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

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

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 **Pages:** ?? + ??

Professorship?: **Code?:** AS-116

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

Instructor:

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: Thesis template, master's thesis

Language: English

Aalto-yliopisto
Sähkötekniikan korkeakoulu

Aalto-yliopisto
Perustieteiden korkeakoulu
??? ???

Tekijä:	Artem YUSHKOVSKIY		
Työn nimi: ?			
Päiväys:	?.?.2018	**Sivumäärä:**	**??** + **??**
Professuuri:	?	**Koodi:**	AS-116
Valvojat:			
Ohjaaja:			

Cras tincidunt bibendum erat, vel tincidunt diam porttitor aliquam. Donec sit amet urna non felis placerat pharetra. Aenean ultrices facilisis nulla vitae semper. Nullam non libero quis dui fermentum aliquam id vel eros. Praesent elementum tortor quis sem congue iaculis sit amet eget nisl. Quisque erat tortor, condimentum eu volutpat et, blandit et augue. Phasellus erat turpis, pretium non feugiat id, posuere id velit. Vestibulum ut sapien felis, quis convallis dui.

In elementum est eu nulla hendrerit feugiat. In sodales diam vel lacus cursus tincidunt. Morbi nibh dui, imperdiet non vestibulum non, dignissim id risus. Sed sollicitudin neque lectus, porttitor sollicitudin elit. Nulla facilisi. Nullam in ante eu mi suscipit sollicitudin. Sed est velit, gravida facilisis varius eget, tempus sed urna. Aliquam erat volutpat. Nam semper condimentum nisi. Nullam scelerisque, metus nec sodales vulputate, purus augue venenatis urna, sit amet mattis turpis nisl ac metus. Mauris nec odio ut neque condimentum vulputate vel in turpis. Nulla facilisi. Nulla id tellus sapien, vitae blandit lorem.

Asiasanat:	Diplomityöpohja
Kieli:	Englanti

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

Artem Yushkovskiy

Abbreviations

LI	Lorem Ipsum
ABC	Quisque et mi lacus, nec porta ante.
DEF	Proin pellentesque accumsan laoreet

Contents

Chapter 1

Introduction

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 [**mckenney2017parallel**].

Traditionally, studies related to concurrent programming concern on more fundamental theoretical questions of designing race-free and lock-free parallel algorithms, asynchronous data structures and synchronisation primitives of a programming language. Unfortunately, when it comes to the real-world concurrent programs, the algorithmic level of abstraction is not enough for guaranteeing their properties of correctness and reliability. 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. For instance, Figure **??** provides simple example of reachability of the state '`(0:EAX=0 /\ 1:EAX=0)`' on x86 machines (such little examples that illustrate specific behaviour of a WMM are called *litmus tests*). This state is allowed because in x86 architecture each processor may cache the write to shared memory variable into its local write buffer, so that they do not become visible by other processes immediately. 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 updated version of the variable `[x]`, neither may have 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 has been already updated in another thread. These problems have lead to the need for formalisation of

{ 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*

semantics of memory operations within different concurrent architectures defined by *weak memory models (WMM)*.

Research of weak memory models firstly aims to *formalise* develop the formal approach of understanding programs with respect to weak memory models which is systematic, sound and complete. The first (and so far the only) such a framework was presented in 2010 [**alglave2010shared**]. 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 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 [**owens2009better**], much more relaxed memory model for Power and ARM architectures [**alglave2009semanticssarkar2011unde** and others. There are projects for validating hardware architectures wrt. a memory model, e.g. [**lustig2014pipechecklustig2016coatcheck**].

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 [**lamport1978time**], it was introduced in J2SE 5.0 in 2004 [**manson2005java**]; the C++11 standard [**iso2012iec**] has introduced the set of hardware-independent synchronisation fences and atomic operations, whenever the C++17 memory model [**batty2011mathematizing**] 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 [**kernel1**]. Furthermore, there is a wide range of tools that perform program verification wrt memory models (see [**alglave2013software**], [**Porthos17**]).

The first memory model for concurrent systems was formulated by Leslie Lamport back in 1979 [**lamport1979make**]. This memory model, called the *sequential consistency (SC)*, allows only those executions (interleavings) that produce the same result as if the operations had been executed by single process. This means that the order of operations executed by a process is strictly defined by the program it executes. The SC model does requires the write to a shared variable performed in one process to become visible by all other processes not instantly, but simultaneously. This means each process communicates to the shared memory directly, without local buffering. Another important requirement of SC memory model is that it forbids memory operations reordering within single process (the order is strictly defined by the program).

The SC model is considered to be the strong memory model in the sense that it provides strong guarantees regarding the ordering and caused effect of memory operations. Different relaxations of this model lead to the class of *weak memory models (WMM)*. They specify how threads interact through shared memory, when a write becomes visible to other threads and what value a read can return. Therefore, 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 expect.

Although weak memory studies is rather young research area, there exist frameworks and tools for exploring WMMs and examining simple programs with respect to the them. The state-of-the-art tool is `diy` (for *do it yourself*), developed by the researchers from INRIA institute, France and University of Cambridge, UK. The `diy` [1] 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 tests concrete executor `litmus` that runs tests on a physical machine while collecting its behaviours, and the weak memory models simulator `herd` that implements reachability analysis for exploring states reachable under specified WMM.

All the `diy` tools work only with 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

[1] Project web site: `http://diy.inria.fr/`

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 [**Porthos17**].

The first tool that performs the WMM-aware portability analysis is porthos[2]introduced in April 2017 [**Porthos17**]. 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 single 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 ([**malik2009boolean**]), the used method can be applicable in solving the real-world problems.

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, reliability and maintainability. As the general architecture and almost all components of porthos have been redesigned, the tool received a new name – porthos2[3] . Considering the enhancements of the architecture, porthos2 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.

1.1 Thesis structure

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

[2]Project web site: http://github.com/hernanponcedeleon/PORTHOS

[3]Hereinafter with the name 'porthos' we refer to the tool porthos version 1 (also addressed as porthos v1), whereas the new version of porthos is called porthos2.

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 [**alglave2016syntax**]. 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 a program as a directed graph of events (the *event-flow graph*). The vertices of such a graph represent *events* (see Section **??**), and edges represent *basic relations* (see Section **??**).

2.1.1 Events

The event is an independent low-level atomic operation.

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 $\mathtt{id}(e_m)$ [**alglave2010shared**].

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 [**alglave2016syntax**].

The basic relations are the following [**alglave2010shared**]:

- 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 `porthos2` 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 [**alglave2010shared**]:

- 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 $\mathtt{po}$ 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:
  $w \xrightarrow{\texttt{fre}} r \triangleq (w \xrightarrow{\texttt{fr}} r \wedge \texttt{proc}(w) = \texttt{proc}(w))$
  $w \xrightarrow{\texttt{fri}} r \triangleq (w \xrightarrow{\texttt{fr}} r \wedge \texttt{proc}(w) \neq \texttt{proc}(w))$

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

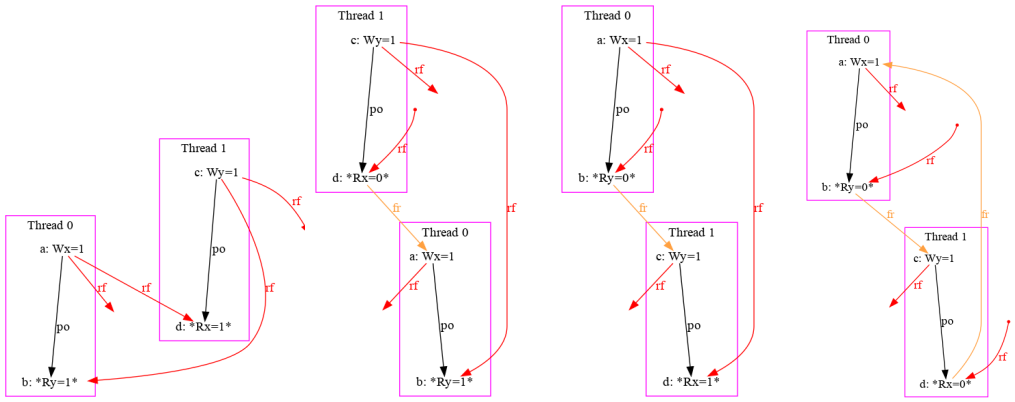
- the semantics of *fences* (memory barriers) specific for different architectures may be defined as derived relations.

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 [**alglave2014herding**]. 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 **??** illustrates four possible candidate executions for the litmus test Example **??** (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 **??**–**??** 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 **??** is still consistent under relaxed-memory architectures, but it becomes inconsistent under SC architecture as it forbids cycles over $\texttt{fr} \cup \texttt{po}$.



(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* **??**

2.2 The cat language

Weak memory models are defined via the `cat` language [**alglave2016syntax**]. 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 [**alglave2016syntax**].

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*[1] of two relations `r1` and `r2` is `r1 ; r2`

For instance, the `fr`-relation is defined as a sequence of inverted `rf`-relation and `co`-relation: $\texttt{fr} = (\texttt{rf}^{-1}; \texttt{co})$. Figure **??** contains an example of x86-TSO weak memory model definition in `cat` language that asserts acyclicity of communication relation, `po-loc` relation, `mfence` relation and some other derived relations [**owens2009better**]:

```
"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
```

Figure 2.2: *The x86-TSO memory model defined in* `cat` *language*[2]

[1] The sequence of two relations `r1` and `r2` 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`

[2] The `cat` -memory model for x86-TSO can be found in official repository of `herd` tool: `https://github.com/herd/herdtools7`

Chapter 3

Portability analysis as an SMT problem

As it has been discussed in Chapter **??**, 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 [**Porthos17**] (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 [**clarke2012model**]. 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 [**clarke2001bounded**]. 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., [**shtrichman2000tuning**]).

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 $\mathcal{M}_\mathcal{S}$ [**Porthos17**]:

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*) [**Porthos17**]. 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

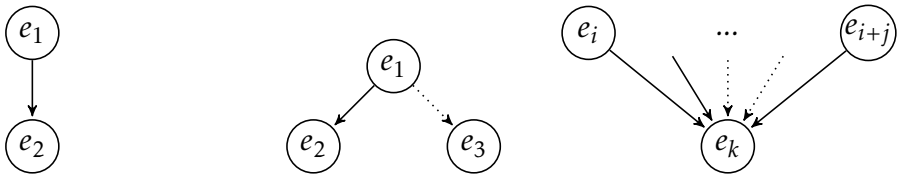
The control-flow of a program is represented in the *control-flow graph*, a directed acyclic connected graph with single source and multiple sink nodes, obtained by the *loop unrolling* (see Section **??**).In control-flow graph, there are two types of transitions (edges): *primary transitions* that 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 `porthos2`, 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 `porthos2` 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 **??**).

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 **??** for details on relations). We distinguish the function $\mathbf{v}_p : \mathbb{C}_{\text{I}} \to \{0, 1\}$ 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*

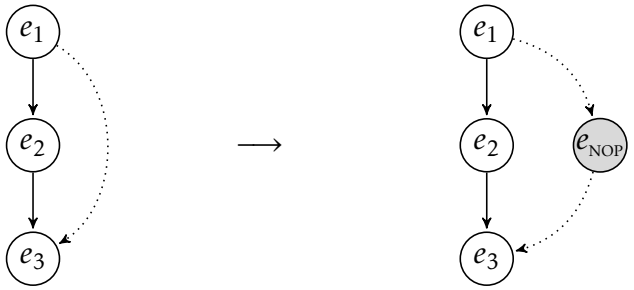


Figure 3.2: *Transformation of the empty-branch nonlinear control-flow*

For listed cases, below we propose the encoding scheme that uniquely encodes each node of graph and allows to encode partially executed program. Equation **??** encodes the sequential control-flow represented in Figure **??** and reflects the fact that the event e_2 can be executed iff the event e_1 has been executed. Equation **??** encodes the branching control-flow depicted in Figure **??** by allowing only following executions: $\{\emptyset, (e_1), (e_1 \rightarrow e_2), (e_1 \rightarrow e_3)\}$. In encoding **??** of the merge-point represented in Figure **??**, 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 **??**:

3.2.2 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 [**Porthos17**].

Following [**Porthos17**], 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 **??**.

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 porthos2: implementation

The main reason for commencing the work on porthos2 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. Although far not the entire 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 v1 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, that uses a C parser written in more than 18.5 thousand lines (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 porthos2, 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 porthos2 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 porthos2 we mainly followed the *KISS principle*, which can be exhaustively described in 17 Unix Rules of Eric Raymond [**raymond2003art**].The following list summarises the main rules we followed during the development of porthos2:

1. *Robustness:*
   1.1. usage of immutable data structures for all DTOs;
   1.2. interrupting the work on errors;
   1.3. modular architecture: each module can be tested independently;
   1.4. usage of software design patterns if necessary;
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.

As porthos v1, the porthos2 uses the open-source SMT solver Z3 from Microsoft Research [**de2008z3**]. However, unlikely its predecessor, the

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

porthos2 has an additional abstraction level Z-formula (see Section **??**) that allow to use any other SMT solver.

The programming language choice for porthos2 was also made in favour of *java*, firstly, in order to be able to reuse some parts and concepts of porthos v1 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++ [**hundt2011loopoaks2014java**]), the performance cornerstone of porthos2 (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-solver *Z3*[3] invoked from porthos2 via java API. However, considering the perspective of using porthos2 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 porthos2 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 porthos2 is presented in Figure **??**. On the picture, rectangles denote processing units (marked with gear sign with a unique number of the component).

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 **??**) to a program AST called Y-tree[4](Section **??**), which then is being pre-processed at the pre-compilation stage (Section **??**) in order to collect information necessary for the compilation. The Y-tree then is being compiled (Section **??**) to an X-graph representation (Section **??**). The compiled X-graph then undergoes a number of transformations (Section **??**) necessary for the encoding into a Z-formula (Section **??**) at the pre-encoding stage (Section **??**). Then, the memory-model constructor (Section **??**) constructs the derived

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

[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.

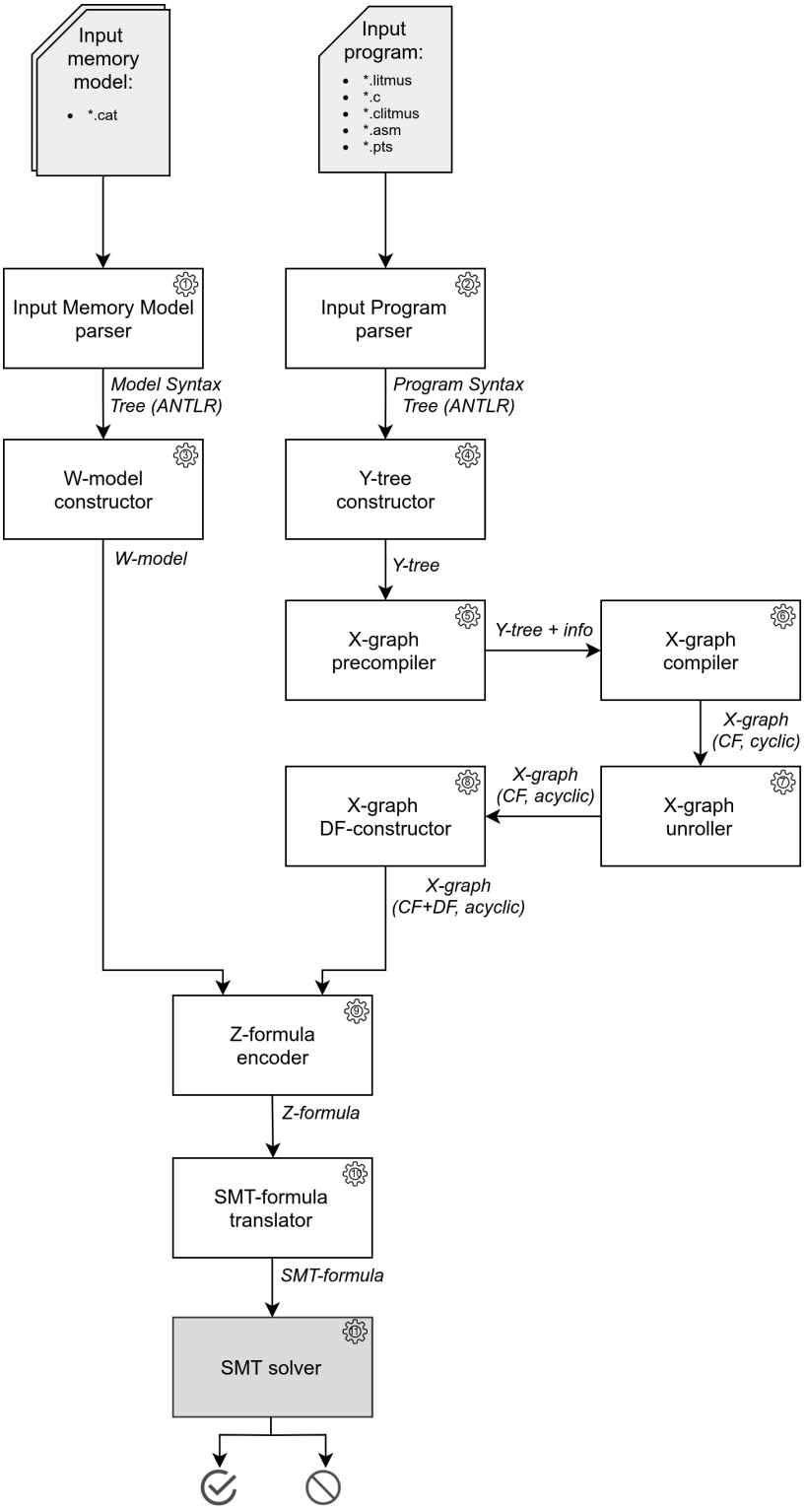


Figure 4.1: *The general architecture of* porthos2

relations on the basis of the X-graph in order to build the weak memory model W-model (Section **??**). Thereafter, W-model and transformed X-graph are encoded (Section **??**) to a Z-formula representation (a wrapper over an SMT logical formula), which then is passed as an input to the SMT-solver.

4.2.1 Program input

Both porthos v1 and porthos2 use the ANTLR parser generator[5] [**parr2013definitive**], 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 **??** represents the simplified grammar of of input language used by previous version of porthos (the full ANTLR grammar is available at Appendix **??**.

The input language parser used by porthos v1 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 (which is called semantic actions). Such a combining of two rich languages[6] makes the code hardly understandable, and, therefore, poorly maintainable. In porthos2, 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, the semantics of operations was defined syntactically, whereas processing programs written in most modern languages (including C) requires the semantics resolution based on typification[7].As the reader can notice from the grammar sketch in Figure **??**, the memory operations of different kinds vary syntactically. For example, the assignment of local

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

[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]for instance, given two functions '`int foo(int a)`' and '`int foo(char a)`', the code '`int a = '1'; foo(a);`' will invoke the first method rather than the second one.

```
<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

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 method name. In porthos2, 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 **??**). The semantics of the methods also is being resolved during the pre-compilation stage via the *invocation hooks* mechanism (see Section **??**).

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 v1. In porthos2, we extended support for primitive

types supported by the Z3 solver (this apply to 32-bit integers encoded as Ints, floats encoded as Reals, enums encoded as Z3 Scalars). Thus, adding support for more advanced syntactic structures as pointers, arrays, function definitions and calls was one of the purposes of revising the tool architecture.

The minor drawbacks of grammar used by porthos v1 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 v1 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 porthos2 supports arbitrary declaration to be performed in initialisation statement.

The porthos2 uses the C language grammar of proposed in the C11 standard [**jtc2011sc22**], 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]).Currently, porthos2 can operate only in the intra-procedural analysis mode (analysing single procedure), assuming that each function defined in the input file is being executed in a separate thread. However, the redesigned architecture of porthos2 allow to easily support inter-procedural (cross-procedure) analysis by inlining function calls and binding variable contexts. Also, Current version of porthos2 simply ignores C processor directives, however, in future it is possible to support it.

4.2.2 The internal representations

As it has already been mentioned, all internal representations used by porthos2 are immutable. 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 porthos2 is the *Y-tree*, which represents a rather high-level AST. The Apendix **??** represents the file tree

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

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).

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 [**gries2012science**]), 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 [**iso2012iec**], 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.

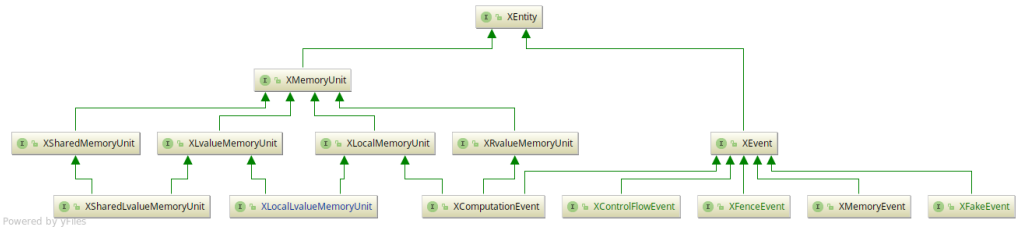


Figure 4.2: *The inheritance tree of interfaces of X-graph*

On the Y-level of abstraction, we define the `YType` as an alias for a type (since the Y-tree is not typed, all expressions do not have type, however, the `YType` is used for storing the information of the type for declarations

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 *X-graph*. The mathematical structure of event-flow graph was discussed in Section **??**. 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 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 **??** represents the main hierarchy of the X-abstraction level.

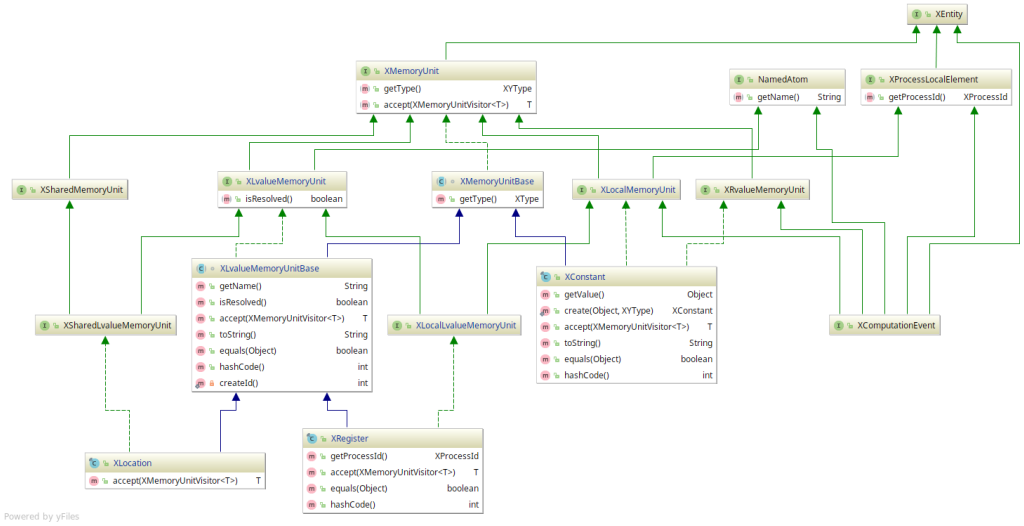


Figure 4.3: *The inheritance tree of X-graph memory units*

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.

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 **??** 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 convertable to a SMT type (modelled as `ZType`).

[9]Here we should note the `porthos2` 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 **??**.

- memory manager: no contexts so far; locals per process + shareds; register/deregister;

Events
The following interfaces capture main types of events (see also Figure **??**):

- `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):

  1. `XRegisterMemoryEvent`:
     `XLocalLvalueMemoryUnit` ← `XLocalMemoryUnit`
  2. `XLoadMemoryEvent`:
     `XLocalLvalueMemoryUnit` ← `XSharedMemoryUnit`
  3. `XStoreMemoryEvent`:
     `XSharedLvalueMemoryUnit` ← `XLocalMemoryUnit`
  4. `XInitialWriteEvent`:
     `XLvalueMemoryUnit` ← `XRvalueMemoryUnit`

- `XComputationEvent`
  - binary and unary
  - implements both `XEvent` and `XMemoryUnit`. This is a model-level optimisation possible **due to** because a computation-event performs a computation over local-only memory and does not change the value of any memory unit, therefore the *computation* abstraction (the CPU time) can be safely removed from the model, and the computation event can be seen as an atomic operation.

- `XControlFlowEvent`
  `XJumpEvent` no computation, no data operation. may be safely removed for optimisation `XMethodCallEvent` : TODO: meth call should inherit both control-flow and computation. itself - computation result (temp register) + if (method resolved) => context binding + jump to the method body

- `XFenceEvent` `XBarrierEvent` : mfence, sync, optsync, lwsync, optlwsync, ish, isb, isync

- `XFakeEvent`

1. XEntryEvent, the source event in the event-flow graph
2. XExitEvent : two kinds (depending on the unrolling boundary)
3. XNopEvent : no-operation event, used for correct encoding (picture); same as simple jump to the next event

Edges (relations edges (just map with enum as kind of edge)):

1. CF edge: then-edge (primary), else-edge (alternative)
2. DF edge: co, rf

Valid graph:

1. single source, two sinks
2. connected
3. each node has 1 or 2 children
4. branching node (2 children) instanceof XComputationEvent (evaluation of the value, no shared-memory operations)

consider the following code in C:
picture of the CF-graph:
XEventInfo: Events are specified by the process generated them and a unique label. This information is stored by events as the immutable structure XEventInfo.

4.2.2.3 W-model

<to be done>
a simple set of recursively defined relations over ..
atoms: sets/basic relations
itemize elements

4.2.2.4 Z-formula

in porthos v1 : 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 porthos2 we created the new abstraction layer Z-formulathat is translated to

also recursive representation of smt formula
used as an abstrac
implemented as a simple wrapper over the z3 formula
typed: Expr, BoolExpr, ArithExpr, ...

4.2.3 The processing units

- numbers – same as in gears in Figure **??**.

4.2.3.1 Input memory model parser (1)

ANTLR grammar is extracted from parser used by herd written in OCaml

4.2.3.2 Input program parser (2)

todo: move something from the input language chapter here

4.2.3.3 W-model constructor (3)

visitor of antlr tree: almost direct
todo: grammar of w-model to appendix
in perspective: work with functional-style definitions
<not implemented yet>

4.2.3.4 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?)

4.2.3.5 X-graph precompiler (5)

The macro preprocessor
typedefs, aliases
not supported yet

Variables kinds determination
- pointers - externs - parameters of process-functions

The label resolution
The label resolution is a process of linkage the referenced labels to declared labels.

In C, the labeled 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 (see Section **??**).

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 Z-formula encoding stage (see Section **??**).

- X-type construction: code of class YType2TypeConverter (todo: rename: it's not conversion, it's a type construction, from the y-level name+qualifiers+

- The typification algorithm: propagation rules for each of Y

Invocation hooking
resolve the semantics of a function given its signature (meth name + params types)

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

4.2.3.6 X-graph compiler (6)

- hierarchy of Compilers (XCompiler is an stateful abstract machine)

Firstly, the Y-tree is compiled to the *cyclic control-flow event-based graph* X-graph$_{CF}$.

Then: unrolled up to bound k to the *acyclic control-flow event-based graph* X-graph$^{u}_{CF}$.

Then: data-flow analysis to the *acyclic full event-based graph* X-graph$^{u}_{CF+DF}$.

4.2.3.7 X-graph unroller (7)

- 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)

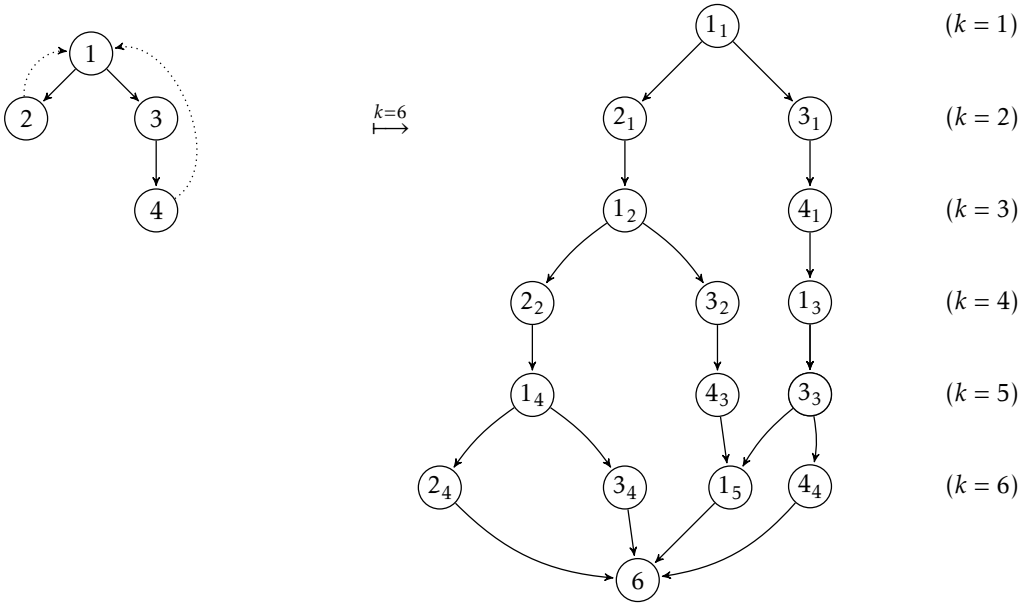


Figure 4.4: *Example of the flow graph from Figure* **??**, *unwinded up to the bound* $k = 6$

4.2.3.8 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)

4.2.3.9 Z-formula encoder (9)

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

4.2.3.10 SMT-formula translator (10)

4.2.3.11 SMT-formula translator (11)

4.2.4 Program output

structure: verdict

4.2.5 Auxiliary components

4.2.5.1 Watchdog timer

<not implemented>
todo

4.2.5.2 Logger

<not implemented>
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

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
```