

Edges are stored as an immutable hash-map from source to target event. The type of edge is stored in the enumeration `XEdgeKind`:

- the *control-flow edges*: a primary edge (epsilon or if-then transition) and an alternative edge (if-else transition), see more in Section 3.2.1;
- the *data-flow edges*: the `co`-relation edge and the `rf`-relation edge.

4.2.2.2.4 Graph invariants

Once being constructed, the graph must conform the following requirements:

1. graph must have a single source with no ingoing edges, and two sinks of different kinds without outgoing edges,
2. graph must be connected,
3. each node can have either one or two direct control-flow successors,
4. only nodes of type `XComputationEvent` can have two direct control-flow successors,
5. **todo: sth about data-flow edges**

4.2.2.2.5 Graph builder

todo

4.2.2.2.6 Graph example

consider the following code in C:
picture of the CF-graph:

4.2.2.3 W-model

a simple set of recursively defined relations over ..
atoms: sets/basic relations
itemize elements

4.2.2.4 Z-formula

in `Porthos` : ctx as an argument, calling ctx to create new clause. drawbacks: - hard to debug - only one solver - non-safe way to construct formula (if ctx has changed, runtime excpetion)

in `PorthosC` we created the new abstraction layer Z-formula[] that is translated to
recursive representation of SMT-formula
used as an abstraction btween

implemented as a simple wrapper over the z3 formula
typed: Expr, BoolExpr, ArithExpr, ...

4.2.3 The processing units

This section describes program units that construct, transform and analyse internal representations described before. Further in this Section, the subsection titles have a number in parentheses, which correspond to a unique number of the processing unit being discussed. These numbers are pictured within the gear in the top-left corner of each processing unit in Figure 4.1.

the full X-graph construction is performed in three stages. First, the Y-tree is compiled to a *cyclic* control-flow event-based graph X-graph$_{CF}$. Then, this graph is unrolled to an *acyclic* control-flow event-based graph X-graph$^{U}_{CF}$. After that, the compiler is able to perform the data-flow analysis and produce the full event-based graph X-graph$^{U}_{CF+DF}$, which remains to be *CF-acyclic* (no cycles among control-flow edges).

All recursive data structures are processed by units that implement the *visitor* pattern [PJ98]. This is a behavioural pattern that separates the program logic from the object implementation by specifying handling methods for each element of the object. The visitor pattern performs the double-dispatching mechanism: the visiting class defines the method for accepting a visitor, and the visitor itself defines the To visit an instance `x` by a visitor `v`, the program should call the accept-method of `x` and pass `v` as an argument. In our implementation, the visitor must carry out the continuation of traversing the recursive data structure itself. The general structure of the visitor pattern is illustrated by the pseudo-java code in Figure 4.4.

We consider the visitor pattern as the most natural way for operating the hierarchical data structures such as AST. However, we use its double-dispatching capabilities to reduce cost of multiple type casting performed while traversing the elements of a non-recursive data structure. The operator instance, instead of having a single method that handles and element of the instance, extends the visitor interface and splits the handler method into multiple methods, one for each type of element.

For traversing plain (non-recursive) complex data structures, we mostly follow the `iterator` pattern: the special object Iterator, that has access to elements of the data structure, iterates over them and ... . Since in `Porthos` we usually do not need to browse an object elements in different order, the iterator creation is performed by the data structure itself. However, in

```
interface Element {
  <T> T accept(Visitor<T> visitor);
  ...
}

class AnElement implements Element {
  @Override
  <T> T accept(Visitor<T> visitor) {
    return visitor.visit(this);
  }
  ...
}
```

```
class Visitor<T> {
  T visit(AnElement e) {
    // visiting logic
    ...
    // continue recursively
    e.getChild().accept(this);
  }
  T visit(AnOtherElement e) {
    ...
  }
  ...
}
```

Figure 4.4: *Illustration of the visitor pattern*

contrast to the visitor of a recursive data structure, the visitor of a plain data structure should be used with an *iterator* that has access to the elements of the object and passes them to the visitor.

4.2.3.1 Input parsers (1,2)

Both input-language and input-model parsers are implemented via ANTLR parser generator.

Details on the input program language grammar are discussed in Section 4.2.1. Currently, PorthosC does not consider preprocessor instructions (it ignores them). For implementing the inter-procedural mode of Porthos, it is crucial to support inclusion of header files (the #include preprocessor directive). Next step would be implementing the support of macros and conditional compilation directives, that are used often in C code. As preprocessor statements may appear in arbitrary place of a program, the preprocessor must be a stateful processing unit that reads the token stream and dynamically instrument the program by interpreting directives and expanding macros.

The input memory model language cat is discussed in Section 2.2. The ANTLR grammar for the cat language was extracted from the parser used by herd tool[10] written in OCaml. **TODO: add reference to the grammar file in github repository OR include a short version as an appendix**

[10] herd project repository: https://github.com/herd/herdtools7

4.2.3.2 W-model constructor (3)

visitor of antlr tree: almost direct
todo: file tree of w-model to appendix
in perspective: work with functional-style definitions

4.2.3.3 Y-tree constructor (4)

from ANTLR syntax tree of C
- desugar, equiv transform
- variables distinguished from
- if we don't support , our parser still parses it, and the error is thrown at the moment of converting syntax tree to the AST (Y-tree).
- So, The language-dependent syntax tree is converted to the AST by the stateless Visitor (e.g., for C11 -> Ytree conversion is made by 'C2YtreeConverterVisitor') + short structure of this visitor (how?.. need ly?)
- Y-Sytnax error

4.2.3.4 X-graph pre-compiler (5)

The precompiler traverses the Y-tree and collects information necessary for its compilation into an X-graph.

4.2.3.4.1 The label resolution

The label resolution is necessary for establishing links to labelled statements. In C, labelled statements are declared via the colon-syntax '<label> : <statement>', and the labels are referenced by the jump-statement 'goto <label>'. The label resolution algorithm traverses the Y-tree and collects all declared labels into a map that points a label to the labeled statement. This information is used during compilation to set up unconditional jumps.

4.2.3.4.2 Type analysis

The C language has a static (resolved at compile-time) manifest (all types are declared explicitly) type system. Comparing to languages that use type inference, the type analysis of a C program constitutes a simple propagating the type information (obtained from variables declarations) to all expressions. Being carried at Y- and X- representation levels, the type is converted to a Z-type at the stage of the Z-formula encoding (see Section 4.2.3.8).

Currently, PorthosC handles only the primitive C types (such as int, char, float, etc.), which is modelled as an enumeration XType. Array dimension is stored as an integer number at the Y-level (at the X- and Z- levels, **TODO: think about it**). In addition, PorthosC has a built-in extensible database storing the semantics of non-primitive C types stored by the X2ZTypeConverter, which can be requested for converting an X-type to a Z-type.

The type analysis algorithm should consider the type aliases supported by C language (defined by the typedef instruction). For resolving the type aliases, the precompiler should make an extra traverse of the Y-tree before the pre-compilation stage and build up a symbol map.

The type analysis also includes the process of resolving semantics of function invocations. The result is stored in a map of invocation expression to the resolved function signature, so that the compiler can decide either to jump to the function body and interpret it, or to hook the invocation if the semantics of the function is resolved. However, due to lack of polymorphism and function overloading mechanism inherent to OOP languages, the function resolution algorithm may set up the mapping only for the function name and thus may neglect analysis of types of the arguments, we build the mapping for the full function signature so that the type analysis preprocessing unit may be used for analysing the C++ code if needed.

4.2.3.4.3 Variable kind analysis

On the compilation stage, once the compiler meets the reference to a variable, it should know whether it refers to a local or global variable. The kinds of variables have to be determined on the pre-compilation stage.

The following types of variables are detected as *global*:

- a variable was declared as a pointer;
- a variable whose address was accessed by any process;
- a variable that was declared as a parameter of the process function;
- a variable that was exported by the extern keyword (in the kernel-analysis mode, the functions EXPORT_SYMBOL and EXPORT_SYMBOL_GPL also export symbols for dynamic linking);

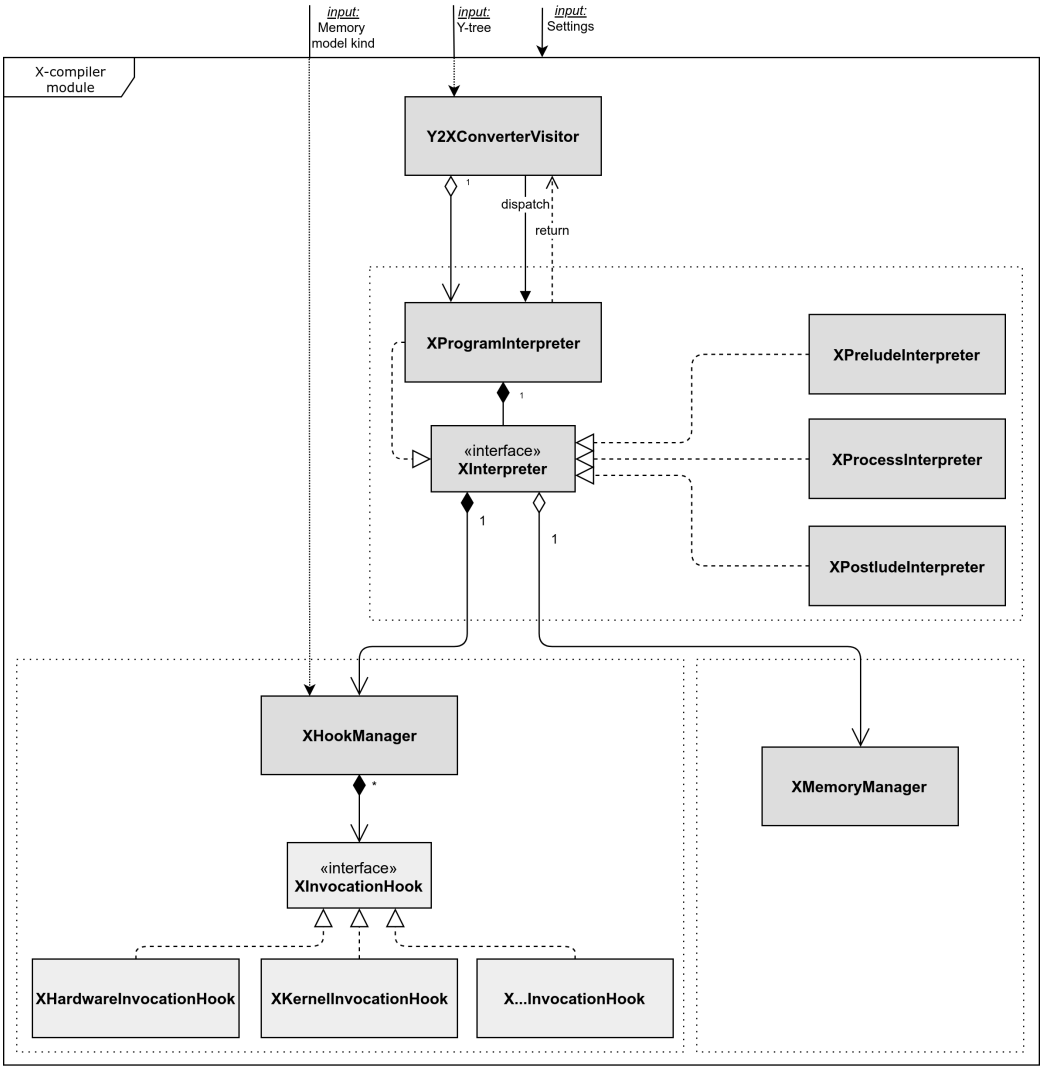


Figure 4.5: *Main components of the X-compilation processing unit*

4.2.3.5 X-graph compiler (6)

The X-compiler is the main component that transforms the recursive Y-tree data structure to the plain X-graph representation.

The X-compiler is a complex processing unit, Figure 4.5 illustrates the relationship between the principal components of the X-compiler in the UML language.

The X-interpreter is a stateful processing unit that carries the X-graph-builder and provides *action* methods for changing its state and thus the state of the builder.

X compilation error
<scheme of who calls whom>

4.2.3.5.1 Memory manager

The X-graph abstract machine model disposes infinite number of memory units, both local and shared (see Section 4.2.2.2.1). The X-compiler accesses all memory units via the `XMemoryManager`, which by the end of pre-compilation stage is already initialised (has registered all shared memory units). However, the memory manager is a stateful component of the compiler as it offers the methods for declaring and removing local memory units dynamically at the compilation time. At each time of the compilation process, the `XMemoryManager` can resolve the memory unit by its name. Following the C standard, local memory units have higher priority over global ones. Since the C language allows the variables that have the same name to be declared in nested contexts, the `XMemoryManager` carries the block-context stack.

<some picture ? >

4.2.3.5.2 Invocation hook manager

The invocation hooking module serves as a knowledge base that stores the semantics of functions. Once the X-compiler meets the function invocation, it calls the `XHookManager`, which tries to match the function signature (in case of program in C language – only the function name) across all signatures it stores. Once the signature matches, the hook manager intercepts the compilation process and interprets the method according to the algorithm it stores. The hook manager operates multiple invocation hooks depending on the user-defined mode (`XInvocationHook`

multiple hooks for mult modes

lambda-hook: knowledge base: InterceptionAction(BiFunction<XMemoryUnit, XMemoryUnit[], ? extends XEntity> action)

todo: kernel hooks

4.2.3.5.3 Interpreter

stateful

stacks: context, readyContext, call, BlockContext (rename): what stores, enum of states

almostReadyContexts: for computation events. Flushed by invocation . When are filled.

setting up edges depending of type of ready-context

- hierarchy of Compilers: pic on inheritance (XCompiler is an stateful abstract machine). Restrictions of lude-compilers postlude: assertion: a new event XAssertionEvent

interfaces that offer compilers: listing of interface

algorithm for interpreting

4.2.3.6 X-graph unroller (7)

up to bound k

recursion – may be, if calls are supported

two kinds of exits

- unrolling: why we cannot encode cyclic structures. reference to the paper (see arXiv version)

simple algorithm

setting up backward edges

how we attempted, why it didn't work in general case:

why we changed the notion of unrolling bound

sth about hashcodes and collections

- After we acquired the event-based representation, we can perform some modifications/simplifications/optimisations on it (separately, allowing user to manage them)

4.2.3.7 X-graph data-flow constructor (8)

- set up co, rf edges => new graph

- computing SSA maps (now: during the encoding. should be: during the post-compilation) (as one of necessary steps before encoding)

4.2.3.8 Z-formula encoder (9)

Z-type : SMT-specific (say about smt-lib)

4.2.3.9 SMT-formula converter (10)

4.2.3.10 SMT-formula translator (11)

4.2.4 Program output

structure: verdict

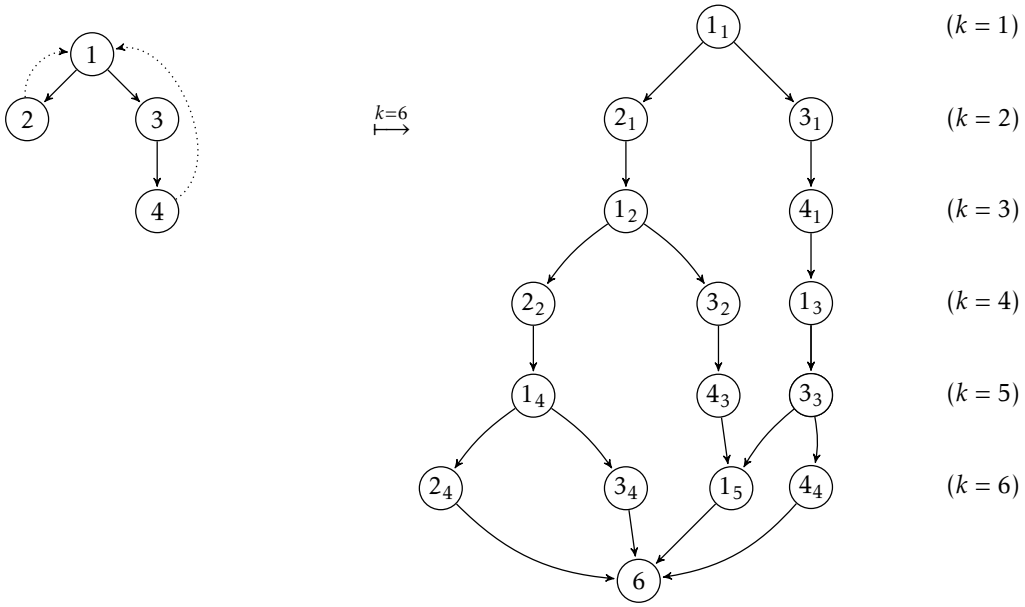


Figure 4.6: *Example of the flow graph from Figure* **??**, *unwinded up to the bound* $k = 6$

4.2.5 Auxiliary components

4.2.5.1 Watchdog timer

todo

4.2.5.2 Logger

todo

Chapter 5

Evaluation

tests

5.1 Comparison with Porthos

5.1.1 Unique Features

5.1.2 Performance

5.2 Comparison with HERD

5.2.1 Unique Features

5.2.2 Performance

Chapter 6

Summary

enumerate goals in past tense + justification

contribution arch – framework new encoding: more laconic (compare!)

kernel?

Bibliography

[AAA+17] Parosh Aziz Abdulla, Stavros Aronis, Mohamed Faouzi Atig, Bengt Jonsson, Carl Leonardsson, et al. "Stateless model checking for TSO and PSO". In: *Acta Informatica* 54.8 (2017), pp. 789–818.

[ACM16] Jade Alglave, Patrick Cousot, and Luc Maranget. "Syntax and semantics of the weak consistency model specification language cat". In: *arXiv preprint arXiv:1608.07531* (2016).

[AFI+09] Jade Alglave, Anthony Fox, Samin Ishtiaq, Magnus O Myreen, Susmit Sarkar, et al. "The semantics of Power and ARM multiprocessor machine code". In: *Proceedings of the 4th workshop on Declarative aspects of multicore programming*. ACM. 2009, pp. 13–24.

[AG96] Sarita V Adve and Kourosh Gharachorloo. "Shared memory consistency models: A tutorial". In: *computer* 29.12 (1996), pp. 66–76.

[AKN+13] Jade Alglave, Daniel Kroening, Vincent Nimal, and Michael Tautschnig. "Software verification for weak memory via program transformation". In: *European Symposium on Programming*. Springer. 2013, pp. 512–532.

[AKN+14] Jade Alglave, Daniel Kroening, Vincent Nimal, and Daniel Poetzl. "Don't sit on the fence". In: *International Conference on Computer Aided Verification*. Springer. 2014, pp. 508–524.

[Alg10] Jade Alglave. "A shared memory poetics". In: *La Thèse de doctorat, L'université Paris Denis Diderot* (2010).

[AMM+18] Jade Alglave, Luc Maranget, Paul E McKenney, Andrea Parri, and Alan Stern. "Frightening small children and disconcerting grown-ups: Concurrency in the Linux kernel". In: *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM. 2018, pp. 405–418.

[AMT14] Jade Alglave, Luc Maranget, and Michael Tautschnig. "Herding cats: Modelling, simulation, testing, and data mining for weak memory". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 36.2 (2014), p. 7.

[BDM13] Ahmed Bouajjani, Egor Derevenetc, and Roland Meyer. "Checking and enforcing robustness against TSO". In: *European Symposium on Programming*. Springer. 2013, pp. 533–553.

[Ben06] Mordechai Ben-Ari. *Principles of concurrent and distributed programming*. Pearson Education, 2006.

[BOS+11] Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. "Mathematizing C++ concurrency". In: *ACM SIGPLAN Notices* 46.1 (2011), pp. 55–66.

[CBR+01] Edmund Clarke, Armin Biere, Richard Raimi, and Yunshan Zhu. "Bounded model checking using satisfiability solving". In: *Formal methods in system design* 19.1 (2001), pp. 7–34.

[CCG+00] Alessandro Cimatti, Edmund Clarke, Fausto Giunchiglia, and Marco Roveri. "NuSMV: a new symbolic model checker". In: *International Journal on Software Tools for Technology Transfer* 2.4 (2000), pp. 410–425.

[CKN+12] Edmund M Clarke, William Klieber, Miloš Nováček, and Paolo Zuliani. "Model checking and the state explosion problem". In: *Tools for Practical Software Verification*. Springer, 2012, pp. 1–30.

[DB08] Leonardo De Moura and Nikolaj Bjørner. "Z3: An efficient SMT solver". In: *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer. 2008, pp. 337–340.

[Gri12] David Gries. *The science of programming*. Springer Science & Business Media, 2012.

[Hol97] Gerard J. Holzmann. "The model checker SPIN". In: *IEEE Transactions on software engineering* 23.5 (1997), pp. 279–295.

[Hun11] Robert Hundt. "Loop recognition in C++/Java/Go/Scala". In: *Proceedings of Scala Days* 2011 (2011), p. 38.

[ISO11] ISO/IEC. *SC22/WG14. ISO/IEC 9899: 2011*. Tech. rep. Geneva, Switzerland, 2011.

[ISO12] ISO ISO. *IEC 14882: 2011 Information technology — Programming languages — C++*. Tech. rep. 2012, p. 59.

[KLS+17] Michalis Kokologiannakis, Ori Lahav, Konstantinos Sagonas, and Viktor Vafeiadis. "Effective stateless model checking for C/C++ concurrency". In: *Proceedings of the ACM on Programming Languages* 2.POPL (2017), p. 17.

[KT14] Daniel Kroening and Michael Tautschnig. "CBMC–C bounded model checker". In: *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer. 2014, pp. 389–391.

[KVY11] Michael Kuperstein, Martin Vechev, and Eran Yahav. "Partial-coherence abstractions for relaxed memory models". In: *ACM SIGPLAN Notices*. Vol. 46. 6. ACM. 2011, pp. 187–198.

[Lam78] Leslie Lamport. "Time, clocks, and the ordering of events in a distributed system". In: *Communications of the ACM* 21.7 (1978), pp. 558–565.

[Lam79] Leslie Lamport. "How to make a multiprocessor computer that correctly executes multiprocess program". In: *IEEE transactions on computers* 9 (1979), pp. 690–691.

[LPM14] Daniel Lustig, Michael Pellauer, and Margaret Martonosi. "PipeCheck: Specifying and verifying microarchitectural enforcement of memory consistency models". In: *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society. 2014, pp. 635–646.

[LSM+16] Daniel Lustig, Geet Sethi, Margaret Martonosi, and Abhishek Bhattacharjee. "Coatcheck: Verifying memory ordering at the hardware-OS interface". In: *ACM SIGOPS Operating Systems Review* 50.2 (2016), pp. 233–247.

[MAM+17] Paul E. McKenney, Jade Alglave, Luc Maranget, Andrea Parri, and Alan Stern. *A formal kernel memory-ordering model (part 1)*. 2017. URL: https://lwn.net/Articles/718628/.

[MB11] Leonardo de Moura and Nikolaj Bjørner. *Z3 - a Tutorial*. 2011.

[McK] Paul E McKenney. *Is parallel programming hard, and, if so, what can you do about it? (v2017. 01.02 a)*.

[MPA05] Jeremy Manson, William Pugh, and Sarita V. Adve. *The Java memory model*. Vol. 40. 1. ACM, 2005.

[MQ08] Madanlal Musuvathi and Shaz Qadeer. "Fair stateless model checking". In: *ACM SIGPLAN Notices*. Vol. 43. 6. ACM. 2008, pp. 362–371.

[MZ09] Sharad Malik and Lintao Zhang. "Boolean satisfiability from theoretical hardness to practical success". In: *Communications of the ACM* 52.8 (2009), pp. 76–82.

[Oak14] Scott Oaks. *Java Performance: The Definitive Guide: Getting the Most Out of Your Code*. " O'Reilly Media, Inc.", 2014.

[OSS09] Scott Owens, Susmit Sarkar, and Peter Sewell. "A better x86 memory model: x86-TSO". In: *International Conference on Theorem Proving in Higher Order Logics*. Springer. 2009, pp. 391–407.

[Par13] Terence Parr. *The definitive ANTLR 4 reference*. Pragmatic Bookshelf, 2013.

[PFH+17a] Hernán Ponce de León, Florian Furbach, Keijo Heljanko, and Roland Meyer. "Portability Analysis for Weak Memory Models. PORTHOS: One Tool for all Models". In: *Static Analysis - 24th International Symposium, SAS 2017, New York, NY, USA, August 30 - September 1, 2017, Proceedings*. 2017, pp. 299–320. DOI: 10.1007/978-3-319-66706-5_15. URL: https://doi.org/10.1007/978-3-319-66706-5_15.

[PFH+17b] Hernán Ponce de León, Florian Furbach, Keijo Heljanko, and Roland Meyer. "Portability Analysis for Axiomatic Memory Models. PORTHOS: One Tool for all Models". In: *CoRR* abs/1702.06704 (2017). arXiv: 1702.06704. URL: http://arxiv.org/abs/1702.06704.

[PJ98] Jens Palsberg and C Barry Jay. "The essence of the visitor pattern". In: *Computer Software and Applications Conference, 1998. COMPSAC'98. Proceedings. The Twenty-Second Annual International*. IEEE. 1998, pp. 9–15.

[Ray03] Eric S Raymond. *The art of Unix programming*. Addison-Wesley Professional, 2003.

[Sht00] Ofer Shtrichman. "Tuning SAT checkers for bounded model checking". In: *International Conference on Computer Aided Verification*. Springer. 2000, pp. 480–494.

[SSA+11] Susmit Sarkar, Peter Sewell, Jade Alglave, Luc Maranget, and Derek Williams. "Understanding POWER multiprocessors". In: *ACM SIGPLAN Notices* 46.6 (2011), pp. 175–186.

[TW16] Oleg Travkin and Heike Wehrheim. "Verification of concurrent programs on weak memory models". In: *International Colloquium on Theoretical Aspects of Computing*. Springer. 2016, pp. 3–24.

Appendices

A.1 The ANTLR grammar for the porthos v1 input language

```
grammar Porthos;

main
 : program
 ;

bool_expression
 : bool_atom
 | bool_atom BOOL_OP bool_atom
 ;

bool_atom
 : TRUE
 | FALSE
 | '(' arith_expr COMP_OP arith_expr ')'
 | '(' bool_expression ')'
 ;

arith_expr
 : arith_atom ARITH_OP arith_atom
 | arith_atom
 ;

arith_atom
 : DIGIT
 | register
 | '(' arith_expr ')'
 ;

register
 : WORD
 ;

location
 : WORD
 ;

local
 : register '<-' arith_expr
 ;

load
 : register '<:-' location
 ;

store
 : location ':=' register
 ;

read
 : register '=' location '.' 'load' '('
   ATOMIC ')'
 ;

write
 : location '.' 'store' '(' ATOMIC ','
   register ')'
 ;

instruction
 : atom
 | sequence
 | while_
 | if
 ;

atom
 : local
 | load
 | store
 | FENCE
 | read
 | write
 ;

sequence
 : atom  ';' instruction
 | while_ ';' instruction
 | if   ';' instruction
 ;

if
 : 'if' bool_expression 'then' '{' instruction '}'
   ('else' '{' instruction '}')?
 ;

while_
 : 'while' bool_expression '{' instruction '}'
 ;

program
 : '{' location (',' location)* '}'
    ('thread t' DIGIT '{' instruction '}'
     ('exists' (location '=' DIGIT ','
      | DIGIT ':' register '=' DIGIT ',')
      )*
    )*
 ;

// Lexer rules:

ATOMIC
 : '_na' | '_sc' | '_rx' | '_acq' | '_rel' | '_con'
 ;

FENCE
 : 'mfence' | 'sync' | 'lwsync' | 'isync'
 ;

COMP_OP : '==' | '!=' | '<=' | '<' | '>=' | '>';
ARITH_OP : '+' | '-' | '*' | '/' | '%';
BOOL_OP : 'and' | 'or';

DIGIT : [0-9];
LETTER : 'a'..'z' | 'A'..'Z';
TRUE  : 'true' | 'True';
FALSE : 'false' | 'False';
WORD  : (LETTER | DIGIT)+;
```

A.2 File trees of `Y-tree` and `X-graph` representations

```
ytree/
├── definitions
│   ├── YDefinition.java
│   └── YFunctionDefinition.java
├── expressions
│   ├── accesses
│   │   ├── YIndexerExpression.java
│   │   ├── YInvocationExpression.java
│   │   └── YMemberAccessExpression.java
│   ├── assignments
│   │   └── YAssignmentExpression.java
│   ├── atomics
│   │   ├── YAtom.java
│   │   ├── YConstant.java
│   │   ├── YLabeledVariableRef.java
│   │   ├── YParameter.java
│   │   ├── YParameterListBuilder.java
│   │   └── YVariableRef.java
│   ├── operations
│   │   ├── YBinaryExpression.java
│   │   ├── YBinaryOperator.java
│   │   ├── YPointerUnaryExpression.java
│   │   ├── YUnaryExpression.java
│   │   └── YUnaryOperator.java
│   ├── ternary
│   │   └── YTernaryExpression.java
│   ├── YEmptyExpression.java
│   ├── YExpression.java
│   └── YOperator.java
├── litmus
│   ├── YPostludeStatement.java
│   ├── YPreludeStatement.java
│   └── YProcessStatement.java
├── statements
│   ├── jumps
│   │   ├── YJumpLabel.java
│   │   └── YJumpStatement.java
│   ├── YBranchingStatement.java
│   ├── YCompoundStatement.java
│   ├── YLinearStatement.java
│   ├── YStatement.java
│   ├── YUnlabeledStatement.java
│   ├── YVariableDeclarationStatement.java
│   └── YWhileLoopStatement.java
├── types
│   ├── YMethodSignature.java
│   └── YType.java
├── YEntity.java
└── YSyntaxTree.java
```

```
xgraph/
├── events
│   ├── barrier
│   │   └── XBarrierEvent.java
│   ├── computation
│   │   ├── XAssertionEvent.java
│   │   ├── XBinaryComputationEvent.java
│   │   ├── XBinaryOperator.java
│   │   ├── XComputationEvent.java
│   │   ├── XOperator.java
│   │   ├── XTypeDeterminer.java
│   │   ├── XUnaryComputationEvent.java
│   │   └── XUnaryOperator.java
│   ├── controlflow
│   │   ├── XEntryEvent.java
│   │   ├── XExitEvent.java
│   │   ├── XJumpEvent.java
│   │   └── XNopEvent.java
│   ├── memory
│   │   ├── XInitialWriteEvent.java
│   │   ├── XLoadMemoryEvent.java
│   │   ├── XLocalMemoryEvent.java
│   │   ├── XMemoryEvent.java
│   │   ├── XRegisterMemoryEvent.java
│   │   ├── XSharedMemoryEvent.java
│   │   └── XStoreMemoryEvent.java
│   ├── XEventInfo.java
│   └── XEvent.java
├── memories
│   ├── XConstant.java
│   ├── XLocalLvalueMemoryUnit.java
│   ├── XLocalMemoryUnit.java
│   ├── XLocation.java
│   ├── XLvalueMemoryUnit.java
│   ├── XMemoryUnit.java
│   ├── XRegister.java
│   ├── XRvalueMemoryUnit.java
│   ├── XSharedLvalueMemoryUnit.java
│   └── XSharedMemoryUnit.java
├── process
│   ├── XCyclicProcessBuilder.java
│   ├── XCyclicProcess.java
│   ├── XProcessBuilder.java
│   ├── XProcessId.java
│   ├── XProcess.java
│   └── XProcessKind.java
├── program
│   ├── XCyclicProgramBuilder.java
│   ├── XCyclicProgram.java
│   ├── XProgramBuilder.java
│   └── XProgram.java
├── types
│   ├── XMockType.java
│   └── XType.java
└── XEntity.java
```

A.3 The x86-TSO memory model defined in cat language

```
"X86 TSO"
include "x86fences.cat"
include "filters.cat"
include "cos.cat"

(* Uniproc check *)
let com = rf | fr | co
acyclic po-loc | com

(* Atomic *)
empty rmw & (fre;coe)

(* Global happens-before *)
#ppo
let po_ghb = WW(po) | RM(po)

#mfence
include "x86fences.cat"

#implied barriers
let poWR = WR(po)
let i1 = MA(poWR)
let i2 = AM(poWR)
let implied = i1 | i2

let com = rfe | fr | co
let ghb = mfence | implied | po_ghb | com
show implied
acyclic ghb as tso
```

A.4 Public interface methods of the X-interpreter

```
public interface XInterpreter {

    XProcessId getProcessId();
    XCyclicProcess getResult();
    void finishInterpretation();

    // memory manager interface methods:
    XLocation declareLocation(String name, XType type);
    XRegister declareRegister(String name, XType type);
    XRegister declareTempRegister(XType type);
    XLvalueMemoryUnit declareUnresolvedUnit(String name, boolean isGlobal);
    XLvalueMemoryUnit getDeclaredUnitOrNull(String name);
    XRegister getDeclaredRegister(String name, XProcessId processId);
    XLocalMemoryUnit tryConvertToLocalOrNull(XEntity expression);
    XComputationEvent tryEvaluateComputation(XEntity entity);
    XRegister copyToLocalMemory(XSharedMemoryUnit shared);

    // linear interpretation methods:
    XEntryEvent emitEntryEvent();
    XExitEvent emitExitEvent();
    XBarrierEvent emitBarrierEvent(XBarrierEvent.Kind kind);
    XJumpEvent emitJumpEvent();
    XNopEvent emitNopEvent();
    XLocalMemoryEvent emitMemoryEvent(XLocalLvalueMemoryUnit destination, XLocalMemoryUnit source);
    XSharedMemoryEvent emitMemoryEvent(XLocalLvalueMemoryUnit destination, XSharedMemoryUnit source);
    XSharedMemoryEvent emitMemoryEvent(XSharedLvalueMemoryUnit destination, XLocalMemoryUnit source);
    XAssertionEvent emitAssertionEvent(XBinaryComputationEvent assertion);

    // non-linear interpretation methods:
    void startBlockDefinition(BlockKind blockKind);
    void startBlockConditionDefinition();
    void finishBlockConditionDefinition(XComputationEvent conditionEvent);
    void startBlockBranchDefinition(BranchKind branchKind);
    void finishBlockBranchDefinition();
    void finishNonlinearBlockDefinition();
    void processJumpStatement(JumpKind kind);
    XEntity processMethodCall(String methodName, @Nullable XMemoryUnit receiver, XMemoryUnit... arguments);

    // create computation event without processing it:
    XComputationEvent createComputationEvent(XUnaryOperator operator,
                                             XLocalMemoryUnit operand);
    XComputationEvent createComputationEvent(XBinaryOperator operator,
                                             XLocalMemoryUnit firstOperand,
                                             XLocalMemoryUnit secondOperand);

    // --

    enum BlockKind {
        Sequential,
        Branching,
        Loop,
        ;
    }

    enum BranchKind {
        Then,
        Else,
        ;
    }

    enum JumpKind {
        Break,
        Continue,
        ;
    }
}
```