Aalto University, Department of Computer 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
18.6.2018

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

Aalto University, Department of Computer 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:	18.6.2018 **Pages:** ?? + ??
Supervisors:	Assoc. Prof. Keijo Heljanko Docent Igor I. Komarov

Software verification is considered to be a hard computational problem vulnerable to the state explosion problem. Concurrent software verification raises the complexity of the problem to a power determined by all the possible interleavings of states of the system. Moreover, the architecture of a modern shared-memory multi-core processor and optimisations performed by a compiler can cause program behaviour that is unexpected from the point of view of traditional concurrency. The guarantees that an execution environment can provide to a programmer are formalised in its *weak memory model (WMM)*. Over the last decade, weak memory models were defined for multiple hardware architectures and programming languages. This opens new challenges in software verification with respect to a weak memory model.

Most existing tools that perform memory model-aware software analysis examine behaviours of the program against a single memory model. The first tool that analyses the *portability* of a concurrent program from one platform to another is *Porthos* [**Porthos17a**] released in April 2017. Porthos can verify that the program is portable from the source platform $\mathcal{S}$ to the target platform $\mathcal{T}$ by checking that the program has no extra states under $\mathcal{T}$. For that, it performs an SMT-based bounded reachability analysis by encoding the constraints of the program and two memory models $\mathcal{M}_{\mathcal{S}}$ and $\mathcal{M}_{\mathcal{T}}$ into a single SMT-formula.

Although the approach has been proven to be efficient, the tool accepts as input the small C-like toy language. Current thesis aims to rework Porthos by extending its input language, so that it is able to process real-world C programs. However, current implementation of Porthos seems to be hard to extend, which raises the need to redesign its whole architecture in order to increase the robustness, transparency, efficiency and extensibility. The result of the work is *PorthosC*, a framework for SMT-based memory model-aware analysis.

Keywords:	Weak memory models, concurrent programming, software verification, portability analysis, bounded reachability analysis, SMT-encoding
Language:	English

Acknowledgements

<TODO>

Espoo, Finland & Saint Petersburg, Russia
18.6.2018

Artem Yushkovskiy

Abbreviations

AST	Abstract Syntax Tree
BDD	Binary Decision Diagrams
BNF	Backus-Naur form
BMC	Bounded Model Checking
CF	Control-Flow
CPU	Central Processor Unit
CTL	Computational Tree Logic
DF	Data-Flow
DFS	Deep-First Search
DSA	Dynamic Single-Assignment form
DTO	Data-Transfer Object
LTL	Linear Temporal Logic
NP	Non-deterministic Polynomial time
OOP	Object-Oriented Programming
SB	Store Buffering
SC	Sequential Consistency
SAT	Satisfiability problem
SMT	Satisfiability Modulo Theories problem
SSA	Static Single-Assignment form
UMC	Unbounded Model Checking
UML	Unified Modeling Language
WMM	Weak Memory Model

Contents

List of Figures

Chapter 1

Introduction

1.1 Problem statement

Most modern computer systems contain large parts that operate concurrently. Although the parallelisation of a system can drastically improve its performance, it opens numerous of problems regarding 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 focus on more fundamental theoretical questions of designing race-free and lock-free parallel algorithms, asynchronous data structures and synchronisation primitives of a programming language [**ben2006principles**]. 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 of the system [**adve1996shared**].

As an example, consider a shared memory concurrent system that executes two parallel processes as it is described in Figure **??** (such little examples that illustrate specific behaviour of a concurrent execution environment are called *litmus tests*). The process `P0` writes the value `1` to the shared variable `x` and reads a value of the shared variable `y`, and the process `P1` writes the `y` and reads the `x`. Considering all interleavings of the processes, one could expect the final state to be one of the following:

- `(0:EAX=0, 1:EBX=1)`,
- `(0:EAX=1, 1:EBX=0)`,
- `(0:EAX=1, 1:EBX=1)`.

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

Figure 1.1: *Store buffering: A litmus test illustrating the write-read reordering allowed by the x86-TSO memory model*

However, on the x86 architecture the state '(0:EAX=0, 1:EAX=0)' is also reachable as the processors may cache the write to shared memory into their local write buffers, so that the write does not immediately become visible by processes running on other cores. This behaviour is known as *store buffering (SB)*.

The formal way to define the semantics of memory operations and synchronisation primitives of a parallel execution environment is to define its *memory model*. There are two main types of formal memory models. Models of the first type characterise the behaviour of the system (its *operational semantics*) in terms of the abstract machine executing the code, as it was done for the SB example above. Models of the second type define the *axiomatic semantics* of the system by specifying the set of assertions over states of the program. Although the former type of memory models may be easier to describe and interpret, existing numerous formal verification tools and methods address the research towards axiomatic memory models.

The first memory model for a concurrent system was formulated by Leslie Lamport back in 1979 [**lamport1979make**]. 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* as each process writes directly to the shared memory, without local buffering.

[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 [**ben2006principles**] by naming a concurrent piece of code the *process*.

Another important requirement of the SC memory model is that it forbids reordering of memory operations within a single process (the order is strictly defined by the program). Originally, the operational semantics was defined for the SC model, however there exist axiomatic specifications for it [**mansky2015axiomatic**].

The SC model is considered to be a *strong memory model* in the sense that it provides firm guarantees regarding the ordering and effect of memory operations. Different relaxations of the SC model lead to *weak memory models (WMMs)* that specify how processes interact through the shared memory, when a write becomes visible to processes running on other cores, and what value a read operation can get. 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 can rely on.

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*. One of the most well-known frameworks for weak memory model-aware analysis was formalised by J. Alglave in 2010 [**alglave2010shared**]. It is the event-based non-deterministic model without global time (see Section **??** for details).

In addition to developing the theoretical basis, researchers work on extracting the memory models for hardware architectures from existing implementations or from the specifications, which are written in natural language and thus suffer from ambiguities and incompleteness. Over the last decade, 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 architectures [**owens2009better**], much more relaxed memory model for Power and ARM architectures [**alglave2009semantics**; **sarkar2011understanding**; **alglave2014herding**], etc. There are projects for validating hardware architectures wrt. a memory model as well, e.g. [**lustig2014pipecheck**; **lustig2016coatcheck**].

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**] and was introduced in J2SE 5.0 in 2004 [**manson2005java**]. The transformations valid for the Java

memory model were discussed in [**sevcik2009program**]. A weak memory model for C and C++ is based on the relation *strongly happens-before* and was introduced in the C++17 standard [**batty2011mathematizing**]. Before that, the C++11 standard [**iso2012iec**] proposed the set of hardware-independent synchronisation fences and atomic operations. Weak memory models 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 [**alglave2018frightening**; **kernel1**].

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 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 [**kokologiannakis2017effective**], CHESS [**musuvathi2008fair**] and Nidhugg [**abdulla2017stateless**], the tool Trencher for checking programs against the TSO memory model [**bouajjani2013checking**], the tool Porthos for portability analysis [**Porthos17a**],
- Some tools tackle the problem of automated synthesis of the synchronisation primitives, such as the automatic fence insertion tool musketeer [**alglave2014don**], and the automatic verification and fence inference tool blender [**kuperstein2011partial**].
- 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 [**kroening2014cbmc**], the tool that performs the sequen-

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

tialisation of concurrent programs [**alglave2013software**], the tool Weak2SC for producing program descriptions which can be fed into standard model checking tools (such as SPIN [**holzmann1997model**] or NuSMV [**cimatti2000nusmv**]) for performing memory model-aware analysis [**travkin2016verification**].

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 platform $\mathcal{T}$ can reach states that are unreachable on the platform $\mathcal{S}$, which is a *portability bug* from the source platform $\mathcal{S}$ to the target platform $\mathcal{T}$ [**Porthos17a**].

The very first tool that performs the WMM-aware portability analysis is Porthos[3] introduced in April 2017 [**Porthos17b**]. This tool reduces described problem to a bounded reachability problem, which can be solved via an SMT solver. This approach allows to capture symbolically the semantics of analysing program and both weak memory models into a 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.

1.3 Task specification

The 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. One of the directions of development was the ability to process the *kernel litmus tests* written in C [**mckenney2017wg21**]. For that, in addition to supporting the C syntax (augmented by litmus-style definitions such as initialisation or assertion statements), the tool must be able to recognise kernel-specific

[3]The Porthos project web site: `http://github.com/hernanponcedeleon/Dat3M`

functions and macros (such as the macro `READ_ONCE` that guarantees memory read).

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 architectural design, 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 analysis of concurrent programs.

1.4 Thesis structure

The thesis is organised as following. Chapter **??** gives a general view on the weak memory model-aware analysis. Chapter **??** examines the portability analysis as an bounded reachability problem that can be encoded into an SMT-formula in order to be solved automatically by an SMT solver. Chapter **??** delves into the description of architectural solutions and implementation details of the PorthosC framework. Chapter **??** provides an example of the portability analysis via PorthosC and presents some evaluation results comparing to other tools. Chapter **??** summarises results of the work and proposes future work directions.

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

Chapter 2

Memory model-aware analysis

The main idea behind the memory model-aware program analysis is that the set of all possible interleavings of the concurrent program (the *anarchic semantics*) can be specified by the axiomatic constraints of the memory model that filter out executions inconsistent in particular architecture (the *analytic semantics*) [**alglave2016syntax**]. The anarchic semantics of the program is a truly parallel semantics with no global time that describes all possible computations with all possible communications. However, the analytic semantics captures the program behaviours on a certain execution environment more precisely.

2.1 The event-based program representation

The classical approach for analysing concurrent programs is to model it as the set of sequentially consistent programs, obtained by enumerating all possible interleavings. These models are deterministic as they include the notion of the *global time*. Although these models are easy to build and analyse, the number of all possible interleavings grows exponentially (known as the *combinatorial explosion*), which affects the completeness of an analysis method in general case.

One way to fight the combinatorial explosion is to exclude the global time from the model and treat executions from one equivalence class together in a non-deterministic fashion. For instance, such an equivalence class can be the set of computations performed by a processor locally that do not affect the global state. This idea is used in the *event-based* model, that represents the program as a directed graph of events (the *event-flow graph*) [**alglave2010shared**]. The vertices of such a graph represent *events*

(see Section **??**), and edges represent basic relations (see Section **??**). The graph represents the set of executions (sequences of events; see Section **??**) defined by the non-deterministic guesses of certain relations on some states.

There are three main types of sources of non-determinism in concurrent programs [**musuvathi2008fair**]:

1. *input non-determinism*, which is a standard undecidable problem for all static analysis methods: to resolve the user input, system call from the environment, unresolved function calls, etc.;
2. *scheduling non-determinism*, caused by the interleavings, which in turn are caused by the scheduler activity; and
3. *memory-model non-determinism*, caused by hardware and compiler relaxations.

The event-based program model is able to emulate effectively the second and the third types of sources of non-determinism, while the first one can be coped by standard static analysis methods [**landi1992undecidability**; **SurveySymExec-CSUR18**].

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 $\texttt{loc}(e_m)$, its processor label $\texttt{proc}(e_m)$, and a unique event label $\texttt{id}(e_m)$ [**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 $e_1 = \texttt{load}(r_1 \leftarrow l_2)$; $e_2 = \texttt{store}(l_1 \leftarrow r_1)$. Also, it is important to sepa-

rate the set of initial write events $\mathbb{IW} \subseteq \mathbb{W}$ 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 define the set $\mathbb{C}_g \subseteq \mathbb{C}$ of *guard* computation events (also called as *branching events*), that are evaluated to 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 the 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 $\mathtt{po} \subseteq \mathbb{E} \times \mathbb{E}$, 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 $e_1 \xrightarrow{\mathsf{po}} e_2$.
- The *data-flow* of a program is defined by *communication relations*:
  - the *read-from* relation $\mathtt{rf} \subseteq \mathbb{W} \times \mathbb{R}$ that maps each write event to the read event that reads the value written by write event; and
  - the *coherence-order* relation $\mathtt{co} \subseteq \mathbb{W} \times \mathbb{W}$ 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* $\texttt{sr} \subseteq \mathbb{E} \times \mathbb{E}$. In contrast to the herd tool, PorthosC does not use hierarchy of scopes (depicted as the scope tree); instead, it uses simple labels that indicate which process has produced certain event.

Below we enumerate some derived relations [**alglave2010shared**]:

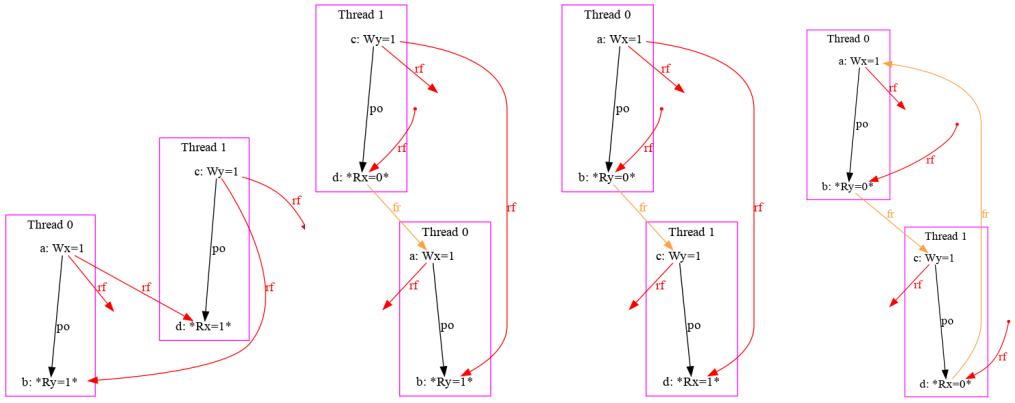
- the *from-read* relation $\texttt{fr} \subseteq \mathbb{R} \times \mathbb{W}$ that maps a read event to all write events succeeding the write event from which the read event gets its value:
  $r \xrightarrow{\texttt{fr}} w \triangleq (\exists w'.w' \xrightarrow{\texttt{rf}} r \wedge w' \xrightarrow{\texttt{co}} w)$;
- the *communication* relation $\texttt{po}$ over memory events, that fully describes the data-flow of a program:
  $m_1 \xrightarrow{\texttt{com}} m_2 \triangleq ((m_1 \xrightarrow{\texttt{rf}} m_2) \vee (m_1 \xrightarrow{\texttt{co}} m_2) \vee (m_1 \xrightarrow{\texttt{fr}} m_2))$;
- the *external* (and *internal*) *from-read* relations that restrict the $\texttt{fr}$-relation to the different (respectively, same) processes:
  $w \xrightarrow{\texttt{fre}} r \triangleq (w \xrightarrow{\texttt{fr}} r \wedge \texttt{proc}(w) = \texttt{proc}(r))$,
  $w \xrightarrow{\texttt{fri}} r \triangleq (w \xrightarrow{\texttt{fr}} r \wedge \texttt{proc}(w) \neq \texttt{proc}(r))$;
- the $\texttt{po-loc}$ relation that is the $\texttt{po}$-relation over events that access to the same shared variable:
  $m_1 \xrightarrow{\texttt{po-loc}} m_2 \triangleq (m_1 \xrightarrow{\texttt{po}} m_2 \wedge \texttt{loc}(m_1) = \texttt{loc}(m_2))$; and
- 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 *execution* is a path in the event-flow graph defined by $\texttt{po}$- and $\texttt{rf}$-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 $\texttt{po}$-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).



(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: *Candidate executions for the litmus test in Example* **??**

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 $\texttt{fr} \cup \texttt{po}$.

2.2 The CAT language

Weak memory models are defined via CAT language [**alglave2016syntax**]. This is a domain specific language for describing consistency properties of concurrent programs. The 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 operations*:

- 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*`, and
- the *inverse* of a relation `r` is $\texttt{r}^{-1}$.

2. *Binary operations:*

   - 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 $\texttt{r1} \setminus \texttt{r2}$, and
   - *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 $\texttt{(x,z)} \in \texttt{r1}$ and $\texttt{(z,y)} \in \texttt{r2}$.

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 of memory model definition in CAT language, the x86-TSO model [**herd10tutorial**] can be found in Appendix **??**. This memory model asserts acyclicity of communication relation, `po-loc` -relation, `mfence`-relation and some other derived relations [**owens2009better**].

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 SMT problem [**Porthos17a**].

3.1 Model checking and reachability analysis

The model checking is the problem of verifying the system (the model) against a set of constraints (the specification) [**dkw2008**]. 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 scheme of model checking is the following. The analysing system (the *model*) is represented as a transition system, a directed graph with labelled nodes representing states of the system. Each state corresponds to the unique subset of atomic propositions that characterise its behavioural properties. Once the model has been constructed, it can be checked for compliance to the *specification*.

Usually, the specification defines temporal constraints over the properties of the system. For instance, the specification assert may state that the property *always* holds (the *safety*) or the property will *eventually* hold (the *liveness*). Commonly, the *Linear Temporal Logic (LTL)* or *Computational Tree*

Logic (CTL) (along with their extensions) is used as a specification language due to the expressiveness and verifiability of their statements.

In the described scheme, 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 modelling 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 fighting the state explosion problem mostly by optimising search space, search strategy or basic data structures of existing algorithms.

One of the first techniques that optimises the search space considerably is the symbolic model checking with *Binary Decision Diagrams (BDDs)*. Instead of processing each state individually, in this approach the set of states is represented by the BDD, a data structure that allow to perform operations on large boolean formulas efficiently [**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 the 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. This technique is called *bounded model checking (BMC)* as the search process is being repeated up to the 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 non-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: $\mathtt{SAT}(Init(\vec{x_0}) \wedge Trans(\vec{x_0}, \vec{x_1}) \wedge ... \wedge Trans(\vec{x}_{k-1}, \vec{x_k}) \wedge Bad(\vec{x_k}))$.

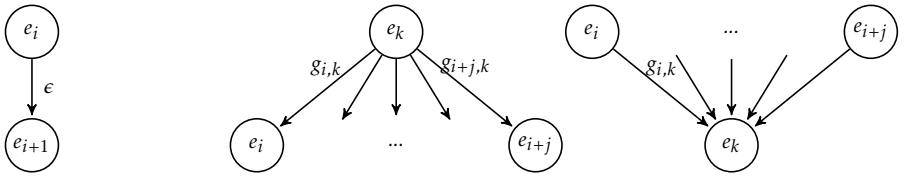
The portability analysis problem may also be stated as a reachability problem, where the undesirable state is one reachable under the target $\mathcal{M}_\mathcal{T}$ memory model and unreachable under the source memory model $\mathcal{M}_\mathcal{S}$. Consider the function $cons_\mathcal{M}(P)$ calculates 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 $\mathcal{M}_\mathcal{S}$ [**Porthos17a**]:

Definition 3.2.1 (Portability)**.** Let $\mathcal{M}_\mathcal{S}$, $\mathcal{M}_\mathcal{T}$ be two weak memory models. The 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*) [**Porthos17b**]. 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 by the *control-flow graph*, a directed acyclic connected graph with a single source and multiple sink nodes. Each control-flow edge contains the label that denotes the *guard*, a predicate determining the transition. The empty guard (a *true* predicate) is denoted as ϵ. A guard depends on the data-flow of the program, it represents a memory unit or a computational expression that is liable to the weak memory model relaxations. The branching expressions that support more than two outgoing control-flow edges may be useful for describing non-deterministic transition systems, where the guards are not necessarily mutually exclusive. However, as the C language supports only binary logic (*if-then-else* branching), PorthosC builds only two possible outcomes of



(a) *Sequence of events (seq)* **(b)** *Conditional branching (br)* **(c)** *Branch merging (mer)*

Figure 3.1: *Possible mutual arrangements of events in a control-flow graph*

evaluating a guard (*primary* and *alternative* transition), see Section **??** for details.

While working on PorthosC, we have applied some modifications to the encoding scheme for the control-flow. These changes were motivated by the need to process an arbitrary control-flow produced by conditional and unconditional jumps of the C language. For that, we compile the *Abstract Syntax Tree (AST)* of the parsed C-code to the plain 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 v1 used 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 encoding 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 **??**). In general, the new encoding scheme follows the one proposed in [**heljanko2008unfoldings**] for encoding Petri-nets.

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). Consider the possible mutual arrangements of nodes in a control-flow graph presented in Figure **??**.

$$\phi_{CF_{seq}} = \quad \mathbf{x}(e_{i+1}) \Rightarrow \mathbf{x}(e_i) \tag{3.1}$$

$$\begin{aligned}\phi_{CF_{br}} = \quad & [\mathbf{x}(e_i) \Rightarrow \mathbf{x}(e_k)] \ \wedge \ \cdots \ \wedge \ [\mathbf{x}(e_{i+j}) \Rightarrow \mathbf{x}(e_k)] \\ & \wedge [\mathbf{x}(e_i) \wedge \mathbf{x}(e_k) \Rightarrow g_{i,k}] \ \wedge \ \cdots \ \wedge \ [\mathbf{x}(e_{i+j}) \wedge \mathbf{x}(e_k) \Rightarrow g_{i+j,k}] \\ & \wedge \ \cdots \\ & \wedge \Big(\bigvee_{e_l \in \ \mathsf{succ}(e_m)} \ \bigvee_{\substack{e_n \in \ \mathsf{succ}(e_k) \\ e_n \neq e_m}} \neg[\mathbf{x}(e_m) \wedge \mathbf{x}(e_n)] \Big)\end{aligned} \tag{3.2}$$

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

For these cases, we propose the encoding scheme that uniquely encodes each node of graph and at the same time allows to encode partially executed program. Equation **??** shows the encoding for 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 **??** shows the encoding for the branching control-flow depicted in Figure **??**, that considers both transitions and guards. Also, adding negations of pairwise conjunctions over all successors of the branching node, the encoding forbids the execution of two branches simultaneously. Equation **??** shows the encoding for the control-flow of a merge-point represented in Figure **??**: the event e_k is executed if either of its predecessors has been executed, regardless the type of the transition. Note that the sequential control-flow is a special case of branching with the only transition guard ϵ (that is encoded as `true`).

For sake of correctness of the encoding, 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 **??**.

As an example of the control-flow encoding, consider the event-flow graph in Figure **??**, which has two branching points and one merge point with three incoming transitions. To illustrate the correctness of the encoding, consider the path $e_1 \rightarrow e_2 \rightarrow e_4$. This means, in the formula ϕ_{CF}, the variables $\mathbf{x}(e_1)$, $\mathbf{x}(e_2)$ and $\mathbf{x}(e_4)$ will have the value 1, and the variable $\mathbf{x}(e_3)$ will be assigned to 0. It is easy to check that the ϕ_{CF} is satisfiable by the chosen SMT-model (considering the guards $g_{1,2}$ and $g_{2,4}$ that define this path to be evaluated to 1).

Note that the generalised encoding scheme does not require the branching transitions to be mutually-exclusive (for instance, consider two branch-

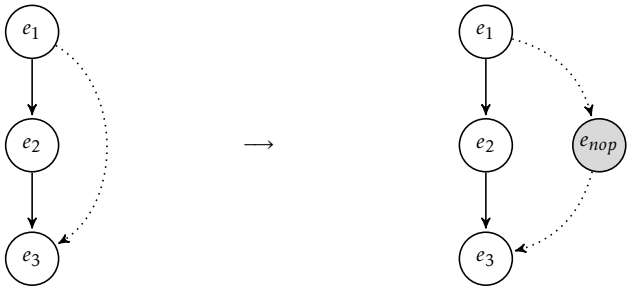
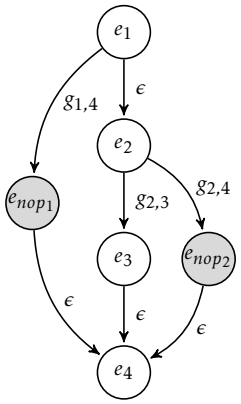


Figure 3.2: *Transformation of the forward-jump control-flow*



$$
\begin{aligned}
\phi_{CF} = {} & [\mathbf{x}(e_2) \Rightarrow \mathbf{x}(e_1)] \\
& \wedge [\mathbf{x}(e_3) \Rightarrow \mathbf{x}(e_2)] \\
& \wedge [\mathbf{x}(e_{nop_1}) \Rightarrow \mathbf{x}(e_1)] \\
& \wedge [\mathbf{x}(e_{nop_2}) \Rightarrow \mathbf{x}(e_2)] \\
& \wedge [\mathbf{x}(e_4) \Rightarrow (\mathbf{x}(e_{nop_1}) \vee \mathbf{x}(e_3) \vee \mathbf{x}(e_{nop_2}))] \\
& \wedge [\mathbf{x}(e_{nop_1}) \wedge \mathbf{x}(e_1) \Rightarrow g_{1,4}] \\
& \wedge [(\mathbf{x}(e_3) \wedge \mathbf{x}(e_2)) \Rightarrow g_{2,3}] \\
& \wedge [(\mathbf{x}(e_{nop_2}) \wedge \mathbf{x}(e_2)) \Rightarrow g_{2,4}] \\
& \wedge \neg[\mathbf{x}(e_2) \wedge \mathbf{x}(e_{nop_1})] \\
& \wedge \neg[\mathbf{x}(e_3) \wedge \mathbf{x}(e_{nop_2})]
\end{aligned}
$$

Figure 3.3: *Example of encoding for the control-flow of the event-flow graph*

ing transitions from the event e_2, both labelled by non-epsilon guards $g_{2,3}$ and $g_{2,4}$). Next, consider the path $e_1 \rightarrow e_3 \rightarrow e_4$, which is not allowed by the control-flow graph. The corresponding model $\mathbf{x}(e_1) = 1$, $\mathbf{x}(e_2) = 0$, $\mathbf{x}(e_3) = 1$ and $\mathbf{x}(e_4) = 1$ does not satisfy the formula ϕ_{CF}. The proposed encoding for the control-flow works also for encoding the partial graph. For example, the SMT-model $\mathbf{x}(e_1) = 1$, $\mathbf{x}(e_2) = 1$, $\mathbf{x}(e_3) = 0$ and $\mathbf{x}(e_4) = 0$ encodes the path $e_1 \rightarrow e_2$, satisfies ϕ_{CF}.

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 the 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 of merge points [**Porthos17a**].

Following [**Porthos17b**], 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:

$$\phi_{DF_{e=\texttt{load}(r\leftarrow l)}} = [\mathbf{x}(e) \Rightarrow (r_{i+1} = l_{i+1})] \tag{3.4}$$

$$\phi_{DF_{e=\texttt{store}(l\leftarrow r)}} = [\mathbf{x}(e) \Rightarrow (l_{i+1} = r_i)] \tag{3.5}$$

$$\phi_{DF_{e=\texttt{eval}(\cdot)}} = [\mathbf{x}(e) \Rightarrow \mathbf{v}(e)] \tag{3.6}$$

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 one graph traverse.

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

The basic scheme for encoding the memory model is proposed in [**Porthos17a**]. The encoding consists of two parts: encoding the *derived relations* and encoding the memory model *assertions*.

In an SMT-formula, the relation $x \xrightarrow{\mathsf{r}} y$ is represented by a boolean variable $r(x, y)$ that indicates whether the relation holds. The derived relations are encoded by fresh boolean variables according to the following rules [**Porthos17b**]:

- $r_1 \cup r_2(e_1, e_2) = r_1(e_1, e_2) \vee r_2(e_1, e_2)$;
- $r_1 \cap r_2(e_1, e_2) = r_1(e_1, e_2) \wedge r_2(e_1, e_2)$;
- $r_1 \setminus r_2(e_1, e_2) = r_1(e_1, e_2) \wedge \neg r_2(e_1, e_2)$;
- $r^{-1}(e_1, e_2) = r(e_2, e_1)$;
- $r^*(e_1, e_2) = r^+(e_1, e_2) \vee (e_1 = e_2)$;
- $r_1; r_2(e_1, e_2) = \bigvee_{e_k \in \mathbb{E}} r_1(e_1, e_k) \wedge r_2(e_k, e_2)$; and
- $r^+(e_1, e_2) = \mathtt{tc}_{\lceil log|\mathbb{E}| \rceil}(e_1, e_2)$, *where*
  $\mathtt{tc}_0(e_1, e_2) = r(e_1, e_2)$, *and*
  $\mathtt{tc}_{i+1}(e_1, e_2) = r(e_1, e_2) \vee (\mathtt{tc}_i(e_1, e_2); \mathtt{tc}_i(e_1, e_2))$.

Note that CAT language allows mutually-recursive definitions of relations (for example, '$\mathtt{r}_1 = \mathtt{r}_2 \cup (\mathtt{r}_1; \mathtt{r}_1)$'). The basic idea of using the Kleene fixpoint iteration for encoding such relations was also proposed in [**Porthos17a**]: for any pair of events $e_1, e_2 \in \mathbb{E}$ and relation $r \subseteq \mathbb{E} \times \mathbb{E}$, we

encode a new integer variable $\Phi^{r}_{e_1,e_2}$ that represents the round of Kleene iteration on which the variable $r(e_1, e_2)$ has been set.

The memory model can assert acyclicity, irreflexivity of emptiness of a relation or a set of events. As it has been proposed in [**Porthos17a**], encoding the acyclicity assertion uses numerical variable $\Psi_e \in \mathbb{N}$ for each event e in the relation to be asserted: $acyclic(r) = (r(e_1, e_2) \Rightarrow (\Psi_{e_1} < \Psi_{e_2}))$. The irreflexivity assertion as $irreflexive(r) = \bigwedge_{e_k \in (E)} \neg r(e_k, e_k)$.

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 the 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 the C language is supported (which, considering its complexity and numerous pitfalls, goes far beyond current thesis[1]), we consider the accomplished work as a step towards it.

4.1 General principles

The previous version of Porthos did not distinguish the event-based program model from the high-level AST, they both were encoded into a single SMT-formula (see classes of package '`dartagnan.program`' of Porthos v1). 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

[1] To ensure this, one merely has to look at existing C compiler implementations, 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`)

the program) and extend (since adding the support for a new high-level 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 (an AST structure) from the low-level event-based representation (an event-flow graph). Such a modular architecture will allow to support multiple input languages[2] by parsing them and converting parsed syntax trees to a simplified AST.

All internal representations used by PorthosC 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[3]).

During the development of PorthosC, 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 PorthosC:

1. *Robustness:*
   - 1.1. preservation the completeness of analysis,
   - 1.2. modular architecture: each module can be tested independently,
   - 1.3. usage of software design patterns where necessary, and
   - 1.4. 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, and
   - 2.3. following the clear code style.
3. *Efficiency:*
   - 3.1. keeping the trade-off between execution time and memory usage.
4. *Extensibility:*
   - 4.1. clear modular architecture.

[2] Apart from the C language, PorthosC must be able to analyse litmus tests in different assembly languages and, as a compatibility mode, the input language of Porthos v1.

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

Robustness of the analysis is the main criterion of PorthosC as a verification tool. Although it makes the analysis more sensitive to the combinatorial explosion, it preserves the completeness of analysis, which is necessary for a model-checking tool. As a robust and transparent tool, PorthosC must adhere to the strategy of aborting its work on any unexpected outcome (for instance, if a parser failed to parse the string and the recovery algorithm is not described). One of the transparency principles is following the clear code style. This means mainly clear and informative naming of classes, functions and fields. It also means unified ordering of methods in classes, minimised size of methods code, clear tabulation, etc.

As its predecessor, PorthosC uses the open-source SMT solver Z3[4] by Microsoft Research [**de2008z3**]. However, unlike its predecessor, PorthosC has an additional abstraction level, Z-formula (see Section **??**), that allows the use any other SMT solver.

As its predecessor, PorthosC is written in *Java*, firstly, in order to be able to reuse some parts and concepts of Porthos also 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 the best results in performance benchmarks (for example, compared to C++ [**hundt2011loop**]), 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 solver invoked by PorthosC via a 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, 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]The Z3 project repository: `https://github.com/Z3Prover/z3`

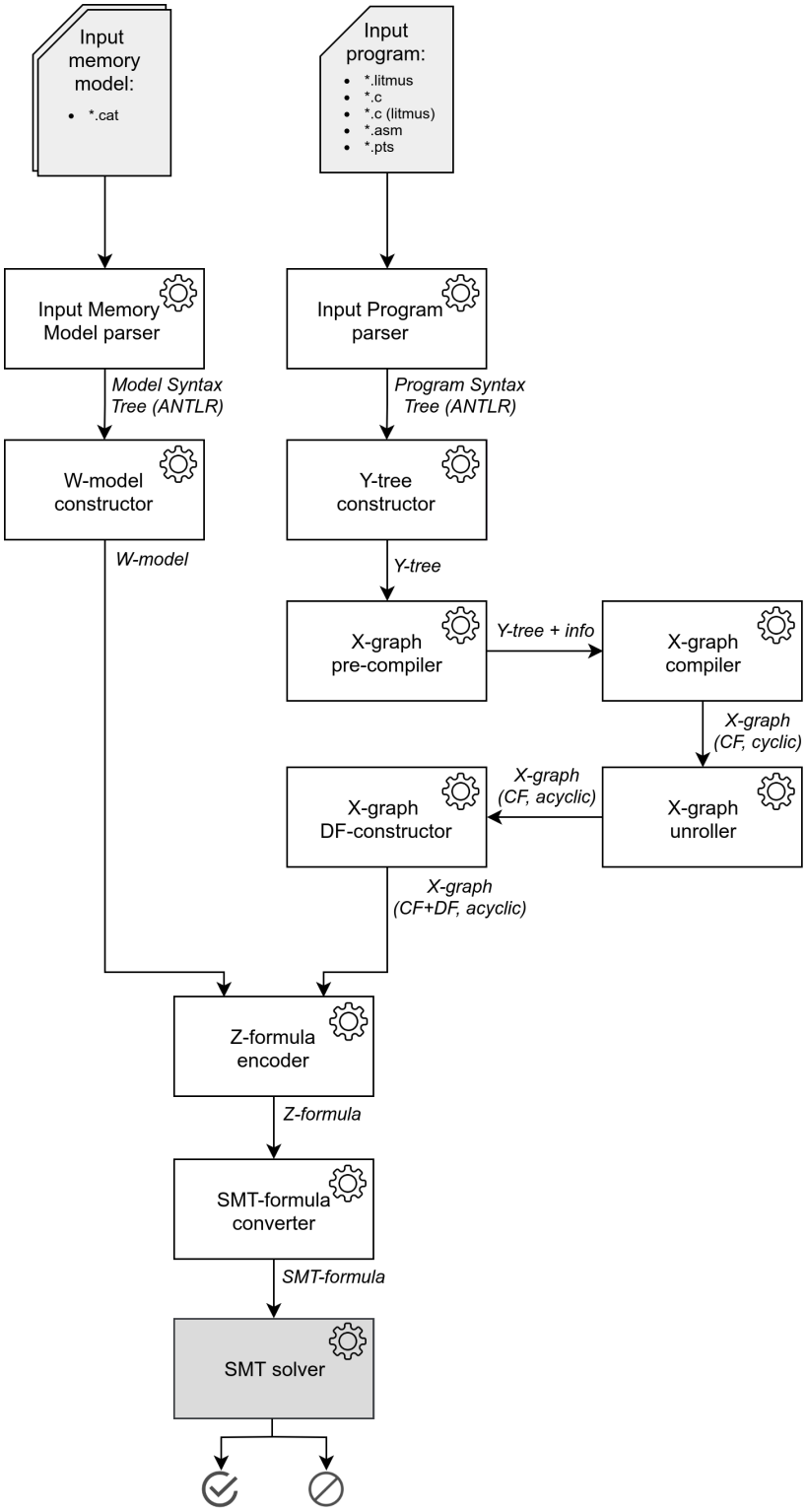


Figure 4.1: *The general architecture of PorthosC*

4.2 The architecture

The general architecture scheme of PorthosC is presented in Figure **??**.

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[5](Section **??**), which then is being preprocessed 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 is being converted to an acyclic form (Section **??**) in order to be encoded into a Z-formula (Section **??**). Apart from that, the memory-model constructor (Section **??**) constructs the abstract syntax tree of derived relations of the weak memory model W-model (Section **??**). Thereafter, W-model and acyclic X-graph are encoded (Section **??**) to a Z-formula representation (a wrapper over an SMT-formula), which then is translated to an SMT-formula (Section **??**), which then is solved by the SMT solver (Section **??**).

4.2.1 Program input

Both Porthos and PorthosC use the ANTLR parser generator[6] [**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 not be as efficient as a hand-written language-optimised parser, it reduces the overhead of implementing the parser significantly. Among other advantages ANTLR, it has a 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 grammar sketch in BNF syntax of the input language used by Porthos v1. The input language parser used by Porthos

[5]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 **w**eak memory model AST.

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

```
⟨program⟩
   :  ⟨initialisation⟩ ⟨thread⟩+ ⟨assertion⟩
   ;
⟨thread⟩
   :  thread ⟨thread−id⟩ ⟨instruction⟩
   ;
⟨instruction⟩
   :  ⟨atom⟩
   |  '{' ⟨instruction⟩ '}'
   |  ⟨instruction⟩ ';' ⟨instruction⟩
   |  'while' '(' ⟨bool−expr⟩ ')' ⟨instruction⟩
   |  'if' ⟨bool−expr⟩ '{' ⟨instruction⟩ '}' ⟨instruction⟩
   ;
⟨atom⟩
   :  ⟨register⟩ '<−' ⟨expression⟩
   |  ⟨register⟩ '<:−' ⟨location⟩
   |  ⟨location⟩ ':=' ⟨register⟩
   |  ⟨register⟩ '=' ⟨location⟩ '.' 'load'  '(' ⟨atomic⟩ ')'
   |  ⟨location⟩ '=' ⟨register⟩ '.' 'store' '(' ⟨atomic⟩ ')'
   |  ('mfence' | 'sync' | 'lwsync' | 'isync')
   ;
⟨bool−expr⟩
   :  'true'
   |  'false'
   |  ⟨expression⟩ ('and'  |  'or') ⟨expression⟩
   |  ⟨expression⟩ ('==' | '!=' | '⟩' | '⟩=' | '<' | '>=') ⟨expression⟩
   ;
⟨expression⟩
   :  [0−9]
   |  ⟨register⟩
   |  ⟨expression⟩ ('∗' | '+' | '−' | '/' | '%') ⟨expression⟩
   ;
```

Figure 4.2: *The sketch of the input language grammar used by Porthos v1*

suffered from several disadvantages. Firstly, it contained the parser code inlined directly into 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 expressive languages makes the code hardly understandable and, therefore, poorly maintainable. In PorthosC, we clearly separated the parser (generated from the grammar file '<*grammar*>*.g4*') from converting the ANTLR syntax tree to the AST, that is one for all languages of an input program.

Secondly, Porthos resolved the semantics of operations syntactically (it was defined in the ANTLR grammar), whereas it should be resolved by a separate module operating on the AST level, so that it does not require to change grammar for encoding the semantics of a new function. As the reader may have noticed from the grammar sketch in Figure **??**, the memory operations of different kinds vary syntactically as well. For example, the

assignment of local computation to a register uses the symbol '<-', the atomic non-relaxed load operation denoted as '<:-', atomic non-relaxed store operation denoted as ':=', and the semantics of relaxed `load` and `store` are resolved syntactically by matching the function name. Moreover, only the operator '<-' could have an expression as the source of data, which means that expressions could be assigned only to registers. 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 **??**). The semantics of the functions also is being resolved during the pre-compilation stage via the *invocation hooking* mechanism (see Section **??**).

Thirdly, the grammar used by Porthos allowed only a restricted set of operations. For example, it accepted the computation expressions only over local variables. Thus, in the assignment expression '`r <- (x + 1);`', the variable `x` was parsed as a local variable even though it can be used as a shared variable in other parts of the program. In PorthosC, all shared variables involved into a computation expression are tentatively copied to temporary local variables.

Fourthly, Porthos v1 supported only integer constants and expressions. In PorthosC, we extended support for primitive types supported by the Z3 solver (this apply to 32-bit integers encoded as `Int`s of Z3, floats encoded as `Real`s, enumerations encoded as `Scalar`s). Although the Z3 supports the array theory (characterised by the select-store axioms [**de2011z3**]), the complexity of pointer analysis for the arrays of non-constant size moves the full support of arrays and pointers out of the scope of current thesis.

Finally, the grammar used by Porthos had the following minor drawbacks. The operators were implemented as non-associative (expressions of the form '`1 + 2 * 3`' could not be parsed). Comparing to C statements, that must end with the semicolon punctuator ';', statements defined in the grammar of Porthos v1 use the semicolon as a separator between statements (the final statement must not end with semicolon). The litmus-specific syntax for variables initialisation was used only for declaring the shared variables (all of them were initialised with default value `0`), however, this syntax should be used as an initial assignment of both shared and local variables with arbitrary values.

PorthosC 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[7]). 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 can be easily extended to support the inter-procedural (cross-procedure) analysis that inlines function calls and binds variable contexts. Thus, instead of analysing a single source code file, the user can analyse the whole code project. In this mode, the functions to be executed in parallel should be specified by the user. Also, the tool may be extended to detect the concurrent parts of the code automatically at the pre-compilation stage. For that, the pre-compiler should recognise functions that commence a new process (such as `pthread_create` from `pthread.h`) and resolve the argument that points to the thread function. This functionality is left beyond current thesis.

4.2.2 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 high-level recursively[8] defined AST. The Apendix **??** presents the file tree of the 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 the case of processing low-level assembly code, it may be directly converted to the X-graph representation). In terms of Porthos v1, the Y-tree is the level of *instructions*.

[7] ANTLR grammars repository path: `https://github.com/antlr/grammars-v4`

[8] Hereinafter we use the term *recursive* data structure (sometimes called *inductive*) to refer a complex data type which can contain elements that contain other elements of the same type. For example, an unary expression contains references to the operator and the child expression of the same type.

However some details of the syntax might have been abstracted away (for instance, array operations may be emulated by functions invocations, see [**gries2012science**]), we find this level of abstraction suitable enough for modelling a high-level language.

Each Y-tree element implements the interface `YEntity` and carries the `OriginLocation` instance that contains information about the coordinates of the input text that has 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). Each Y-tree expression carries the original code coordinates, that can be resolved to text by the service `LocationService`.

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); and
- 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; and
- 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

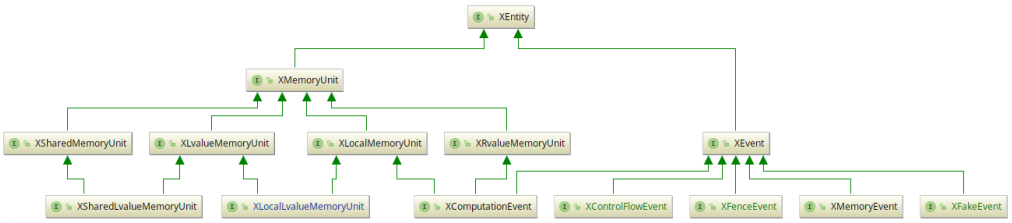


Figure 4.3: *The inheritance tree of interfaces of X-graph*

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 **??**. 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$_{CF}$ $\cup$ X-graph$_{DF}$. The UML diagram on Figure **??** represents the hierarchy of main interfaces of the X-abstraction level. Internally, the graph is represented by an adjacency matrix (to be exact, by multiple adjacency matrices that store edges of different kinds, see more details in Section **??**).

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.

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 the 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

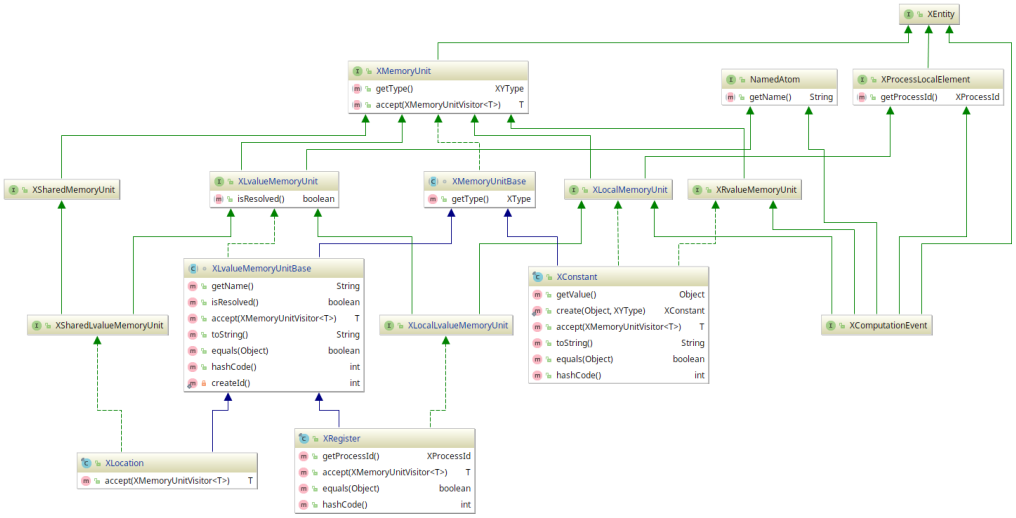


Figure 4.4: *The inheritance tree of X-graph memory units*

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 **??**.

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 convertible to an SMT-type (modelled as ZType).As the type of an X-memory unit may not be resolved correctly, the memory units keep this information as a boolean flag.

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 **??**.

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. Also, each event carries the reference to the Y-instruction that has generated it.

The following interfaces model basic kinds of events (see 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):
  - XRegisterMemoryEvent:
    (XLocalLvalueMemoryUnit) ← (XLocalMemoryUnit),
  - XLoadMemoryEvent:
    (XLocalLvalueMemoryUnit) ← (XSharedMemoryUnit),
  - XStoreMemoryEvent:
    (XSharedLvalueMemoryUnit) ← (XLocalMemoryUnit), and
  - XInitialWriteEvent:
    (XLvalueMemoryUnit) ← (XRvalueMemoryUnit).
- XComputationEvent
  We distinguish two types of computation events:
  - XUnaryComputationEvent that encodes bit negation and no-operation; and

[9] Here we should note that 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 **??**.

  - `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; and
  - `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. The call stack is bounded by the user-defined parameter; once the stack is full,

the interpretation continues without jumping to the body of the invoked function (such cases are properly logged). Each `return` statement creates the assignment of the function call event on the top of call stack. Note, this approach works for any kind of recursion, which is unrolled up the user-defined boundary as well as explicit loops.
According to the Rice's theorem [**rice1953classes**], in general case there always will be functions unresolved within an analysis launch. If the function has not been resolved, it can be safely assumed to be a no-operation function. In other words, we suppose that the knowledge base of PorthosC is complete and the tool can resolve all memory-operation functions and fence instructions. Although this can affect the completeness of the analysis, it is safe to make such an assumption as the user can check the log file and manually analyse the semantics of an unresolved call and it to the knowledge base if necessary.

- `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 the auxiliary elements of X-graph.

  - `XEntryEvent`, the per-process unique source event in the event-flow graph,
  - `XExitEvent`, the process sink event,
  - `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 **??**), and
  - `XAssertionEvent`, the reachability assertion made at the postlude statement of the program. An assertion is modelled as an event for the purpose of encoding the postlude statement as a separate process that is compatible with the encoder.

Edges. As the graph is represented by an adjacency matrix, its edges are stored in (immutable) hash-maps. We distinguish the following kinds of edges:

- the *control-flow edges*:
  - the *primary edges*, that denote both ϵ-labelled transitions (in case of linear sequence of events) and conditional transition that evaluates the conditional event (the source of the transition) to the *true*, and
  - the *alternative edges*, that denote conditional transitions for which the conditional event (the source) was evaluated to the *false*; and
- the *data-flow edges*:
  - the `co`-relation edges, and
  - the `rf`-relation edges.

Graph invariants. Once being constructed, the graph must conform the following requirements (see also Section **??**):

1. the graph must have a single source with no ingoing edges, and two sinks of different kinds without outgoing edges,
2. the graph must be connected,
3. each node of the graph can have either one or two direct control-flow successors,
4. only nodes of type `XComputationEvent` can have two direct control-flow successors,
5. a `co`-edge connects two writes, an `rf`-relation edge connects a write and a read,
6. all write-event nodes except initial write-nodes must have exactly one `co`-predecessor, and
7. all write-event nodes except final write-nodes must have exactly one `co`-successor.

4.2.2.3 W-model

The *W-model* represents a recursive AST of *computations over relations* and assertion expressions defined by the memory model. The atomic elements of W-model are the basic relations (`po`, `rf` and `co`; see Section **??**) and sets of events ($\mathbb{R}$, $\mathbb{W}$, $\mathbb{IW}$, etc.; see Section **??**). The expressions of W-model are unary (such as complement, transitive closure, etc.) and binary operations (such as union, intersection, etc.) over relations or sets of events; see Section **??**. For the sake of transparency, each element of a W-model

contains the origin location as the coordinates of the string in the model file.

4.2.2.4 Z-formula

The *Z-formula* representation is a wrapper for an SMT-formula used as an additional abstraction level used to increase the transparency of the architecture, simplify the debugging process and ease the support of different SMT solvers. As all other internal representations, the Z-formula is implemented as an immutable data structure.

The Z-abstraction level models the the logical formulas (definitions and assertions) that are put on the assertion stack of the SMT solver. Generally, a Z-formula represents the S-expression-based syntax of the SMT-LIB language [**smt-lib**]. Currently, the Z-formula supports definitions and assertions over *variables* and binary and unary *expressions*. However, in future it may be extended to support *function symbols* and *binders* (existential and universal quantification, pattern matching and functional type construction).

All expressions of a Z-formula are *typed* (or *sorted*, in terms of SMT-LIB standard). Basically, we distinguish *boolean* and *numeric* (integer, bitvector, real) expressions. The typing of a Z-formula is necessary for checking the basic validity of its expressions and converting it to a typed SMT-LIB formula. For the sake of transparency, each element of a Z-formula contains the origin location as the reference to the X-element that produced current Z-element.

4.2.3 Processing units

This section describes program units that construct, transform and analyse internal representations described before. We assign unique numbers to the processing units, which can checked in Figure **??** (within gears depicted in the top-left corner of each processing unit).

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

Most data structures are processed by units that implement the *visitor* pattern [**palsberg1998essence**]. 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 general structure of the visitor pattern is illustrated by the pseudo-java code in Figure **??**. The visitor pattern performs the double-dispatching, a mechanism for decreasing the cohesion between the DTO (the visitee) and the processor class (the visitor): the visitee implements the *accepting* method that gets the visitor as an argument and invokes its *visiting* method with itself as an argument. Thus, the method call resolution is performed statically at compile-time without any overhead at run-time. In our implementation, the visitor (not the visitee) cares about the continuation of traversing its children.

```
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 4.5: *Illustration of the visitor pattern*

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 graph data structures, we mostly follow the *iterator* pattern: the special object `Iterator`, that has access to elements of the data structure, iterates over them in some order (for example, in topological order as it is implemented for X-graph). The construction of immutable data structures is performed by units that follow the *builder* pattern. A

builder is a mutable object that contains methods "filling" it by the elements that will constitute the result complex object. The methods contain basic validity checks of the elements. Once a builder has been built, it cannot be modified.

4.2.3.1 Input parsers

Both input-language and input-model parsers are implemented via ANTLR parser generator. Details on the input program language grammar are discussed in Section **??**. 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 **??**. The ANTLR grammar for CAT language was extracted from the parser used by herd tool[10] written in OCaml.

4.2.3.2 W-model constructor

The W-model representation is constructed by the stateless visitor `Cat2WmodelConverterVisitor` from the ANTLR syntax tree `CatParser.MainContext` of the memory-model defined in CAT language. Currently, the visitor supports the small subset of CAT language, which constitutes only non-functional declarative expressions of the language as the support for functional-style expressions requires implementing the full OCaml interpreter. However, some most commonly used functional-style expressions may be supported by mapping them syntactically to corresponding W-elements by the W-model constructor.

Following the principle of transparency, the W-model constructor aborts its work with the `NotSupportedException` if met the unrecognised syntax construction (in contrast to the Porthos v1 approach in which the `null` value was produced in all exceptional states).

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

4.2.3.3 Y-tree constructor

The Y-tree is constructed by the stateless visitor `C2YtreeConverterVisitor` from the ANTLR syntax tree `C11Parser.MainContext` of the C language. As the W-model constructor, the Y-tree constructor aborts its work with the `NotSupportedException` once it meets an unsupported syntax construction. A syntax exception `YConverterException` is thrown if the converted syntax tree contains semantic errors that prevent it to be converted to the Y-tree.

As a Y-tree constitutes a generic AST, the Y-tree constructor expands the syntactic sugar expressions and statements (for example, a `switch`-statement is converted to the equivalent `if`-statement).

4.2.3.4 X-graph pre-compiler

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

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.

Type analysis. 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 **??**).

Currently, PorthosC handles only the primitive C types (such as `int`, `char`, `float`, etc.), which is modelled as an enumeration `XType`. The array dimension is stored at the Y-level as an integer number (currently this information is not used as PorthosC does not support arrays, although this information can be used at the stage of pre-compilation for converting array elements to separate variables). 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.

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 variables*:

- 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; and
- 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).

4.2.3.5 X-graph compiler

The X-compiler is the main component that transforms the recursive Y-tree data structure to the plain X-graph representation. It is a complex processing unit; Figure **??** illustrates the relationship between the principal components of the X-compiler in the UML language.

The main class representing the X-compiler is `Y2XConverter`. It receives as input the Y-tree, the memory model kind and user settings (for instance, the interpreter mode defining the set of invocation hooks enabled during the

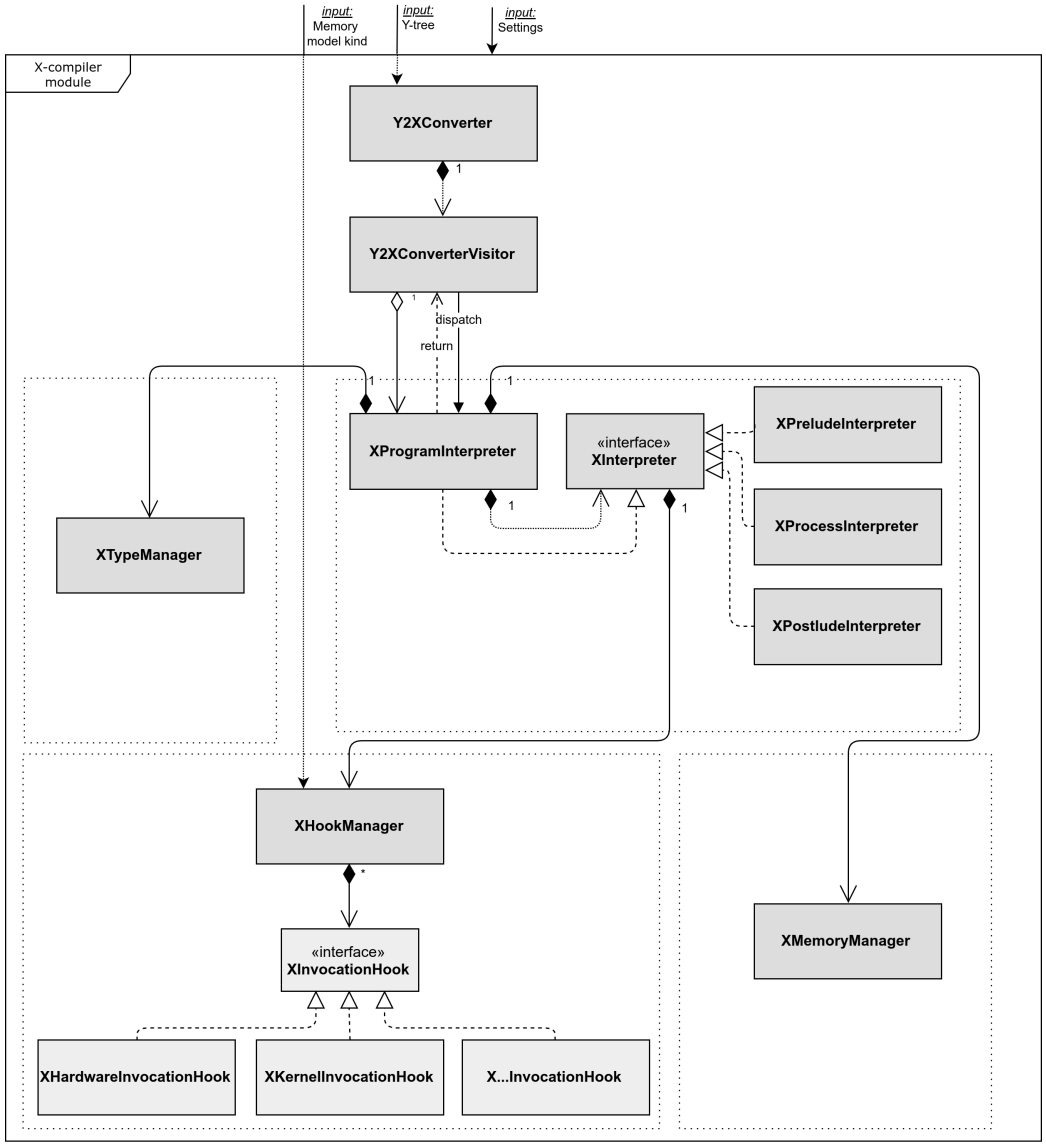


Figure 4.6: *Main components of the X-compilation processing unit*

analysis run). The `Y2XConverter` creates an instance of the stateless visitor `Y2XConverterVisitor` that traverses the Y-tree while invoking the stateful interpreter `XInterpreter`. The `XInterpreter` carries the X-graph-builder and provides *action* methods for changing its state and thus the filling in the builder. The interpreter requests the managers such as `XMemoryManager`, `XHookManager`, `XTypeManager` for additional information.

We distinguish three types of processes (each implementing the `XInterpreter` interface):

- the prelude process `XPreludeInterpreter` (to be executed before all other processes) that allows only declaration of local or shared variables, computations and memory operations;
- the postlude process `XPostludeInterpreter` (to be executed after all other processes) that allows only declaration of local variables, memory operations, computations and assertions;
- the process `XProcessInterpreter` (to be executed in parallel) that allows all X-compiler interface operations except declaration of shared variables and program assertions (i.e., it allows declarations of local variables, computations, barriers, unconditional jumps, non-linear branching statements, and method calls).

As the prelude process can declare shared variables, its compilation must go first. The first event of each process must be the `XEntryEvent` and the last events must be events of type `XExitEvent`.

Memory manager. The X-graph abstract machine model disposes infinitely many memory units, both local and shared (see Section **??**). 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.

The interface methods exposed by the `XMemoryManager` are presented in Figure **??**. 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 (the method `getDeclaredUnitOrNull` returns the first memory unit found, either a global one or a local one, or null if no memory units with requested name have been registered). Since C language allows usage of variables that have the same name to be declared in nested contexts, the `XMemoryManager` carries the stack of block contexts for local variables.

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 the case of program in C language – only the function name) across all signatures it stores. If the signature

```
interface XMemoryManager {
  XLocation declareLocation(String name, XType type);

  XRegister declareRegister(String name, XType type);

  XRegister declareTempRegister(XType type);

  XLvalueMemoryUnit declareUnresolvedUnit(String name, boolean global);

  XLvalueMemoryUnit getDeclaredUnitOrNull(String name);

  XRegister getDeclaredRegister(String name, XProcessId processId);
}
```

Figure 4.7: *X-memory manager public interface*

matches, the hook manager intercepts the function call and interprets the hook action instead of invoking the function directly. The hook manager operates multiple invocation hooks depending on the user-defined mode. All invocation hooks implement the interface `XInvocationHook`. The result of an invocation hook interception is the `XInvocationHookAction`, a delayed operation implemented on the top of lambda functions of Java. The hook action is invoked with actual arguments and returns an arbitrary `XEntity` as the result of invocation.

Current version of PorthosC contains two invocation hooks, the `XPorthosInvocationHook` for describing the compatibility-mode functions of the input PorthosC input language, and `XKernelInvocationHook` for describing the Linux kernel-specific functions. An invocation hook can model any type of operations (memory, fence, computational, etc.). For instance, the `XPorthosInvocationHook` intercepts calls `'location.store(atomic,register)'` and processes the store operation with respect to the specified atomic.

Interpreter. The X-program interpreter is invoked by the `Y2XConverterVisitor` which walks down the recursive syntax tree Y-tree. The calls to the X-program interpreter are dispatched to currently maintained X-process interpreter. To be able to properly recognise nested statements of the Y-tree, the X-process interpreter needs to have stacks. On any semantic error, the X-interpreter throws an `XInterpretationError`.

The interface methods of the `XInterpreter` are presented in Appendix **??**. Note the clear modular independence of the X-interpreter as it has no methods that operate entities of the Y-level (all conversion work with Y-entities is done by the `Y2XConverterVisitor`).

All interface methods can be divided into two groups. The first group constitute methods that emit events. These methods construct an event object (as events contain the `XEventInfo` structure that stores information about the owning process, all events must be created by the process constructor, i.e. X-interpreter) and change the state of interpreter considering the newly created event. Note that the methods `createComputationEvent` do not emit a computation event but only create one. This is performed as optimisation that removes unused computations from the model (otherwise, for instance, the assignment '`x = 1 * (2 + 3)`' would be compiled to two consequent events '`eval(2 + 3); write(register <- eval(1 * eval(2 + 3)));`' instead of a single event '`write(register <- eval(1 * eval(2 + 3)));`'). If the computation event was not used at all (for example, in the following C code: '`foo(); 1; bar();`' the execution event '`eval(1)`' is skipped by the model), it is also removed from the event-graph (see justification in Section **??**). The second group of X-interpreter methods consists of the methods for defining non-linear statements (branchings and loops). These methods change the state of the interpreter and set up additional non-linear control-flow edges.

As a high-level instruction (Y-level) may be compiled into a sequence of low-level instructions (for example, a computation that involves shared variables should firstly load them into the local memory and then process the computation event over local-only memory), the interpreter must maintain the *stack of contexts* and remember the *previous event* to be able to correctly process nested non-linear statements of C language. The context is a data structure that carries the *state* and some additional information in the case of processing non-linear statements (e.g., conditional event, first and last then- and else-branch events for binding, etc.). Once the new event has been emitted, the interpreter sets up control-flow edges w.r.t. state of the context on the top of the stack (the context stack is always non empty: the bottom context is a linear one). The context state is an enumeration of the following values:

- `WaitingAdditionalCommand`: the interpreter is in the state of defining the complex (non-linear) statement and is not able to process any new event (will throw an exception if any) until the state is not changed;

- `WaitingFirstConditionEvent`: the first next emitted event will be accepted as the first event of the condition evaluation (later, the loop edges will be set to this event);
- `WaitingLastConditionEvent`: the next event must be of type `XComputationEvent`; it will be saved as the conditional branching event;
- `WaitingFirstSubBlockEvent`: the first next emitted event will be accepted as the first event of the branching statement (later the interpreter will set up jumps to that event);
- `WaitingNextLinearEvent`: the standard interpreter state for processing next linear event, and
- `Idle`: the interpreter does not set up the edge from the previous event to the new event.

Each new event emitted by the interpreter is processed considering the state of the context stack: the interpreter iterates over the context stack and sets up the edges by the graph builder depending on the state of each stack. The state `WaitingFirstConditionEvent` is necessary for correctly interpreting the branching conditions, which shared variables are involved in. If the non-linear statement is a loop statement, the loop back-edges will be set to this event.

Once interpretation of the (non-linear) branch is completed, its non-linear context is being popped out of the context stack (by interpreter method `startBlockBranchDefinition`) and added to the queue of `almostReadyContexts`. The almost-ready-context becomes a ready-context (i.e., it moves to the queue `readyContexts`) once the non-linear statement definition has finished (the method `finishNonlinearBlockDefinition`).

Before processing a newly emitted event, the interpreter checks whether the queue `readyContexts` is not empty. If so, it iterates over all ready-contexts and sets up control-flow edges considering that the newly emitted event is an exit-event" of the non-linear context (e.g., for a branching context the interpreter adds primary edges from condition-event to the first-true-branch-event and from the last-true-branch-event to the exit-event, the same is done with alternative edges for the false-branch).

For the sake of simplicity of the interpreter, jump events (no-computation events) are present in the flow-graph, however they can be removed from it with rebinding ingoing and outgoing edges.

4.2.3.6 X-graph unroller

In order to be encoded into an SMT-formula, the compiled graph needs to be acyclic [**Porthos17b**]. To convert it to an acyclic form, we perform the *unrolling* (sometimes called *unwinding*) transfomation: each cycle is unrolled up to the user-defined bound k. Comparing to Porthos v1, we have changed the meaning of the bound: instead of executing all cycles at most k times, the PorthosC interprets the bound as the maximum number of events in the trace of the unrolled graph[11] . The meaning of a bound was changed in order to increase predictability of the size of the result SMT-formula. Not that recursive function calls create unconditional jumps back in the event-flow graph, which will also be recognised as a loop and be affected by the unrolling algorithm.

Considering the new meaning of the unrolling bound, the graph unrolling may be *complete* (the loop has been executed a whole number of times) or *incomplete* (the unrolling bound has triggered on not-the-last event of the loop), which can be modelled by two types of the sink events. The user may need this information for understanding how the PorthosC interprets the cyclic program. However, current implementation loses this information by having only one sink event. Figure **??** illustrates the unrolling for the left-hand side cyclic control-flow graph (the square node S_+ denotes the complete sink node, and the S_- denotes the incomplete sink).

The unrolling procedure is performed by the `XFlowGraphUnroller`. For unrolling the flow-graph, we perform the *Deep-First Search (DFS)* while counting the Y-level instructions (for the unrolling bound) and keeping track of the depth stack (for detecting back edges needed for determining the type of sink node). Also, during the unrolling each next event increments the *unrolling depth counter* (the *event reference-ID*) – the integer that is stored by the `XEvent` thus making it *an event reference* on non-zero values. Each `XEvent` has the method '`XEvent asNodeRef(int refId)`' that clones

[11] The original specification of PorthosC stated that the unrolling bound k must be interpreted as the maximum number of instruction in the original code (technically, the number of expressions in the ANTLR syntax tree). Nonetheless, current implementation of the X-graph unroller counts the X-level events as it is much simpler to implement (the questions about how to count complex expressions that involve multiple shared variables and method invocations are left open for the future versions of PorthosC). For now, the information about the element of the ANTLR tree that corresponds to the X-event or Y-tree element is being lost during the transformations. Instead, we use the `LocationService` discussed in Section **??** that in perspective may be extended for providing this information along with the text citation of the original code.

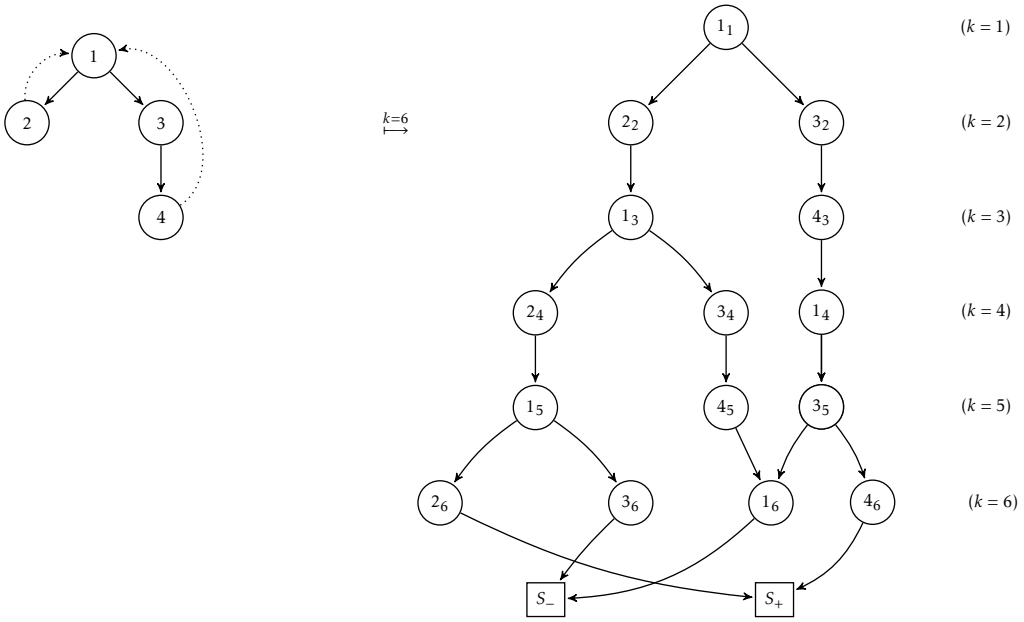


Figure 4.8: *Example of the flow graph unrolling up to bound $k = 6$*

the event-receiver with new value of reference-ID. The methods `hashCode` and `equals` consider the reference-ID as the uniqueness field for all events except sink and source events (considering the usage of `HashMap` and Guava's `ImmutableMap` for storing events, proper setting of the hash-code methods is a crucial programming task).

4.2.3.7 X-graph data-flow constructor

Once the graph is unrolled, it can be augmented by data-flow edges. As it was discussed in Paragraph **??**, the data-flow edges can represent either `rf`- or `co`-relation. The `rf`-edges join each write event to all read events that access the same location; the `co`-edges join write events to the same location.

4.2.3.8 Z-formula encoder

The Z-formula encoder transforms the program and the memory models to a single SMT-like representation. It follows the encoding scheme described in Chapter **??**.

4.2.3.9 SMT-formula converter

As the Z-formula follows the SMT-LIB standard, its translation to an SMT-formula constitutes almost one-to-one mapping. The translation is performed by the stateless visitor `Z2SmtTranslator`.

4.2.3.10 SMT solver

As it has been mentioned in Section **??**, PorthosC uses the third-party SMT solver Z3, which offers the Java API for constructing requests to the Z3 solver and interpreting its responses.

4.2.4 Program output

The result of execution of PorthosC is the verdict modelled by the class `AppVerdict`. This is a structure that contains the result of analysis and auxiliary information such as collected errors and time of execution (separately for each stage of computing). The app verdict may be rendered to any format convenient for the user.

Chapter 5

Evaluation

5.1 The PorthosC in work

5.1.1 The X-graph compilation

As a simple example of the compilation process discussed in Section **??**, consider the tiny C test in Figure **??**. This function does not perform any useful computation, however it contains several syntactic elements (such as prefix increment or loop-breaking statements), which are supported by PorthosC comparing to its predecessor. The control-flow subgraph X-graph$_{CF}$ of the non-unrolled event-flow graph is presented on the right-hand side of the Figure (all graphs generated by PorthosC are produced with the help of the open-source library `Graphviz` [**ellson2001graphviz**]).

Each event of the control-flow graph contains the unique number (generated by the `hashCode` method) in curly brackets below the value, which is necessary for correct displaying the graph. Write events are denoted with the left-directed arrow '<-', and the functions `load` and `store` denote the type of the shared memory event. The primary transitions that denote unconditional jumps or if-true-transitions are pictured with solid lines, and the alternative transitions that denote if-false-transitions are pictured with dotted lines. The graph contains a single source event and a single sink event (represented by the grey triangles).

The arguments `x` and `y` of the function are passed by reference, therefore they are treated as global variables. The local variable declaration '`int r;`' produces no events (it is processed by the X-graph pre-compiler that invokes the memory manager to create the new local variable `r`).

```
void test(int &x, int &y) {
    int r;
    while ((5 + 4/2 * x) 3
      == 1) {
        if (y > 6)
            continue;
        else if (++y > 7) {
            r = r + 10;
            break;
        }
        r = 11;
    }
    y = x + 1;
}
```

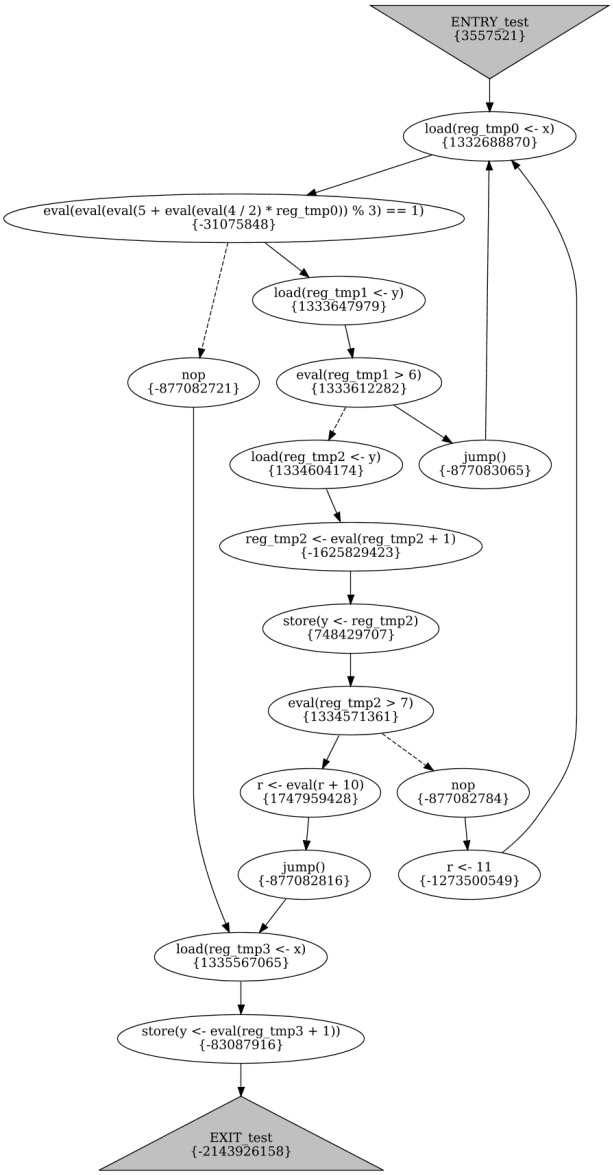


Figure 5.1: *An example of a C function compiled to the event-flow graph*

The first event '`load(reg_tmp0 <- x`' loads the value of the global variable `x` into the temp register `reg_tmp0` in order to satisfy the requirement that all computation must be performed over local variables; note that each element of the computational tree '`eval(eval(eval(5 + eval(eval(4 / 2) * reg_tmp0)) % 3) == 1)`' is represented by a local memory unit (either

`XComputationEvent` or `XConstant` or `XRegister`). The node of this computational event in the control-flow graph has two outgoing edges, the primary edge to the '`load(reg_tmp1 <- y`', the first event of the then-branch of the while-loop, and the alternative edge to the `nop`-event representing the only event of the else-branch.

As is was discussed in Section **??**, all computation events have no impact to the global state of the concurrent system. Therefore, the interpreter does not *emit* computational events, but it *creates* them. This means, when the `Y2XConverterVisitor` processes the expression tree '`(5 + 4/2 * x) 3 == 1`', it calls the method `createComputationEvent` of the interpreter that only creates the computation event and does not change its state. However, once the converter meets the global variable `x`, it calls the interpreter method `emitMemoryEvent` to copy its value to a temp register; since it meets the global variables involved to the computation before it ends to process the whole computation expression, the values of all these global variables will be copied to temp registers *before* the computation expression is used by any other event. Note that if the computation expression has not been used by any other event (for example, as the constant `1` in the following C code: '`foo(); 1; bar();`'), it is lost from the model (by the term *use* here we mean that the computation event is evaluated as a guard or assigned to another memory unit).

5.1.2 The X-graph unrolling

Once the X-graph$_{CF}$ is constructed, it should be unrolled to an acyclic flow-graph (as it was discussed in Section **??**). The Figure **??** shows the control-flow subgraph X-graphCF of the event-flow graph from the previous example (see Figure **??**) unrolled up to bound $k = 16$. The labels of events in the picture are augmented by the event reference-id number (the unrolling depth), separated from the event value by the comma.

Note that the graph does not become a tree after removing the sink node. Some branches of the graph are merged when the executions have the same event with the same unrolling depth number. For example, primary transitions of both events '`[r <- 11, 10]`' and '`[jump(), 10]`' (produced by executions of the first iteration of the `while` loop) lead to the same event '`[load(reg_tmp0 <- x), 11]`' (the first event of the second loop iteration).

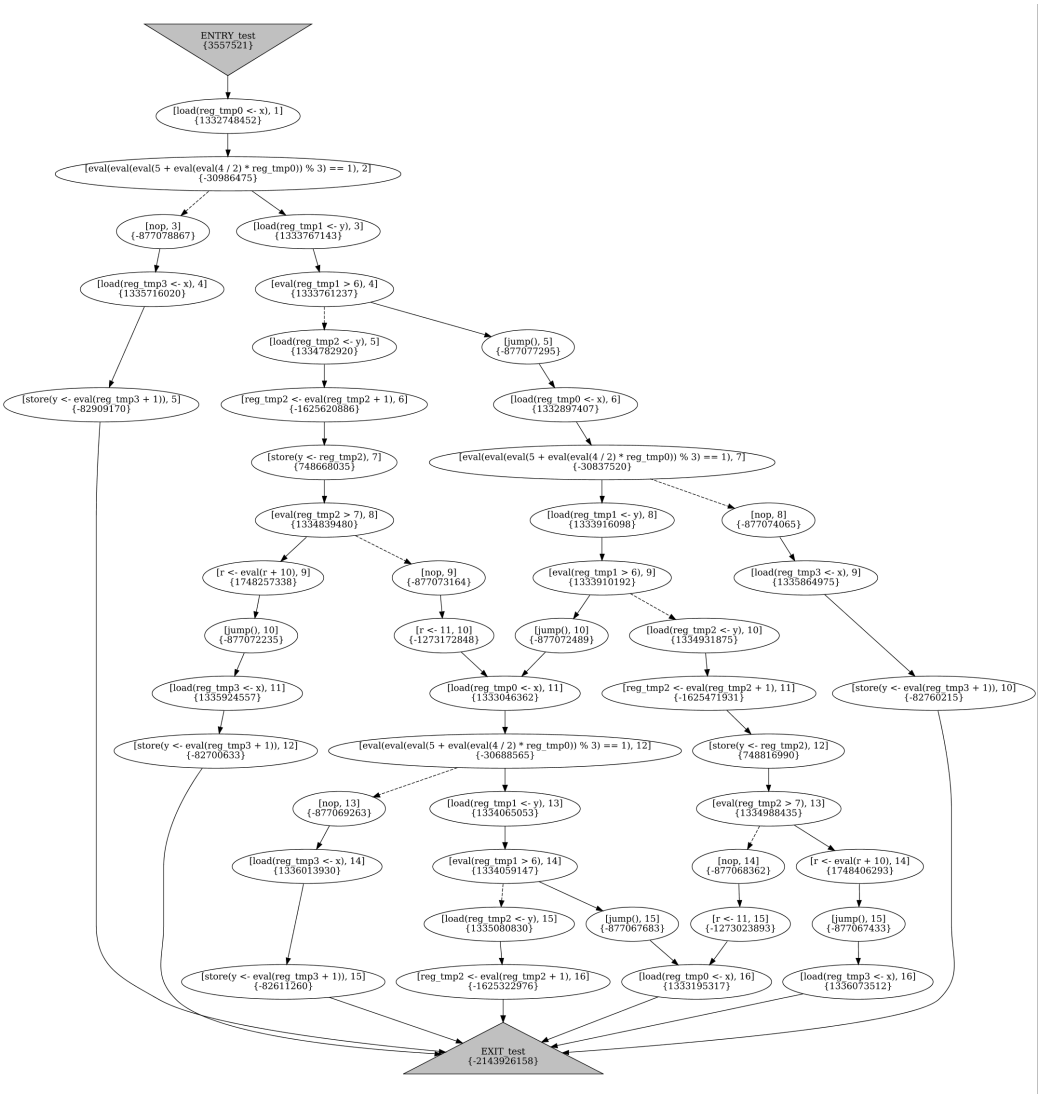


Figure 5.2: *Unrolling of the event-flow graph presented in Figure* **??**

5.2 Performance

5.2.1 State reachability analysis

As an example of the C program liable for the reachability and portability analysis, consider the Dekker's algorithm for mutual exclusion of two processes, originally described by Dijkstra [**dijkstra1962over**]; the program is presented in Appendix **??**.

For checking the reachability of the final state of the last states (sink events) of unrolled graphs compiled for a certain hardware architecture, PorthosC encodes both program and hardware memory model constraints as it was discussed in Section **??**.

The standard output of the tool looks as following:

```
$ java Dartagnan -i benchmarks/C11/Dekker.c -s tso
Interpreting...
Unrolling...
Program encoding...
Memory model encoding...
#=59
Solving...
{
  "result": "NonReachable",
  "interpretationTimer": {
    "elapsedTimeSec": 0.134
  },
  "unrollingTimer": {
    "elapsedTimeSec": 0.031
  },
  "programEncodingTimer": {
    "elapsedTimeSec": 0.14
  },
  "memoryModelEncodingTimer": {
    "elapsedTimeSec": 2.984
  },
  "solvingTimer": {
    "elapsedTimeSec": 0.037
  },
  "errors": []
}
```

For time benchmarking we ran the tool 5 times and computed the median of the encoding time. Benchmarking was performed on the Linux machine 8-core Intel(R) Core(TM) i7-3632QM CPU @ 2.20GHz, Java(TM) SE Runtime Environment (build 1.8.0_161-b12) (Java virtual machine was configured by default parameters). The time was measured by the tool itself via the Java method `System.currentTimeMillis`. For Porthos v1 we specify the unrolling bound $k_1 = 2$ (which means that all loops are executed twice), and for PorthosC we specify the bound $k_2 = 32$ (which means approximately the same, because

PorthosC shows the full encoding time for the Dekker's algorithm 2.699 sec (from which 0.152 sec spent for the program encoding, and other part spent for the memory model encoding). In contrast, Porthos v1 shows the encoding time 6.152 sec (from which 0.223 sec spent for the program encoding). Thus, the performance of the encoding stage has been improved **in 2.7 times**.

As the time spent for the interpretation and unrolling stages is negligible comparing to the encoding time, we conclude that the new architecture implies minor performance overhead comparing to the previous version of the tool. The time spent by the SMT solver while considering the SMT formula encoded by PorthosC was 0.039 sec, and in case of Porthos v1 the time was 1.291 sec. The number of events in the event-flow graph of PorthosC is 82 events (among them 59 memory events), and the number of events encoded by Porthos v1 is 95 (among them 51 memory events). Even though the number of memory events processed by Porthos v1 was less, the time of solving the result SMT-formula was significantly more. We consider this improvement to be the result of technical optimisations applied to PorthosC. The optimisations include:

- *memoisation* of the frequently requested calls.
  For example, the code of Porthos v1 contains 39 calls that take the subset of a set of events. The pattern is the following: '`program.getEvents().stream().filter(e -> e instanceof MemEvent || e instanceof Local).collect(Collectors.toSet())`',
  10 of them are performed in a loop over all events (in `Domain.encode` method of PorthosC). These calls were replaced by the lazily initialisable memoising methods of `XProgram`:

```
public ImmutableSet<XMemoryEvent> getMemoryEvents() {
    return memoryEvents != null
            ? memoryEvents
            : (memoryEvents = getAllNodesExceptSource(
    XMemoryEvent.class));
}
```

-

<TODO: compare numbers of clauses of SMT formulas> !!!
TODO: profile both programs!
Applied optimisations: memoisation immutable struct

Compare sizes of formulas
memory usage – for both encoding and solving

5.3 Comparison with HERD

5.3.1 Unique Features

- (inherited from the first version) : two memory models, smt, ... (?)
  - (new) – ?
  - kernel?? TODO: SUPPORT some kernel funcs.

5.3.2 Performance

- perhaps, better performance on a large test

Chapter 6

Summary

enumerate goals (from Task specification) in past tense + justification
    contribution
    architecture: framework
    new encoding: more laconic (compare!) + actually show size
    kernel???

6.1 Comparison with Porthos

6.1.1 New features

enumerate extensions of the grammar
    ... (?)

Appendices

A.1 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.2 The simplified x86-TSO memory model defined in the CAT language [herd10tutorial]

```
"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.3 Public interface methods of the X-interpreter

```
public interface XInterpreter {
  XProcessId getProcessId();
  void finish();
  XCyclicProcess getResult();

  // linear interpretation methods:
  XEntryEvent emitEntryEvent();
  XExitEvent emitExitEvent();
  XBarrierEvent emitBarrierEvent(XBarrierEvent.Kind kind);
  XJumpEvent emitJumpEvent();
  XNopEvent emitNopEvent();
  XAssertionEvent emitAssertionEvent(XBinaryComputationEvent assertion);

  XLocalMemoryEvent emitMemoryEvent(XLocalLvalueMemoryUnit destination,
                                    XLocalMemoryUnit source);
  XSharedMemoryEvent emitMemoryEvent(XLocalLvalueMemoryUnit destination,
                                     XSharedMemoryUnit source);
  XSharedMemoryEvent emitMemoryEvent(XSharedLvalueMemoryUnit destination,
                                     XLocalMemoryUnit source);

  // create computation event without processing it:
  XComputationEvent createComputationEvent(XUnaryOperator operator,
                                           XLocalMemoryUnit operand);
  XComputationEvent createComputationEvent(XBinaryOperator operator,
                                           XLocalMemoryUnit operand1,
                                           XLocalMemoryUnit operand2);

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

```
  // --

  enum BlockKind {
      Sequential,
      Branching,
      Loop;
  }

  enum BranchKind {
      Then,
      Else;
  }

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

A.4 The Dekker's algorithm for mutual exclusion written in C

```
{ int flag0 = 0, flag1 = 0, turn = 0; }

void P0() {
    while (true) {
      int a = 1, b = 0;
      flag0.store(memory_order_relaxed, a);
      f1 = flag1.load(memory_order_relaxed);
      while (f1 == 1) {
        t1 = turn.load(memory_order_relaxed);
        if (t1 != 0) {
          flag0.store(memory_order_relaxed, b);
          t1 = turn.load(memory_order_relaxed);
          while (t1 != 0) {
            t1 = turn.load(memory_order_relaxed);
          }
          flag0.store(memory_order_relaxed, a);
        }
      }
    }
}

void P1() {
    while (true) {
      int c = 1, d = 0;
      flag1.store(memory_order_relaxed, c);
      f2 = flag0.load(memory_order_relaxed);
      while (f2 == 1) {
        t2 = turn.load(memory_order_relaxed);
        if (t2 != 1) {
          flag1.store(memory_order_relaxed, d);
          t2 = turn.load(memory_order_relaxed);
          while (t2 != 1) {
            t2 = turn.load(memory_order_relaxed);
          }
          flag1.store(memory_order_relaxed, c);
        }
      }
    }
}
```