ACKNOWLEDGEMENTS

First of all, I wish to thank Professor Keijo Heljanko for his continuous support in both technical and mental aspects of the work. He has opened for me the opportunity of working in this extremely interesting project; he guided and assisted me during six months, explaining all the things I don't understand and helping to find the way when I have lost it.

Also, I feel grateful to the development team of the static analyser Application Inspector by Positive Technologies, where I've been working in 2016-17. This was an important experience for me, there I learned how the high-class code analysers look like. Especially I want to thank Vladimir Kochetkov who has a gift of explaining the complex matter in simplest words, and Stanislav Matveev who has taught me how to be a perfectionist in writing the code.

Although I did far not all things that I could have done, I learned a lot during the work on this project: about the compilation theory and logic, about concurrency and programming, and, most importantly, about myself.

Espoo, Finland
Saint Petersburg, Russia
18.6.2018

Artem Yushkovskiy

CONTENTS

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 [47]. In this work, we consider only *Symmetric Multiprocessor (SMP)* parallelism (systems with multiple processors connected to a single shared memory), leaving aside the discussion on distributed concurrency (systems with autonomous nodes that communicate with each other by passing messages through the network). This thesis discusses mostly the questions on the *memory consistency* rather than performance benefits of the concurrency since inconsistent memory operations can create new program behaviours, unexpected from the programmer's point of view (as well as the loss of expected behaviours), which all can be considered as a security flaw.

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 [16]. 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 [2].

As an example, consider two x86 assembly programs in Figure 1 (such little code examples that explain specific behaviour of the concurrent execution environment are called *litmus tests*). Two programs are the same, except the right-hand side program uses the synchronisation instruction `MFENCE` after the store to the shared memory. In both programs, 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 to the `y` and reads the `x`.

Taking into account all possible interleavings of the read and write instructions, both litmus tests allow the following final states:

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

(a) Without synchronisation

{ x=0; y=0; }	
P0	P1

MOV [x],1
MFENCE
MOV EAX,[y]
	MOV [y],1
	MFENCE
	MOV EBX,[x]

exists (P0:EAX=0, P1:EBX=0)	
x86-TSO: forbidden	

(b) With synchronisation

Figure 1 — Store buffering (SB): A litmus test illustrating the write-read reordering allowed by the x86-TSO memory model

- (P0:EAX=0, P1:EBX=1),
- (P0:EAX=1, P1:EBX=0),
- (P0:EAX=1, P1:EBX=1).

However, the non-synchronised litmus test in Figure 1a allows the state '(P0:EAX=0, P1:EAX=0)' on the x86 architecture as x86 processors may cache the writes to the shared memory into their local *store buffers* so that the updated value does not immediately become visible by processes running on other cores. This behaviour is known as *Store Buffering (SB)*. For preventing such a behaviour, the x86 architecture offers the synchronisation instruction MFENCE that flushes the store buffer of the process. In Figure 1b, this instruction is inserted for both processes after the store instructions, thus once it is executed in one process, the other process will read the updated value of this shared memory location. The work [48] gives a comprehensive review of common hardware architectures from the perspective of the structure of their caches and write buffers, that determine the effect of memory operations, and memory barriers they provide.

The formal way to define the semantics of memory operations and synchronisation primitives of a parallel *execution environment* (hardware, programming language, compiler, database, operation system, etc.) 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 a 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.

Chronologically the first memory model for a concurrent system was formulated by Leslie Lamport back in 1979 [37]. 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 (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 [16] by naming a concurrent piece of code the *process*.). 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. 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 [45].

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. Weakening of the guarantees (such as memory operations reordering, write buffering, etc.) is called *relaxation* of the memory model. The 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 a set of guarantees made by designers of an execution environment 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* an 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 [4]. It is the event-based non-deterministic model without global time (see Section 2.1 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 [54], much more relaxed memory model for Power and ARM architectures [12, 60, 7], etc. There are projects for validating hardware architectures wrt. a memory model as well, e.g., [40, 42].

Most modern high-level programming languages rely on relaxed memory models as well. Thus, the memory model for Java is based on the *happens-before* principle [38] and was introduced in J2SE 5.0 in 2004 [46]. The transformations valid under the Java memory model are discussed in [61]. The weak memory model for C and C++ was defined the C++11 standard [32]. The standard introduced a set of hardware-independent synchronisation fences and atomic operations, which were formalised in the work [15]. The native support for the synchronisation primitives by the programming language, defined in the standard, has replaced the library-based approach for concurrency, therefore the compiler became aware of the concurrent parts of the code. Nonetheless, in 2015 some common compiler transformations were shown to be invalid under the C11 memory model [64].

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 under active development nowadays [9, 49, 51]. This project has also an influence on the C language: the revision P0124R4 of the C standard [50] compares the Linux kernel and C11 memory models (including variable access, memory barriers, locking and atomic operations).

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 (the diy project web site: `http://diy.inria.fr/`) 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.
- Some tools that perform the weak memory model-aware program verification and model checking. The notable examples are the stateless model checkers RCMC [33], CHESS [53], Nidhugg [1], Trencher [17], and others.
- Some tools tackle the problem of automated synthesis of the synchronisation primitives, such as the automatic fence insertion tool musketeer [8], and the automatic verification and fence inference tool blender [36].
- Along with synthesis of synchronisation primitives, some tools were designed to perform the automatic synthesis of litmus tests for the weak memory model, for example litmustestgen by the NVidia Research [41].
- The framework Alloy [65] can be used for understanding WMMs; it can automatically generate conformance tests between two memory models, distinguish two WMMs, and check monotonicity and compiler mappings of a program.
- 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 [35],
    - the tool that performs the sequentialisation of concurrent programs [10],
    - the tool Weak2SC generates weak memory model descriptions of the program, which can be fed into standard model checking tools (such

as SPIN [30] or NuSMV [19]) for performing memory model-aware analysis [63].

All the tools listed above consider only a single memory model, however, in real life we face serious engineering problems involving more than one execution environment. One of these problems is the *portability analysis* of the program from one hardware architecture to another. A program written in a high-level language is compiled for different hardware architectures. Even if all the compiler optimisations were disabled (which is a 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 for 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 $\mathcal{T}$ [58].

The very first tool that performs the WMM-aware portability analysis is Porthos (the Porthos project repository: `http://github.com/hernanponcedeleon/Dat3M`) introduced in April 2017 [58]. This tool reduces described problem to a bounded reachability problem, which can be solved via an SMT-solver (see Section 3.2). 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 ([44]), 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 [51]. 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 functions and macros

(such as the macro `READ_ONCE` that guarantees memory read) and be easily extensible for defining new functions (have the separate module for the purpose of a knowledge base).

As the general architecture and almost all components of Porthos were to be redesigned, the tool received a new name *PorthosC* (hereinafter with the names *Porthos* and *Porthos v1* we refer to the tool Porthos of version 1, whereas the new implementation of Porthos is called *PorthosC*.). 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 reachability and portability analysis, but serve as a basis for other kinds of static analysis of concurrent programs.

1.4 Thesis structure

The thesis is organised as following. Chapter 2 gives a general view on the weak memory model-aware analysis. Chapter 3 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 4 delves into the description of architectural solutions and implementation details of the PorthosC framework. Chapter 5 gives a comparison of key features of PorthosC and Porthos v1 by providing examples of the compilation and the unrolling stages; and it provides some performance benchmarks of PorthosC in both reachability and portability analysis modes. Chapter 6 summarises results of the work and proposes possible directions for future work.

2 MEMORY MODEL-AWARE ANALYSIS

The main idea behind the memory model-aware program analysis is that the set of all possible executions 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*) [5]. 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 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*) [4, 7]. The vertices of such a graph represent *events* (see Section 2.1.1), and edges represent basic relations (see Section 2.1.2). The graph represents the set of executions (sequences of events; see Section 2.1.3) 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 [53]:

a) *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.;
b) *scheduling non-determinism*, caused by the interleavings, which in turn are caused by the scheduler activity; and
c) *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 [39, 13].

2.1.1 Events

An *event* is a fact of executing the low-level primitive atomic operation such as memory access, threads synchronisation, computation over the local-memory, control-flow jump, 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 $\texttt{id}(e_m)$ [4].

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 separate 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.

Synchronisation instructions (fences) cause *barrier events*, which do not perform any computation or memory value transfer, instead, they the set of program behaviours by adding barrier relations to the program model. Functionally, a fence may be a synchronisation barrier or a instruction for flushing memory caches into the main memory, etc. For instance, the `mfence` instruction of the x86 assembly flushes the store buffers of the thread, and thus does not allow the `rf`-relation to hold (see Section 2.1.2).

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 [5].

The basic relations are the following [4]:

– The *control-flow* of a program is defined by the *program-order* relation $\mathsf{po} \subseteq \mathbb{E} \times \mathbb{E}$, 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 $e_1 \overset{\mathsf{po}}{\rightarrow} e_2$.
– The *data-flow* of a program is defined by *communication relations*:

  - the *read-from* relation $\texttt{rf} \subseteq \mathbb{W} \times \mathbb{R}$ that maps each write event to the read event that reads the value written by write event; and
  - the *coherence-order* relation $\texttt{co} \subseteq \mathbb{W} \times \mathbb{W}$ 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 [4]:

– 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:
$r \xrightarrow{\texttt{fr}} w \;=\; (\exists w'.w' \xrightarrow{\texttt{rf}} r \wedge w' \xrightarrow{\texttt{co}} w)$;

– the *communication* relation po over memory events, that fully describes the data-flow of a program:
$m_1 \xrightarrow{\texttt{com}} m_2 \;=\; ((m_1 \xrightarrow{\texttt{rf}} m_2) \vee (m_1 \xrightarrow{\texttt{co}} m_2) \vee (m_1 \xrightarrow{\texttt{fr}} m_2))$;

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

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

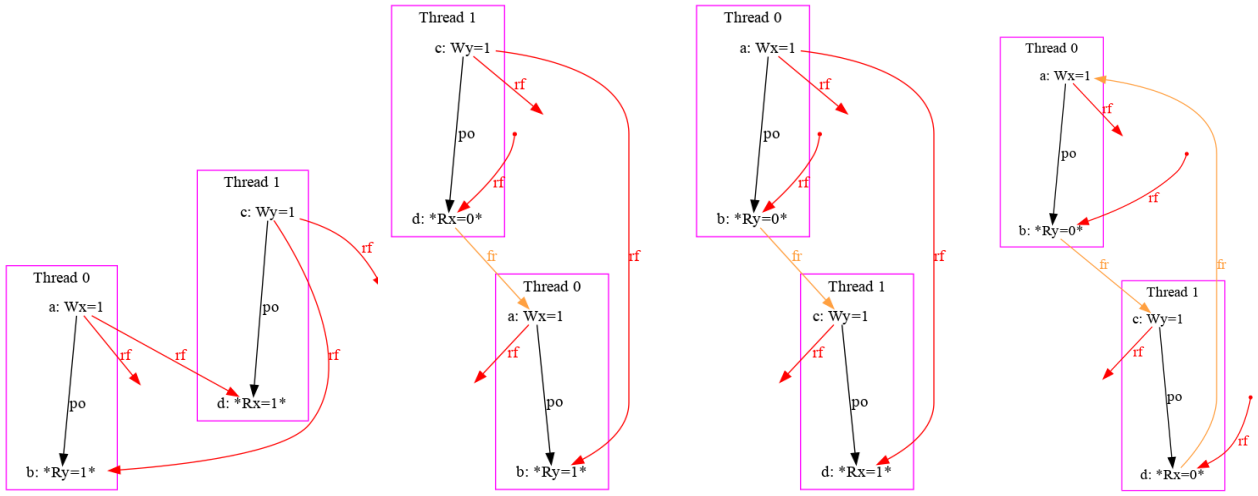
– the semantics of *fences* (memory barriers) specific for different architectures may be defined as derived relations.

2.1.3 Executions

The semantics of a concurrent program is represented as the set of allowed executions. An *execution* is a path in the event-flow graph defined by po - and rf -relations and set of final writes to a given memory location that is valid under

certain memory model [7]. It can be interpreted as a sequence of guesses which event is to be executed next. A *candidate execution* is an execution that is not yet constrained by a memory model.

Figure 2 illustrates four possible candidate executions for the litmus test Example 1 (the pictures are generated by the herd7 tool, version 7.47). Since there are no conditional jumps, the `po`-relation is defined and we do not need to guess it. Since each thread performs a 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: `(P0:EAX=1, P1:EAX=1)`

(b) Final state: `(P0:EAX=1, P1:EAX=0)`

(c) Final state: `(P0:EAX=1, P1:EAX=1)`

(d) Final state: `(P0:EAX=0, P1:EAX=0)`

Figure 2 — Candidate executions for the litmus test in Example 1

Thus, there are only four possible executions defined by the choice of `rf`-relation. The candidate executions pictured in Figures 2a–2c 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 2c 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 [5]. It is a domain specific language for describing consistency properties of concurrent programs. The language

combines expressive power of a functional language (being inspired by OCaml, it adopts the OCaml types, first-class functions, pattern matching and some other features) with concepts of memory models (sets of events, relations, operations and assertions over relations). In CAT, new relations can be defined via the keyword `let` and the following operators over relations [5].

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

a) *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 $\texttt{r}^{-1}$.

b) *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, Figure 3 presents the excerpt from the x86-TSO memory model [6]. This memory model specification asserts acyclicity of the communication relation (the union of `rf`-, `fr`- and `co`- relations), `po-loc`-relation, `mfence`-relation and some other derived relations [54].

```
...
let po_ghb = WW(po) | RM(po)
let implied = PA(poWR) | WR(po)
let GHB = mfence | implied | po_ghb | rfe | fr | co
let com = rf | fr | co

empty atom & (fre;coe)
acyclic po-loc | com
acyclic GHB
```

Figure 3 — Excerpt from the x86-TSO memory model in the CAT language

3 PORTABILITY ANALYSIS AS AN SMT PROBLEM

As it was discussed in Chapter 1, a concurrent program may behave differently when compiled for different hardware architectures. This may cause the portability bugs, the behaviour that is allowed under one architecture and forbidden under another. This Chapter provides the short introduction into the general model checking and reachability analysis problem, after which it describes of the concurrent software portability analysis stated as a *bounded reachability* problem, which in turn can be reduced to an SMT problem [57].

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*) [24]. As the state machine 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 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, a set of constraints.

Usually, the specification defines temporal constraints over the system properties. For instance, the specification can assert that the property *always* holds (the *safety* property) or the property will *eventually* hold (the *liveness* property). 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 [20].

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 serious due to the exponential number of possible interleavings of states. Therefore, over past 30 years the research in model checking has been fighting the state explosion problem mostly by optimising search space, search strategy or basic data structures of existing algorithms [23].

One of the first techniques that optimises the search space considerably is the symbolic model checking with *Binary Decision Diagrams (BDDs)* [18]. 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 [21]. 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 [22]. 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 (not all the state space has been checked). However, there exist numerous techniques for making BMC complete for finite-state systems (e.g., [62]).

At present time, satisfiability solvers are behind the numerous verification and bounded model checking tools. Although, as most modern programming languages are typed, a SAT-formula that encodes the constraints of a program must include non-boolean constraints encoded as boolean variables, which is ineffective, therefore most software analysis and verification tools use *Satisfiability Modulo Theories (SMT)* solvers, that integrate first-order reasoning with the background theory reasoning [34]. For example, the Integer theory includes operators '$+$', '$-$', '$\geqslant$' and others). Most modern SMT-solvers support integer arithmetic, real arithmetic, fixed-size bit-vectors, arrays, and other theories. The direction of the development of SMT-solvers is provided by the *SMT-LIB standard* [14], which provides definition of the unified language

for defining SMT-formulas, and description of most common underlying logics and theories.

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: $\mathtt{SAT}(\mathit{Init}(\vec{x_0}) \wedge \mathit{Trans}(\vec{x}_0, \vec{x}_1) \wedge ... \wedge \mathit{Trans}(\vec{x}_{k-1}, \vec{x}_k) \wedge \mathit{Bad}(\vec{x}_k))$.

3.2 Portability analysis as a bounded reachability problem

A portability analysis problem may 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 $\mathit{cons}_\mathcal{M}(P)$ which calculates the set of executions of the program P that are consistent under the memory model $\mathcal{M}$. 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 $\mathcal{M}_\mathcal{S}$ [58]:

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 $\mathit{cons}_{\mathcal{M}_\mathcal{T}}(P) \subseteq \mathit{cons}_{\mathcal{M}_\mathcal{S}}(P)$

Note that the definition of portability requirements against *executions* is strong enough, as it implies the portability against *states* (the *state-portability*) [57]. 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.

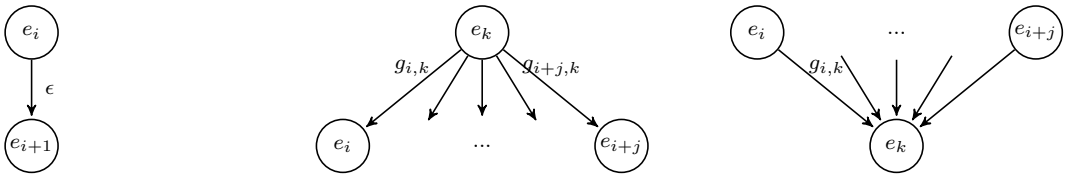
3.2.1 Encoding for the control-flow

The control-flow of a program can be represented by the *control-flow graph*, a directed acyclic connected graph with a single source and one or multiple sink nodes. Each control-flow edge contains the label that denotes the transition predicate called *guard*. The empty guard (a *true* predicate) is denoted as 'ϵ'. A guard represents a variable or a computational expression over local variables. As a guard depends on the data-flow of the program, it 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 evaluating a computation (the *primary* and *alternative* transitions), see Section 4.3.1.2 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. The new encoding is to be smaller than the old one used in Porthos since it does not produce new variables for each high-level statement of the input language.

For instance, Porthos v1 uses the encoding scheme where the control-flow of the sequential instruction $i_1 = i_2; i_3$ is encoded recursively 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 the control-flow of the branching instruction $i_1 = (c\,?\,i_2 : i_3)$ was encoded recursively as $\phi_{CF}(c\,?\,i_2 : i_3) = (cf_{i_1} \Leftrightarrow (cf_{i_2} \vee cf_{i_3})) \wedge \phi_{CF}(i_2) \wedge \phi_{CF}(i_3)$. In contrast, the new encoding scheme implemented in PorthosC firstly compiles the recursive high-level instructions into the linear low-level event-flow graph, which is then encoded into an SMT-formula. A guard in the event-flow graph is a value of conditional variable on the branching event, which is encoded as a data-flow constraint (see Section 3.2.2). In general, the new encoding scheme follows the one proposed in [28, Chapter 5.1.2] for encoding Petri-nets.

Let $\mathbf{x} : \mathbb{E} \to \{0, 1\}$ be the predicate that signifies the fact that the event has been executed (and, consequently, has changed the state of the system). Consider the



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

Figure 4 — Possible mutual arrangements of events in an event-flow graph

possible mutual arrangements of nodes in an event-flow graph presented in Figure 4. 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:

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

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

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

Equation 3.1 shows the encoding for the sequential control-flow represented in Figure 4a and reflects the fact that the event e_2 can be executed iff the event e_1 has been executed. Equation 3.2 shows the encoding for the branching control-flow depicted in Figure 4b, 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 3.3 shows the encoding for the control-flow of a merge-point represented in Figure 4c: 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 and simplicity 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 branch represents a conditional jump forward), we add a synthetic *nop-event* as it is shown in Figure 5.

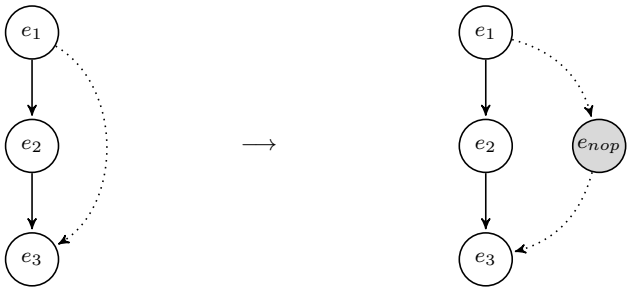
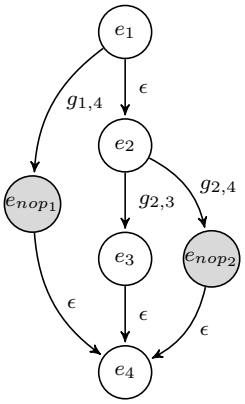


Figure 5 — Transformation of the forward-jump control-flow



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

Figure 6 — Example of encoding for the control-flow of the X-graph

As an example of the control-flow encoding, consider the event-flow graph in Figure 6, 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 SMT-variables $\mathbf{x}(e_1)$, $\mathbf{x}(e_2)$ and $\mathbf{x}(e_4)$ should be assigned to 1, and the variable $\mathbf{x}(e_3)$ should be assigned to 0.

Note that the generalised encoding scheme does not require the branching transitions to be mutually-exclusive (for instance, consider two branching 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 assignment $\mathbf{x}(e_1) = 1$, $\mathbf{x}(e_2) = 1$, $\mathbf{x}(e_3) = 0$ and $\mathbf{x}(e_4) = 0$, which 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 v1 uses the *Dynamic Single-Assignment (DSA)* form, that requires indices to be unique only within a control-flow 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 [58].

Following [57], 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 a single thread:

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

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

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

In Equation 3.6, the function $\mathbf{v} : \mathbb{C} \rightarrow \mathbb{R}$ e**v**aluates the computation event (the value is determined by `co` - and `rf` - relations). 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 7.

Algorithm

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: $S \leftarrow$ empty map; $S[e_0] \leftarrow$ empty map
3: **for each** event $e_i \in G.N$ in topological order **do**
4: **for each** predecessor $e_j \in \mathtt{pred}(e_i)$ **do**
5: $S[e_i] \leftarrow \mathtt{copy}(S[e_j])$
6: **for each** variable $v_k \in$ set of variables accessed by e_i **do**
7: $S[e_i][v_k] \leftarrow \mathtt{max}(S[e_i][v_k],\ S[e_j][v_k])$
8: **if** need to update the index of v_k **then** ▷ cases (1)-(2)
9: $S[e_i][v_k] \leftarrow S[e_i][v_k] + 1$

Figure 7 — Algorithm for computing the SSA-indices

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

As it has been described before, the `rf` -relation links data-flow between events. The encoding of this linkage has been left untouched as it is implemented in Porthos v1:

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

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 $e_1 \xrightarrow{\mathtt{rf}} e_2$ (i.e., 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 was proposed in [58]. The encoding consists of two parts: encoding of the *derived relations* and encoding of the memory model *assertions*.

In the SMT-formula, a relation $x \xrightarrow{\mathsf{r}} y$ is represented by a boolean variable $r(x, y)$ that indicates whether the relation holds. Derived relations are encoded by fresh boolean variables according to the following rules [57]:

$$r_1 \cup r_2(e_1, e_2) = r_1(e_1, e_2) \vee r_2(e_1, e_2);$$
$$r_1 \cap r_2(e_1, e_2) = r_1(e_1, e_2) \wedge r_2(e_1, e_2);$$
$$r_1 \setminus r_2(e_1, e_2) = r_1(e_1, e_2) \wedge \neg r_2(e_1, e_2);$$
$$r^{-1}(e_1, e_2) = r(e_2, e_1);$$
$$r^*(e_1, e_2) = r^+(e_1, e_2) \vee (e_1 = e_2);$$
$$r_1; r_2(e_1, e_2) = \bigvee_{e_k \in \mathbb{E}} r_1(e_1, e_k) \wedge r_2(e_k, e_2); \text{ and}$$
$$r^+(e_1, e_2) = \mathtt{tc}_{\lceil log|\mathbb{E}| \rceil}(e_1, e_2), \ where$$
$$\mathtt{tc}_0(e_1, e_2) = r(e_1, e_2), \ and$$
$$\mathtt{tc}_{i+1}(e_1, e_2) = r(e_1, e_2) \vee (\mathtt{tc}_i(e_1, e_3); \mathtt{tc}_i(e_3, e_2)).$$

Note that CAT language allows mutually-recursive definitions of relations (for example, '$\mathsf{r}_1 = \mathsf{r}_2 \cup (\mathsf{r}_1; \mathsf{r}_1)$'). The basic idea of using the Kleene fixpoint iteration for encoding such relations was also proposed in [58]: 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 [58], encoding the acyclicity assertion uses numerical variable $\Psi_e \in \mathbb{N}$ for each event e in the relation to be asserted: $\mathit{acyclic}(r) = (r(e_1, e_2) \Rightarrow (\Psi_{e_1} < \Psi_{e_2}))$. The irreflexivity assertion as $\mathit{irreflexive}(r) = \bigwedge_{e_k \in (E)} \neg r(e_k, e_k)$.

4 PORTHOSC: THE 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, taking into account its complexity and numerous pitfalls, goes far beyond current thesis (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`.), we consider the accomplished work as a step towards it. The PorthosC repository is located at `https://github.com/ajuszkowski/PorthosC`.

4.1 General principles

The Porthos v1 does not distinguish the event-based program model from the high-level AST, and they both are encoded into a single SMT-formula (see classes of package '`dartagnan.program`' of Porthos v1). Moreover, the syntax tree is implemented in Porthos v1 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 v1 that recursively compute some properties of the AST and change its state). We are inclined to consider the old architecture to be fast to develop, but difficult to maintain (since it is difficult to guarantee the correctness of 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 clearly separated the high-level intermediate code representation (the AST structure) from the low-level

event-based representation (the event-flow graph). Such a modular architecture will allow to support multiple input languages (apart from the C language, PorthosC should be able to analyse litmus tests in different assembly languages and, as a compatibility mode, the input language of Porthos v1.) by parsing them and converting parsed syntax trees to a simplified AST.

All data-transfer objects (DTOs) used by PorthosC must be immutable, so that it is possible to guarantee the correctness of the program by controlling preservation of 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 user-defined immutable object, or an immutable collection provided by the library Guava by Google (the Guava project repository: `https://github.com/google/guava/`)).

During the development of PorthosC, we mainly followed the *KISS principle*, which can be exhaustively described in 17 Unix Rules of Eric Raymond [59]. The following list summarises the main rules we followed during the development of PorthosC:

1. *Robustness:*
   1.1. completeness of the analysis,
   1.2. modular architecture: each module can be tested independently,
   1.3. use of software design patterns where necessary, and
   1.4. use 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.

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 mainly means the clear and informative naming of classes, functions and fields. It also implies the unified ordering of methods in classes, minimised size of methods code, clear tabulation, etc.

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 concepts of Object-Oriented Programming (OOP) inherent in Java suitable for modelling the programming languages. Although Java does not show the best results in performance benchmarks (for example, compared to C++ [31]), 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 Z3 (the Z3 project repository: `https://github.com/Z3Prover/z3`) by Microsoft Research [25], which is 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. 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 if necessary.

Currently, PorthosC can operate only in the intra-procedural analysis mode (not supporting invocations of the user-defined functions), 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-function) analysis by inlining user-defined functions calls and binding variable contexts. In this mode, instead of analysing a single source code file, the tool can process the whole code project, where the functions to be executed in parallel are specified separately by the user. Also, the tool can 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 Program input

Both Porthos v1 and PorthosC use the ANTLR parser generator (the ANTLR project repository: `https://github.com/antlr/antlr4`) [56] for creating the input-language parsers. 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 processing languages.

Figure 8 represents the grammar sketch in BNF syntax of the input language used by Porthos v1. The input language parser used by Porthos v1 suffers from several disadvantages. Firstly, it contains the parser code inlined directly into the grammar, so that the grammar serves as a template for the parser code (which is called semantic actions in ANTLR). 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 resolves 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 the grammar for encoding the semantics of a new function. As the reader might have noticed from the grammar sketch in Figure 8, different kinds of memory operations of of the Porthos v1 input language vary syntactically as well. For example, the assignment of a local computation to a register uses the symbol '<-', the atomic non-relaxed load operation is denoted by '<:-', atomic non-relaxed store operation is denoted by ':=', and the semantics of relaxed `load` and `store` is resolved syntactically by matching the function name and arguments. Moreover, only the operator '<-' accepts an expression as the source of data, which means that expressions could be assigned only to registers. In

```
⟨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 8 — Sketch of the input language grammar used by Porthos v1

PorthosC, the semantics of the data-flow operation is determined according to the types of operands, that are determined during the pre-compilation stage (see Section 4.3.2.4). The semantics of the functions also being resolved during the compilation stage via the *invocation hooking* mechanism (see Section 4.3.2.5).

Thirdly, the grammar used by Porthos v1 accepts only a restricted set of operations. For example, the computation expressions can use only local variables. Thus, in the assignment expression '`r <- (x + 1);`', the variable `x` will be parsed as a local variable even though it can be used as a shared variable in other parts of the program, which may lead to the inconsistency of analysis. In PorthosC, all shared variables involved into a computation expression are tentatively copied to temporary local variables.

Fourthly, Porthos v1 supports only integer constants and expressions. In PorthosC, we extended support for primitive types supported by the Z3 solver (32-bit integers

are encoded as `Int`s of Z3, floats are encoded as `Real`s). Although the Z3 supports the array theory (characterised by the select-store axioms [52]), 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 has the following minor drawbacks. The operators are non-associative (expressions of the form '`1 + 2 * 3`' can not be parsed). Comparing to C statements, which 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 test-specific syntax for variables initialisation is used only for declaring the shared variables (all of them are 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 [32], 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 (the ANTLR grammars repository path: `https://github.com/antlr/grammars-v4`). Current version of PorthosC does not recognise C processor directives (it ignores them), however, in future it can be extended to support them.

4.3 Architecture

The general processing scheme of the old tool Porthos v1 is the following. The input program is parsed to the AST, where leaves are the events. The control-flow of the program is encoded into an SMT-formula as it was discussed in Section 3.2.1, and the data-flow (`po` - and `rf` - relations) and some derived relations are encoded by the single function ('`dartagnan.wmm.Domain.encode`') that loops over all events, checks their properties (the kind of event, the operands of the memory event, the control-flow properties such as condition level, etc.) and adds the assertions of corresponding relations to the formula. Encoding all the relations (the *program domain encoding*) in a single large function is an error-prone and hardly manageable approach. Moreover, the

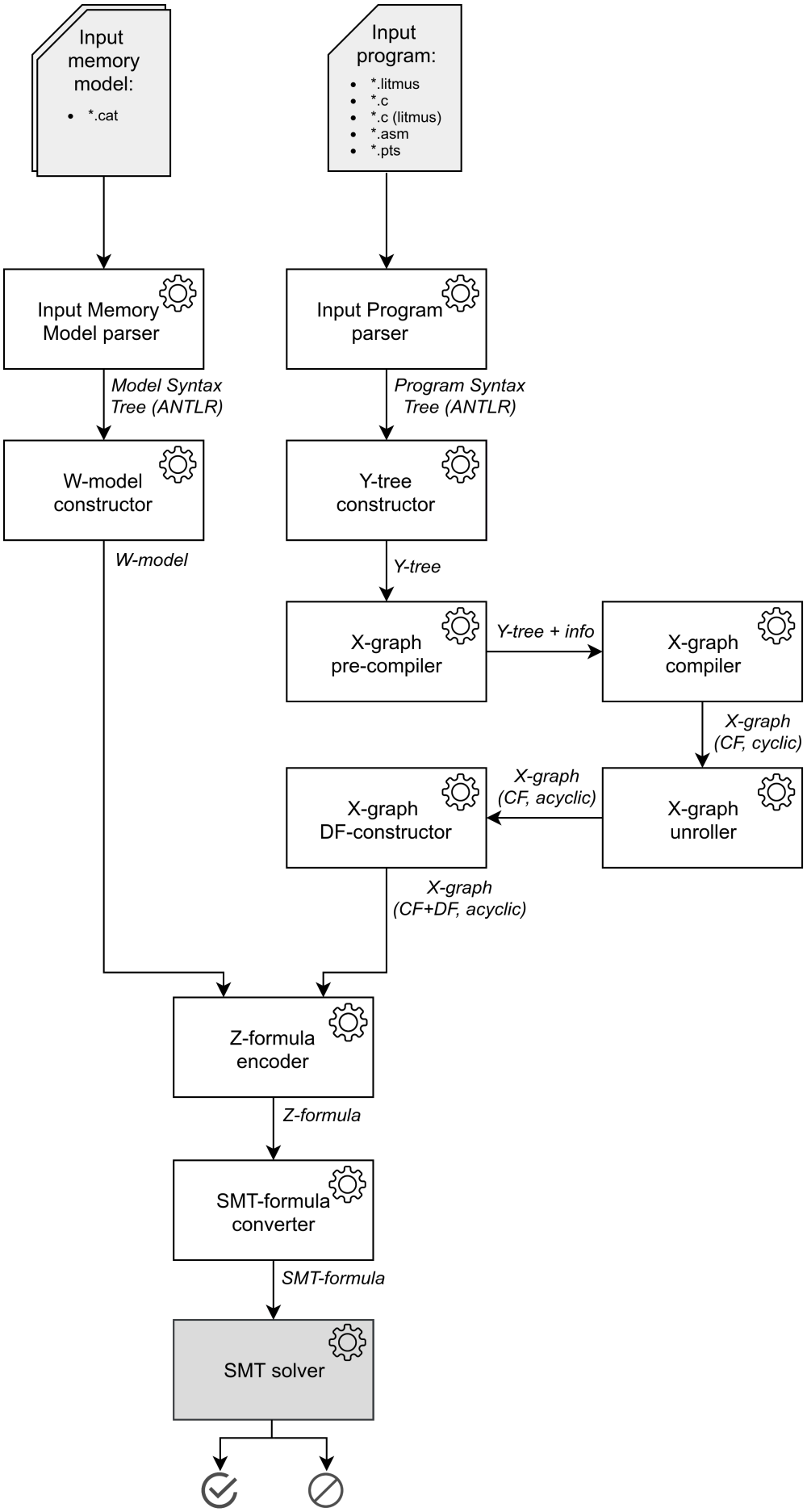


Figure 9 — The general architecture of PorthosC

events, that constitute the elements of a composite data structure, carry the information about it (such as the condition level, which is the property of the control-flow graph), while it should be carried by the data structure itself. In addition, the program domain encoding function does not consider the memory model, therefore the result formula can contain relations that are not constrained by the memory model specified by the user and thus are redundant (for instance, the TSO memory model does not contain Power-specific relations `ii`, `ic`, `ci`, etc., that are always encoded regardless the memory model). In PorthosC, we moved the encoding logic into a separate module (see Section 4.3.2.8) that accepts the input program and the user-defined memory model, therefore does not add constraints over unused relations to the result SMT-formula.

The high-level architecture of PorthosC is presented in Figure 9. The program takes as input the program to be analysed and one (the reachability analysis mode) or two (the portability analysis mode) memory models. However, instead of parsing memory models from the CAT file, the tool may operate with pre-defined memory models (SC, TSO, PSO, RMO, Alpha, Power, ARM; this feature is inherited from Porthos v1). The parsed program syntax tree is then converted (Section 4.3.2.3) to a program AST called Y-tree (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 e**x**ecuted; and with letter 'W' we prefix elements of the **w**eak memory model AST.)(Section 4.3.1.1), which then is being preprocessed at the pre-compilation stage (Section 4.3.2.4) in order to collect information necessary for the compilation. The Y-tree then is being compiled (Section 4.3.2.5) to an X-graph representation (Section 4.3.1.2). The compiled X-graph then is being converted to an acyclic form (Section 4.3.2.6) in order to be encoded into a Z-formula (Section 4.3.1.4). Apart from that, the memory-model constructor (Section 4.3.2.2) constructs the abstract syntax tree of derived relations of the weak memory model W-model (Section 4.3.1.3). Thereafter, W-model and acyclic X-graph are encoded (Section 4.3.2.8) to a Z-formula representation (a wrapper over an SMT-formula), which then is translated to an SMT-formula, which then is solved by the SMT-solver (note, current implementation of PorthosC excludes the Z-formula as at the moment it is enough for PorthosC to use only a single SMT-solver).

4.3.1 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.

4.3.1.1 Y-tree

The first internal representation used by PorthosC is the *Y-tree*, which represents an untyped high-level recursively defined AST (hereinafter we use the term *recursive* data structure (also called *inductive* data structure) to refer a complex data type that can contain elements of the same type.) . The Apendix A.1 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*. However some details of the syntax might have been abstracted away (for instance, array operations may be emulated by functions invocations, see [29, Chapter 5]), 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 `Origin` (original code coordinates) instance that contains information about the coordinates of the input text that has generated the Y-tree element. The origin can be translated to the code citation by the `CitationService`.

Following the C11 standard [32], 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. Although the pointer arithmetic is not fully

supported by PorthosC, the Y-tree contains information about the *pointer level* of an expression (which is modelled as an integer number). We distinguish the subset of expressions that imply no side-effects, they implement the interface `YAtom` and can be global or local (which is determined syntactically).

The Y-tree expressions are the following:

- `YBinaryExpression` that model the C binary operator (*relative* operator that compares two expressions of any type, *logical* that processes two boolean expressions, and *numerical* that processes two integer or real 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, see Section 4.3.2.5);
- `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, the `YType` is used for storing the type information on variables or types declarations). It consists type modifiers and qualifiers, that will be converted to the X-type by the `Y2XTypeConverter` during the pre-compilation stage (see Section 4.3.2.4).

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 definition `YProcessDefinition` used in intra-procedural analysis mode. The function definition contains the `YCompoundStatement` body and the `YMethodSignature` signature (method name and types of formal parameters), which is used in the function resolution during the pre-compilation stage. The other litmus-specific definitions 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`.

4.3.1.2 X-graph

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

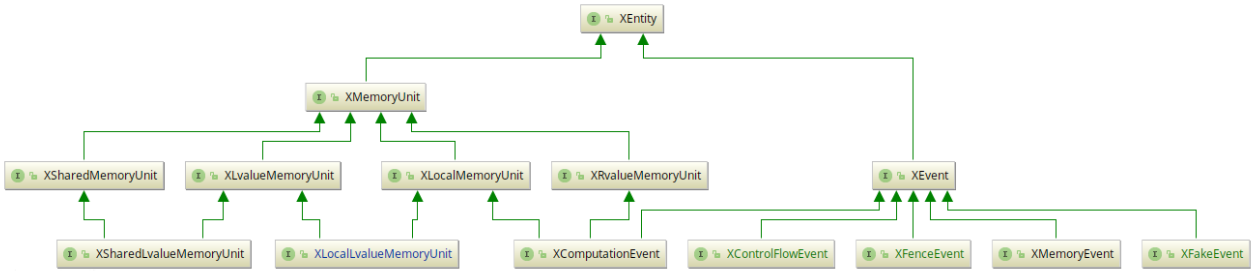


Figure 10 — The inheritance tree of interfaces of X-graph

edges as X-graph$_{CF}$. The complete X-graph is X-graph$_{CF+DF}$ = X-graph$_{CF}$ ∪ X-graph$_{DF}$. The UML diagram in Figure 10 represents the class hierarchy (hereinafter all class diagrams are generated by IntelliJ IDEA Ultimate.)of main interfaces of the X-abstraction level. The *full* list of classes that constitute the X-graph hierarchy can be found in Appendix A.1. 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 4.3.1.2).

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 format of litmus tests, we distinguish three types of X-graph: one for the parallel processes, one for the 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 all are modelled by the same data structure. The examples of restrictions on X-graphs are the following: the initialisation block can not have branchings, fence events; the process cannot have assertion events; the assertion block can not have shared-memory events or fence events. We discuss the interpretation of different types of statements in more detail in Section 4.3.2.5.

Memory units. An *X-memory unit* is a memory cell of an abstract machine executing the code (that represents the X-graph). The abstract machine has infinite number of arbitrary-sized *registers* (local memory units) and *locations* (shared memory units). Local memory units extend the `XProcessLocalElement` interface, that stores

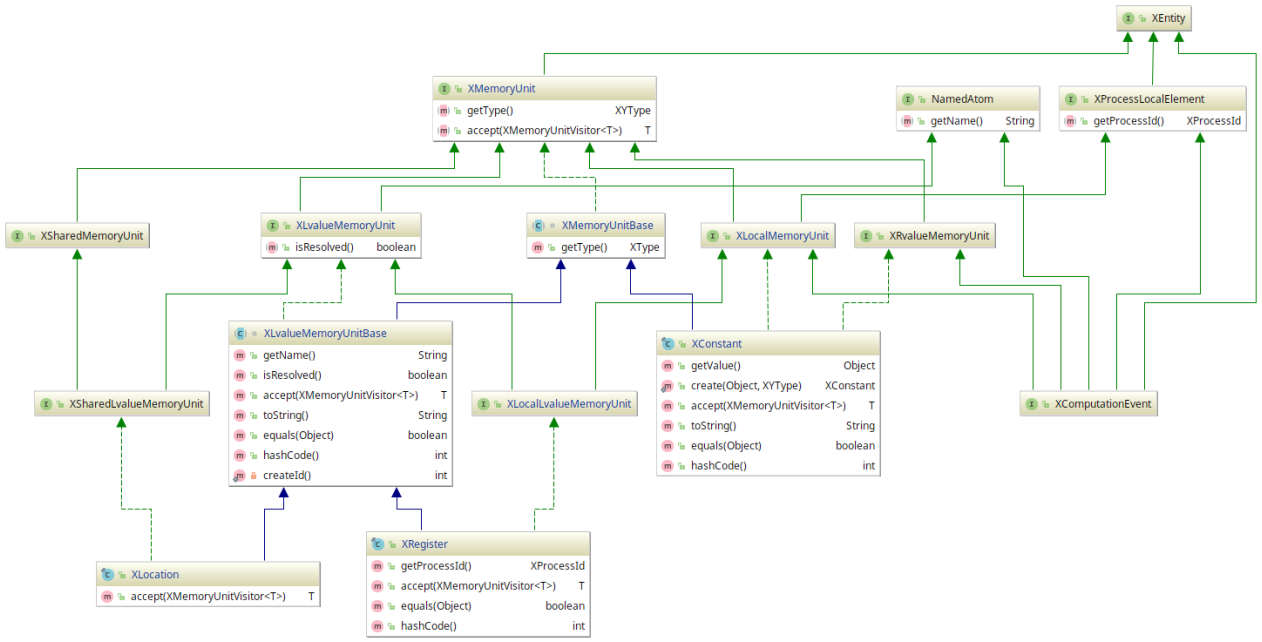


Figure 11 — The inheritance tree of X-graph memory units

the ID of the owning process. Figure 11 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 the current Section).

Each memory unit has an `XType` associated with it. The X-type is a symbolic representation of the C primitive type (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 4.3.2.4.) that is easily convertible to an SMT-type (modelled by `ZType`). All X-memory units carry the boolean flag that indicates whether it has been resolved correctly by the X-memory manager.

The memory units are created and stored by the `XMemoryManager`, which provides interface for accessing memory units during compilation stage. For more detailed description of memory management see Section 4.3.2.5.

Events. An *X-event* represents the fact of executing the primitive operation, which is independent from other events. Each X-event implements the `XEvent` interface and is carries the information about the process generated them and a unique event label, which is modelled by an immutable structure `XEventInfo`. Each event should carry the reference to the Y-instruction that has generated it (this information can be useful for the unrolling discussed in Section 4.3.2.6, however currently each X-event contains the reference to the original code location, similarly to Y-instructions discussed above).

The following interfaces model basic kinds of events (see Figure 10):

– `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`) $\leftarrow$ (`XLocalMemoryUnit`),
- `XLoadMemoryEvent`:
  (`XLocalLvalueMemoryUnit`) $\leftarrow$ (`XSharedMemoryUnit`),
- `XStoreMemoryEvent`:
  (`XSharedLvalueMemoryUnit`) $\leftarrow$ (`XLocalMemoryUnit`), and
- `XInitialWriteEvent`:
  (`XLvalueMemoryUnit`) $\leftarrow$ (`XRvalueMemoryUnit`).

– `XComputationEvent`

We distinguish two types of computation events:

- `XUnaryComputationEvent` that encodes bit negation and no-operation; and
- `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
- `XFunctionCallEvent` that models function invocations. The function call also implements the `XLocalMemoryUnit` since it represents the computed value returned by the function call. Currently, PorthosC only supports invocations of the method calls registered in its knowledge base (see invocation hooking mechanism described in Section 4.3.2.5). For interpreting known function invocations, PorthosC must perform three steps: bind actual arguments to the formal parameters (treated as temporary registers as in the `fastcall` calling convention), push the called `XFunctionCallEvent` onto the call stack, and the perform the control-flow jump to the function body. The call stack must be bounded by the user-defined parameter; once the stack is full, the interpretation should continue without jumping to the body of the invoked function (such cases must be properly logged). Each `return` statement must create the assignment of the function call event that is on the top of call stack. Note, this approach will work for recursion as well, and the recursive calls together with loops must be unrolled up the user-defined boundary (see Section 4.3.2.6). The discussion on how to set up the `po`-relations in the

event-flow graph with recursive function calls can be found in [11].

If the function has not been resolved, it can be safely assumed to be a no-operation function (with proper logging). 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 add 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 5), 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 Z-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 2.1.1):

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

4.3.1.3 W-model

The *W-model* represents a recursive AST of *computation 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 2.1.2) and sets of events ($\mathbb{R}$, $\mathbb{W}$, $\mathbb{IW}$, etc.; see Section 2.1.1). 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 2.2. Each element of W-model implements the interface `WModel`, and for the sake of transparency contains the origin location in the model file. As the W-model hierarchy almost completely follows the mathematical structure discussed in Sections 2.2 and 3.2.3, we skip the details of its implementation.

4.3.1.4 Z-formula

The *Z-formula* representation is supposed to be a wrapper over an SMT-formula used as an additional abstraction level to increase the transparency of the architecture, simplify the debugging process and ease the support of different SMT-solvers. However, currently PorthosC skips this internal representation and both the program and memory models directly into the SMT-formula via Java API offered by the Z3 solver.

The Z-abstraction level should model the logical formulas (definitions and assertions) that are put on the assertion stack of the SMT-solver. Generally, a Z-formula should represent the S-expression-based syntax of the SMT-LIB language [14]. Currently, the Z-formula is supposed to represent definitions and assertions over variables and binary and unary expressions. However, for the purpose of improving the encoding scheme, 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 must be *typed* (or *sorted*, in terms of SMT-LIB standard). Basically, the Z-type must support 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-formula. For the sake of transparency, each element of a Z-formula should contains the origin location as the reference to the X-element that produced current Z-element.

4.3.2 Processing units

This section describes program units that construct, transform and analyse internal representations described above.

The construction of the X-graph is performed in three stages. Firstly, 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 over the $\text{X-graph}^{\text{U}}_{\text{CF}}$ and produce the

full event-based graph X-graph$^{U}_{CF+DF}$, which remains to be *CF-acyclic* (no cycles among control-flow edges).

Most data structures are processed by units that implement the *visitor* pattern [55]. 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 12. 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.

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

For traversing plain data structures (such as graphs), 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 for "filling" it with 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.

4.3.2.1 Input parsers

As it was discussed in Section 4.2, both input-language and input-model parsers are implemented via ANTLR parser generator. Currently, PorthosC does not consider C preprocessor instructions (it ignores them). For implementing the inter-procedural

```
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 12 — Illustration of the visitor pattern

mode of Porthos, it is crucial to support inclusion of header files (the `#include` preprocessor directive). Next step would be implementing the support of macros and conditional compilation directives, that are used often in C code. As preprocessor statements may appear in arbitrary place of a program, the preprocessor must be a stateful processing unit that reads the token stream and dynamically instrument the program by interpreting directives and expanding macros.

The input memory model language CAT is discussed in Section 2.2. The ANTLR grammar for CAT language was extracted from the parser used by herd tool (the herd project repository: `https://github.com/herd/herdtools7`) written in OCaml.

4.3.2.2 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 interpreter for OCaml-like language. However, some most commonly used functional-style expressions may be supported by mapping them syntactically to corresponding W-elements directly in the W-model constructor.

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

4.3.2.3 Y-tree constructor

The Y-tree is constructed by the stateless visitor `C2YtreeConverterVisitor` from the ANTLR syntax tree `C11Parser.MainContext` of the C language. The Y-tree constructor aborts its work with the `YParserNotImplementedException` once it meets an unsupported syntax construction. A syntax exception `YParserException` 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).

4.3.2.4 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 `JumpsResolver` that points a label to the labelled 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 4.3.2.8).

Currently, PorthosC handles only the primitive C types (such as `int`, `char`, `float`, etc.), which are modelled on the X-level by the enumeration `XType`. The type analysis algorithm should consider the type aliases supported by C language (defined by the `typedef` instruction), which is not implemented yet. 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.

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 declared as a parameter of the process function; and

– a variable exported by the `extern` keyword (in the kernel-analysis mode, the functions `EXPORT_SYMBOL` and `EXPORT_SYMBOL_GPL` also export symbols for dynamic linking [3]).

4.3.2.5 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 13 illustrates the relationship between main 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 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 has additional modules `XMemoryManager`, `XHookManager` and `XTypeManager` that provide specialised functionality.

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

– the *program* interpreter `XProgramInterpreter` that is the compositional element that dispatches calls to currently maintained process interpreter;
– 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,

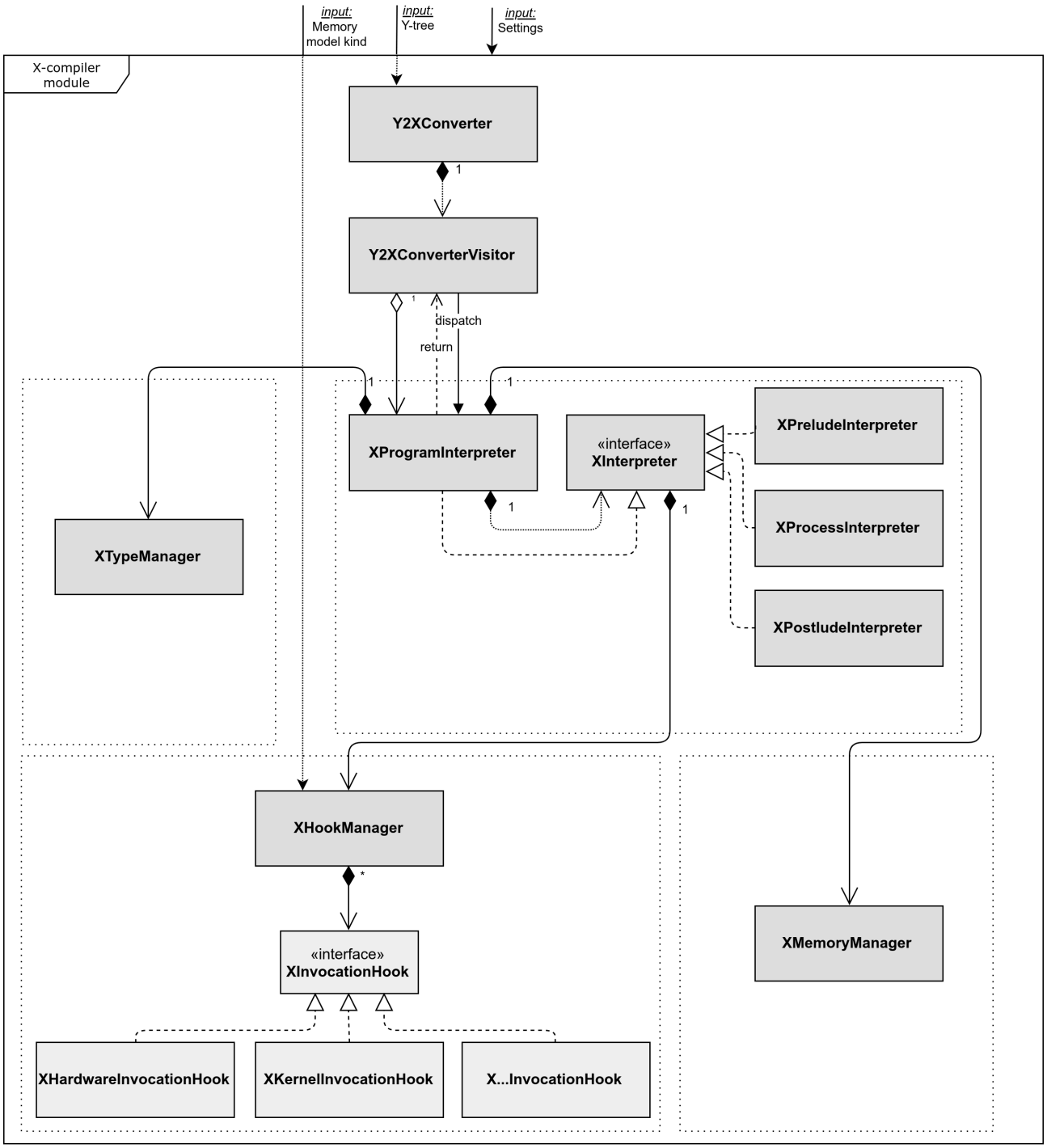


Figure 13 — Main components of the X-compilation processing unit

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 4.3.1.2). 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 14. 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 use of variables that have the same name to be declared in nested contexts, the `XMemoryManager` should carry the stack of block contexts for local variables (which is not currently implemented).

```
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 14 — X-memory manager public interface

Invocation hooking manager. The invocation hooking module serves as a compiler knowledge base that stores the semantics of high-level functions in terms of low-level event-flow graph elements. For example, the C11 sequentially consistent load operation '`l.store(memory_order_seq_cst, r)`' whould be compiled for the `x86` architecture as the write instruction '`MOV l, r`' followed by the fence instruction

‘MFENCE’, whether for the Power architecture this code would be compiled as the ‘hwsync’ followed by the store instruction. This is called the *compiler mapping*. While the X-graph is the abstract (hardware-agnostic) low-level program representation, the compiler mapping is the only component of the compiler that produces differences between two versions of the same program compiled for different architectures.

The function invocation mechanism works as follows. 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 matches, the hook manager intercepts the function call and interprets the hook action instead of invoking the function directly. The hook manager enables only invocation hooks that are compliant with the language of the input program. 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 XLegacyInvocationHook for intercepting the compatibility-mode functions of the PorthosC input language, and XKernelInvocationHook for describing the Linux kernel-specific functions. The same mechanism (with minor modifications) can be used for processing assembly language calls. An invocation hook can model any type of operations (memory, fence, computational, etc.). For instance, the XKernelInvocationHook intercepts the invocation ‘WRITE_ONCE(dst_shared, src_local)’ and replaces it with the hook action shown in Appendix A.2 (as the functions READ_ONCE, WRITE_ONCE, and ACCESS_ONCE may be modelled as volatile memory_order_relaxed accesses [50], this invocation hook simply emits a memory event without emitting any synchronisation events). For instance, the XKernelInvocationHook intercepts the invocation ‘WRITE_ONCE(dst_shared, src_local)’ and replaces it with the hook action shown in Appendix A.2 (as the functions READ_ONCE, WRITE_ONCE, and ACCESS_ONCE may be modelled as volatile memory_order_relaxed accesses [50], this invocation hook simply emits a memory event without emitting any synchronisation events).

Interpreter. The X-program interpreter is invoked by the Y2XConverterVisitor which recursively walks down the 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 A.4. Note the clear modular independence of the X-interpreter: it knows nothing about Y-tree or other internal representations, all its interfaces use only X-level elements (all conversion with Y-entities is done by the `Y2XConverterVisitor`).

All interface methods of the X-interpreter 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 an optimisation that removes unused computations from the model (otherwise, for instance, the assignment '`x = 1 * (2 + 3)`' would be compiled into 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)));`'). Note, if the computation event has not been 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 4.3.1.2). 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 (Y-level) instruction 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 the linear one that sets up the primary edge from the previous-event (last processed) to the currently processed one and updates the previous-event. 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 branch (later the interpreter will set up jumps to that event);
- `WaitingNextLinearEvent`: the default 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 interpretation and encoding, jump events (no-computation events) are present in the flow-graph, however they can be removed from it with rebinding ingoing and outgoing edges. As an example of usage of the interpreter, consider the code of the `Y2XConverterVisitor` for processing `if-then-else` statement of the Y-tree in Appendix A.3.

4.3.2.6 X-graph unroller

In order to be encoded into an SMT-formula, the compiled graph needs to be acyclic [57]. For that, we perform the *unrolling* (also called *unwinding*) transformation: all cycles are unrolled so that the result number of events in a trace of the X-graph does not exceed the user-defined bound k (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 `CitationService` discussed in Section 4.3.1.1 that in perspective may be extended for providing this information along with the text citation of the original code.) . This definition of the unrolling bound differs from one used by Porthos v1, where each cycle was executed k times. The meaning of a bound was changed in order to increase predictability of the size of the result SMT-formula. Note that recursive function calls create unconditional back-jumps in the event-flow graph, which can also be recognised as a loop and be affected by the unrolling algorithm (however, currently not supported).

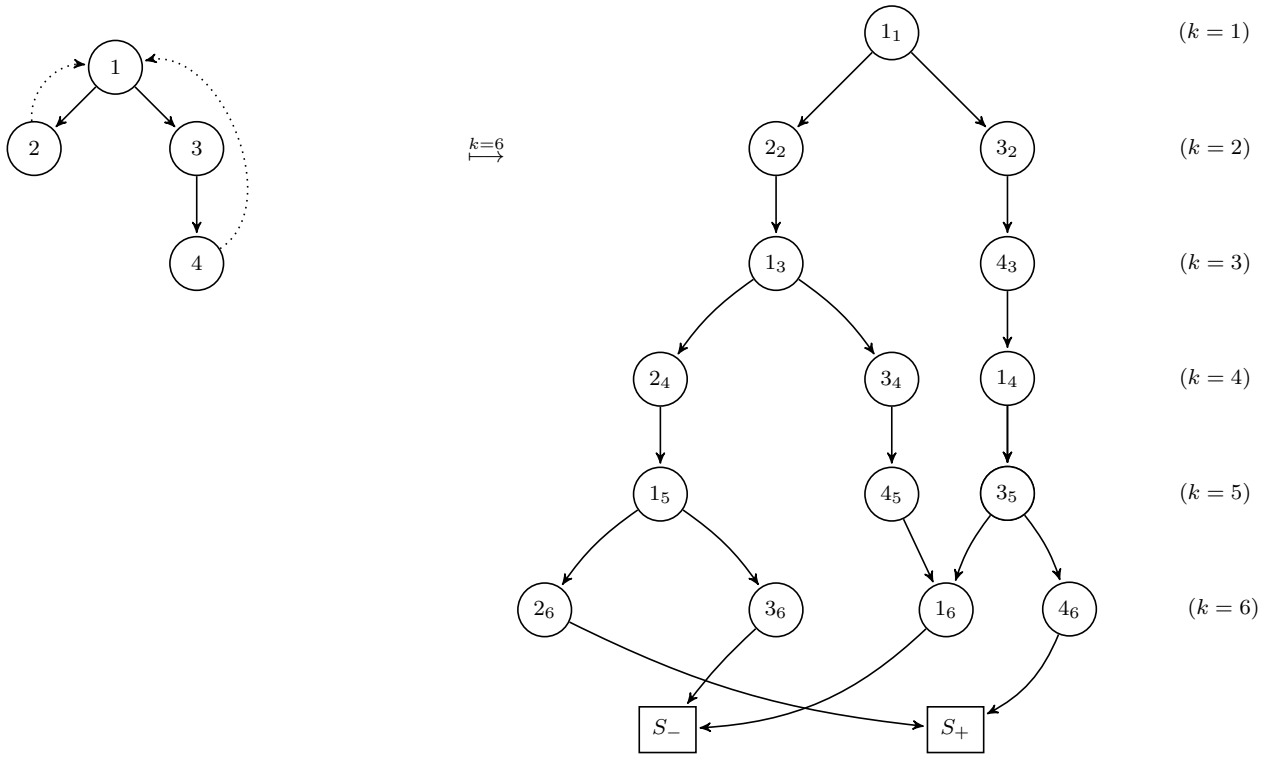


Figure 15 — Example of the flow graph unrolling up to bound $k = 6$

Considering the new meaning of the unrolling bound, the execution in the unrolled graph can be *completed* (the loop has been executed a whole number of times) or *uncompleted* (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 15 illustrates the unrolling for the left-hand side cyclic control-flow graph (the square node S_+ denotes the sink node for completed executions, and the S_- denotes the sink for uncompleted executions).

The unrolling procedure is performed by the `XFlowGraphUnroller`. For unrolling the flow-graph, we perform the *Deep-First Search (DFS)* while counting the X-events (for the unrolling bound) and keeping track of the depth stack (for detecting back edges needed for determining the type of sink node). Back-edges are determined via the depth stack (each next event is pushed onto the stack; if the new event is present in the stack, the last processed edge is a back-edge). Also, during the unrolling each next event increments the *unrolling depth counter* – the non-zero integer *event reference-ID* stored by each X-event (only events of a non-unrolled graph can have the zero-valued reference-ID) As an X-event is immutable, it has the

method ‘`XEvent asNodeRef(int refId)`’ that clones it 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 use of `HashMap` and Guava’s `ImmutableMap` for storing events, proper setting of the hash-code methods is a crucial programming task).

4.3.2.7 X-graph data-flow constructor

Once the graph is unrolled, it can be augmented by data-flow edges. As it was discussed in Section 4.3.1.2, 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.

4.3.2.8 Z-formula encoder

The Z-formula encoder accepts as input the unrolled X-graph and the input memory models. It visits the recursive memory model AST and each its derived relation definition (recall the ‘`let`’ keyword of the CAT language) it transforms to the set of SMT-clauses, generated on the basis of the program events with respect to the encoding scheme described in Chapter 3. As the Z-formula representation is not yet implemented, the Z-formula encoder encodes the X-graph and W-model directly into the SMT-formula via Java API of the Z3 solver (the program domain encoding is currently a part of the Z-formula encoder until the full X-graph$_{\text{CF+DF}}$ is implemented).

4.3.3 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.

5 EVALUATION

5.1 Comparison with Porthos v1

Consider two equivalent functions `t0` in Figure 16: in the C language (left) and in the Porthos v1 input language (right). The functions contain three nested `while`-loops and thus can serve as an illustration of differences in the program compilation and unrolling processes between Porthos v1 and PorthosC.

```
void t0(int &x) {
     int a = 1;
     int c = 1;
     while (a == 1) {
         int b = 1;
         while (b == 1) {
             while (c == 1) {
                 x = c;
             }
             x = b;
         }
     }
     x = a;
 }
```

(a) An example in the C language

```
{ x }
thread t0 {
     a <- 1;
     c <- 1;
     while (a == 1) {
         b <- 1;
         while (b == 1) {
             while (c == 1) {
                 x := c
             };
             x := b
         };
     };
     x := a
 }
```

(b) An example in the Porthos v1 input language

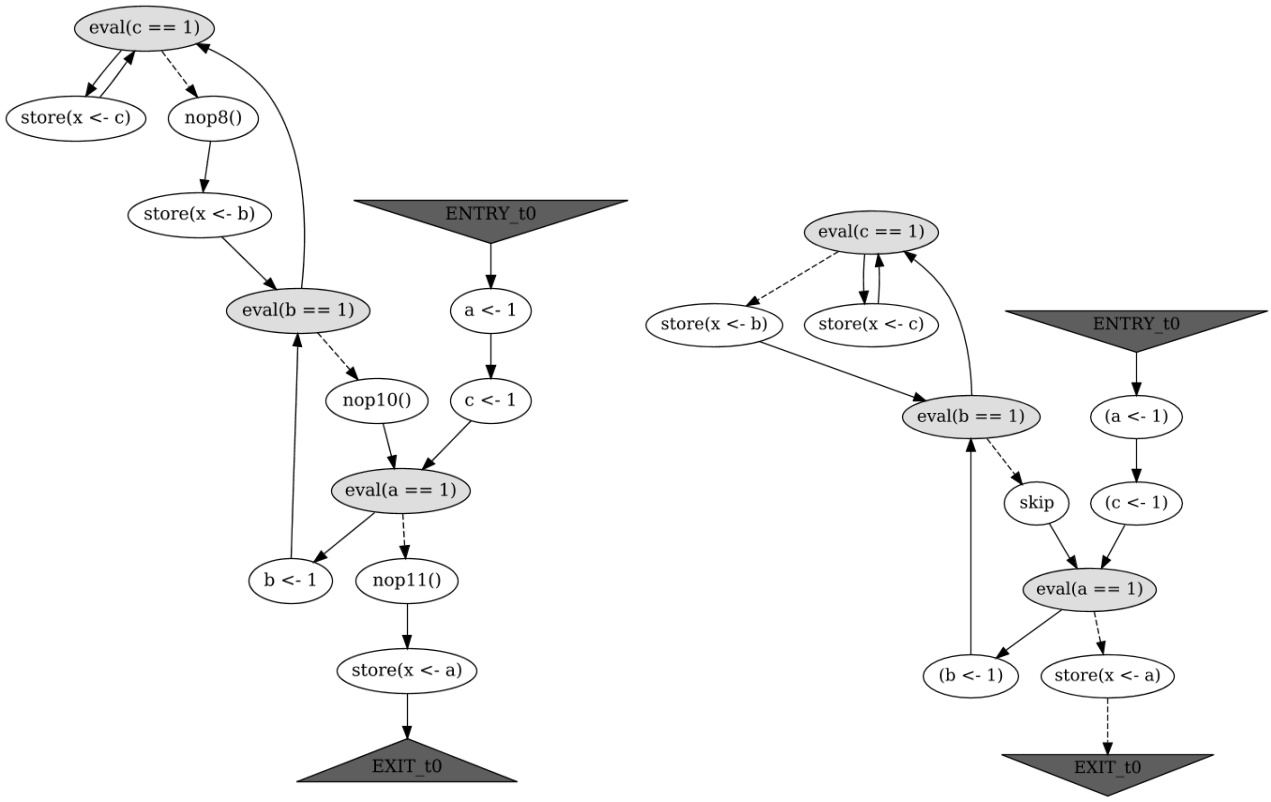
Figure 16 — Example: A demonstrative cyclic function

The functions are analysed for reachability of the final state by PorthosC and modified version Porthos v1 which is able to render the program AST to an event-based control-flow graph (for that, the branching statement `if-then-else`, the `while` loop and the `sequence` statement are expanded recursively and the head and the tails of each instruction are bound by the method processing the parent). The graph generation is performed via the open-source library `Graphviz` [27]).

5.1.1 Compilation

Figure 17 illustrates the data structure which the functions in Figure 16 are compiled to. The left-hand side picture represents the non-unrolled X-graph$_{CF}$ generated by PorthosC, and the right-hand picture represents the AST generated by Porthos v1.

In both pictures, the writes 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 graphs contain a single source event and a single sink event represented by the dark-grey triangles (in fact, the graph produced by Porthos v1 does not have sink and source nodes, but they were added to the picture for demonstrative purposes). For clarity, all branching events, that in current example serve as the conditional events of loops, are highlighed with light-grey colour.



(a) The compiled X-graph of the function in Figure 16a

(b) The compiled AST of the function in Figure 16b

Figure 17 — The control-flow graphs of the functions represented in Figure 16

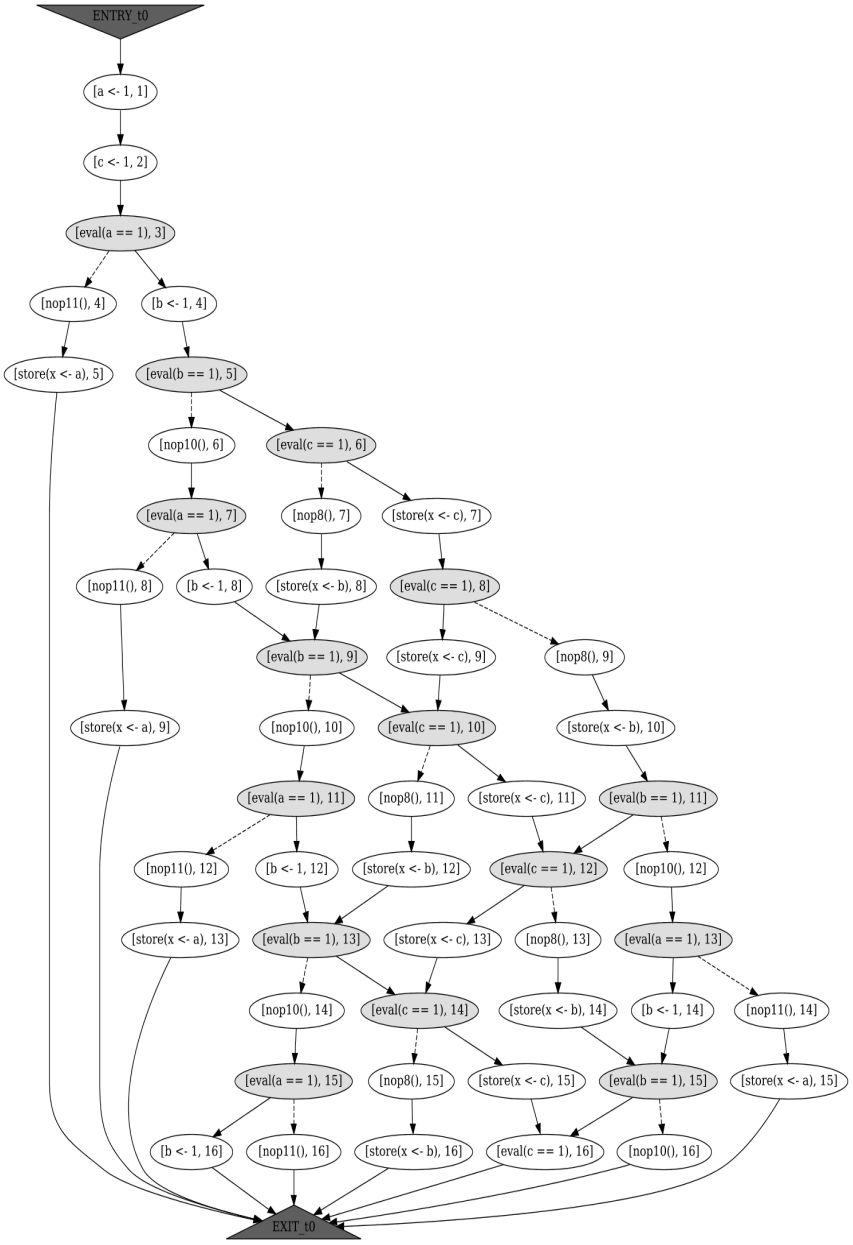
Note that two compiled graphs are equivalent up to the extra `nop`-events in the PorthosC graph, that are necessary for correct encoding as it was discussed in Section 3.2.1, and `skip`-events in Porthos v1 graph. However, the unrolled graphs presented in Figure 18 are different as the tools use different unrolling algorithms. The labels of events in the left-hand side picture (produced by PorthosC) are augmented by the unrolling depth number, which is separated from the event label by comma.

5.1.2 Unrolling

The unrolling algorithm used by Porthos v1 (right-hand side picture) unrolls *all* loops k times (where k is the unrolling bound), and the unrolling algorithm of PorthosC unrolls loops so that not more than k events are executed. As it is illustrated by the picture, the new algorithm produces a better set of program executions (for example, the unrolled graph of Porthos v1 does not contain executions of the inner loops more than $k = 2$ times in a raw, which makes the old unrolling algorithm not complete). As the new unrolling algorithm is based on the DFS algorithm, it discovers *all* possible paths, therefore the result graph contains *all* possible executions and thus the program analysis performed on this graph can be complete up to bound k.

Note that the unrolled graph produced by PorthosC does not necessarily 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 '`[b <- 1, 8]`' and '`[store(x <- b), 8]`' (produced by executions of the first iteration of the `while` loop) lead to the same event '`[eval(b == 1), 9]`' (the first event of the second iteration of the second loop).

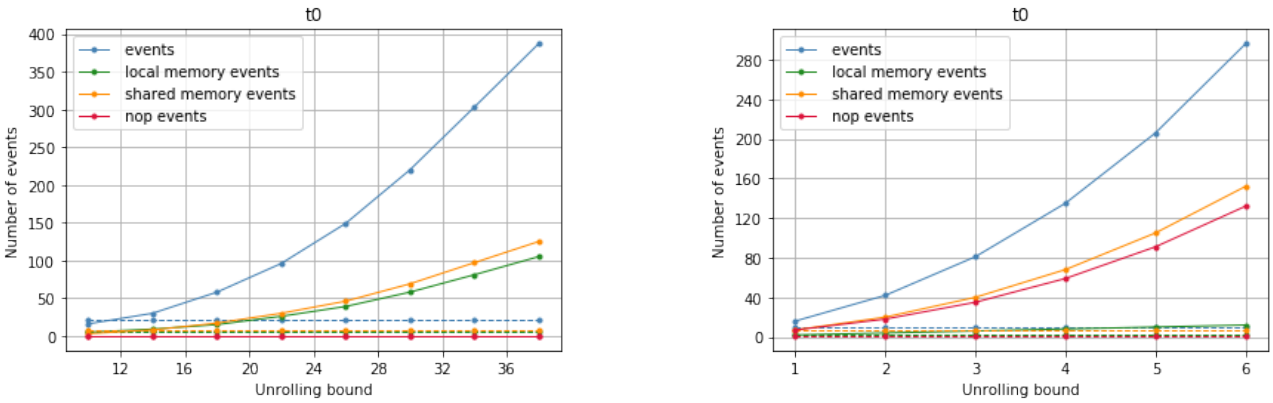
As another example of program liable for analysis consider the Dekker's algorithm for mutual exclusion of two processes, originally described by Dijkstra [26]; the program is presented in Appendix A.5. For testing Porthos v1, we used the same file in old Porthos input language (`dekker.pts`), which was used in evaluation tests in the original paper [58]. For performing tests, PorthosC was slightly modified in the branch *ay/test-timing* (the modifications were pushed into the Porthos v1 repository `github.com/hernanponcedeleon/Dat3M`) for printing detailed timing information and



(a) The unrolled X-graph of the function in Figure 16a

(b) The unrolled AST of the function in Figure 16b

Figure 18 — The unrolled control-flow graphs of the functions represented in Figure 16



(a) The graph unrolled by PorthosC

(b) The graph unrolled by Porthos v1

Figure 19 — Dependency of number of events in the unrolled Dekker's algorithm on the unrolling bound

statistics about number of events (commit f8a6c97f) and writing this information in strict *json* format so that it can be easily processed later by a script that analyses and prints the results (commit 48ba7161). The version of PorthosC used for the tests is actual up to commit 01dde6b1 of the PorthosC GitHub repository. Both Porthos v1 and PorthosC are run in the state reachability mode for TSO memory model (see sample output of PorthosC in Appendix A.6).

Figure 19 illustrates the dependency of overall number of events in all unrolled event-flow graphs of the thread `t0` on the unrolling bound k for the `Dekker` test (the left-hand side picture illustrates the case for PorthosC, and the right-hand side picture illustrates the case for Porthos). The solid lines denote dependency of number of unrolled events on the unrolling bound, when the dashed lines denote the number of events in non-unrolled graphs. The blue line depicts the dynamics of the overall number of events in the graph (including memory events, nop events, etc.). Initially, the thread `t0` has 22 events. As in the unrolling scheme of Porthos v1, each loop is executed exactly k_1 times, therefore if $k_1 = 6$, the loop has been executed at least 6 times by producing around 300 events. Nonetheless, in the new unrolling scheme used by PorthosC, the same number of events is produced by the unrolling bound $k_2 = 34$, which contains approximately 1.5 executions of the 22-event loop. Therefore, according to rough estimates the new encoding scheme produces $6/1.5 = 4$ times more events than the older one.

The same dynamics can be seen with shared memory events (yellow lines). However, the new algorithm produces much more local memory events (green line on

the left-hand side picture), which grow exponentially with the unrolling bound, when the number of local memory events produced by the old algorithm (green line in the right-hand side picture) grows relatively slowly. This can be explained by the need to copy all shared variables to local ones when performing computation over them. On the other hand, the right-hand side picture reveals exponential-like growth of the number of `nop`- (or `skip`-) events, that are necessary for implementing the control-flow as an AST, however constitute a redundant part of the model (in the new unrolling scheme, the number of `nop`-events remains to be relatively constant).

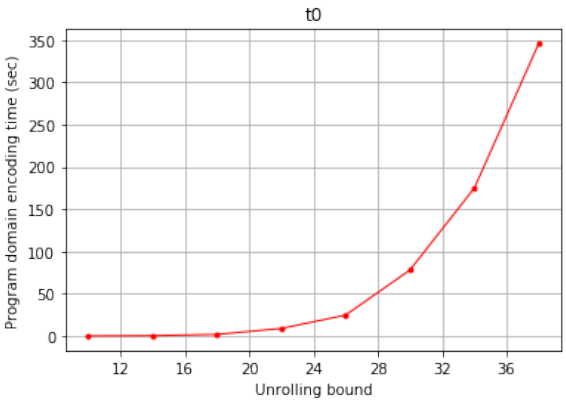
As the new unrolled graph has all possible executions unlike the older one, by default PorthosC uses the new unrolling scheme. However, in some applications the older unrolling mode may be useful (for example, where the user can use other instruments to prove the limitations on the number of executions of a loop), thus the user should be able to change configure the unrolling algorithm (support for the older algorithm is left to a future work).

5.2 Performance evaluation

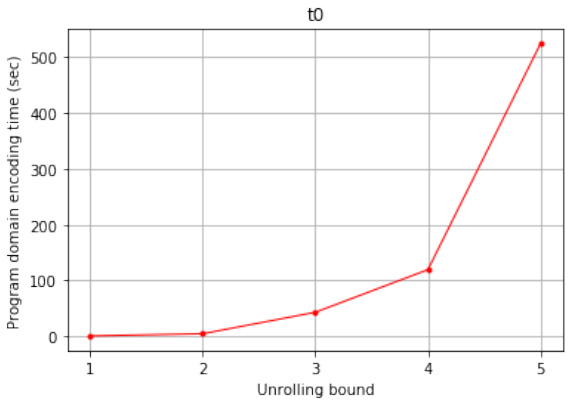
Performance evaluation has also been performed on the example of the `Dekker` test. For time benchmarking we ran the tools 5 times and computed the median of the encoding time. Benchmarking was performed on the Linux machine, 8 Gb RAM, 8 cores Intel(R) Core(TM) i7-3632QM CPU @ 2.20GHz, running under Java(TM) SE Runtime Environment (build 1.8.0_161-b12) (Java virtual machine was configured by default parameters with maximum heap size 1982 Mb). The time was measured by the tool itself via the native Java method `System.currentTimeMillis`.

5.2.1 State reachability analysis

The sample output of PorthosC in the state reachability analysis mode is presented in Appendix A.6. The output includes detailed time measurements for each



(a) Encoding of the graph unrolled by PorthosC



(b) Encoding of the graph unrolled by Porthos v1

Figure 20 — Dependency of program domain encoding time (in seconds) on the unrolling bound

stage of the analysis, and optionally statistical information about the unrolled program (number of events, transitions, etc.)

As the unrolling algorithm has been changed, the number of events after unrolling differs considerably, therefore the correct performance comparison with Porthos v1 is not manageable as the performance of the tools depends directly on number of events.

Figure 20 illustrates the dependency of the time spent for the program domain encoding (for the process `t0`) on the unrolling bound. Since the program domain encoding time is heavily dependent on the number of events (specifically, shared and local memory events as the encoding function contains several multiple nested loops over memory events), the dependency graph follows the results presented in Figure 19 and resembles exponential relationship between encoding time and unrolling bound.

An example with detailed execution time information for PorthosC can be found in Appendix A.6 (part ‘`timers`’ of the output). Note that the time spent for the interpretation (0.222 sec) and unrolling (0.088 sec) stages remains negligible comparing to the full execution time (40.679 sec). Therefore, we conclude that the new architecture implies no performance overhead comparing to the previous version of the tool.

5.2.2 Portability analysis

For evaluating the tool working in the portability analysis mode, we used the same files `Dekker.c` and `Dekker.pts` tested for the state-portability from TSO to SC: As the portability analysis requires compiling the program under two memory models, the overall program domain encoding time 72.744 sec with unrolling bound $k_2 = 27$ is almost double as the same time in reachability analysis mode (Porthos v1 shows program domain encoding time 41.027 sec with the bound $k_1 = 3$).

5.2.3 New features

Current section demonstrates some new features of PorthosC, which is hard or impossible to implement on the top of its predecessor Porthos v1 (see detailed discussion in Chapter 4).

5.2.4 Interpretation of a code with an arbitrary control-flow

The following example in Figure 21 illustrates the interpretation of the program that contains *continue*, *break* and two `goto` instructions (both forward and backward), that can produce an arbitrary control-flow graph. As an addition, the example illustrates the interpretation of complex expressions that involve shared variables (such as the condition of the `while`-loop in the function `thread_0`). Note that the `entry event` of the loop is not necessarily the guard computation event. In the case when a computation event involves shared variables, all of them need to be copied to a register before the computation event is emitted. The first event that performs such a copy becomes an entry event of the `while`-loop, which the loop back edge and all `continue`-jumps point to.

```
{
  int x = 1, *y; //initialisation
}

void thread_0(int &x, int &y) {
  L0: x = 0; //labelled statement
  int r;
  while (x * (5 + 4 / 2) % 3 == 1) {
    if (x != 0) {
      goto L0; //backward jump
    }
    if (y > 6) {
      continue; //jump to the loop entry
    }
    else if (++y > 7) {
      r = r + 10;
      break; //jump to the loop exit
    }
    else {
      goto L1; //forward jump
    }
    r = 11;
  }
  y = x + 1;
  L1: x = r; //labelled statement
}

void thread_1() {
  while (true) {
    if (x > 7)
      break;
  }
  y = x;
}

exists (y == x + 1 && thread_0:r > 21)
```

Figure 21 — Example: A litmus test in C with an arbitrary control-flow

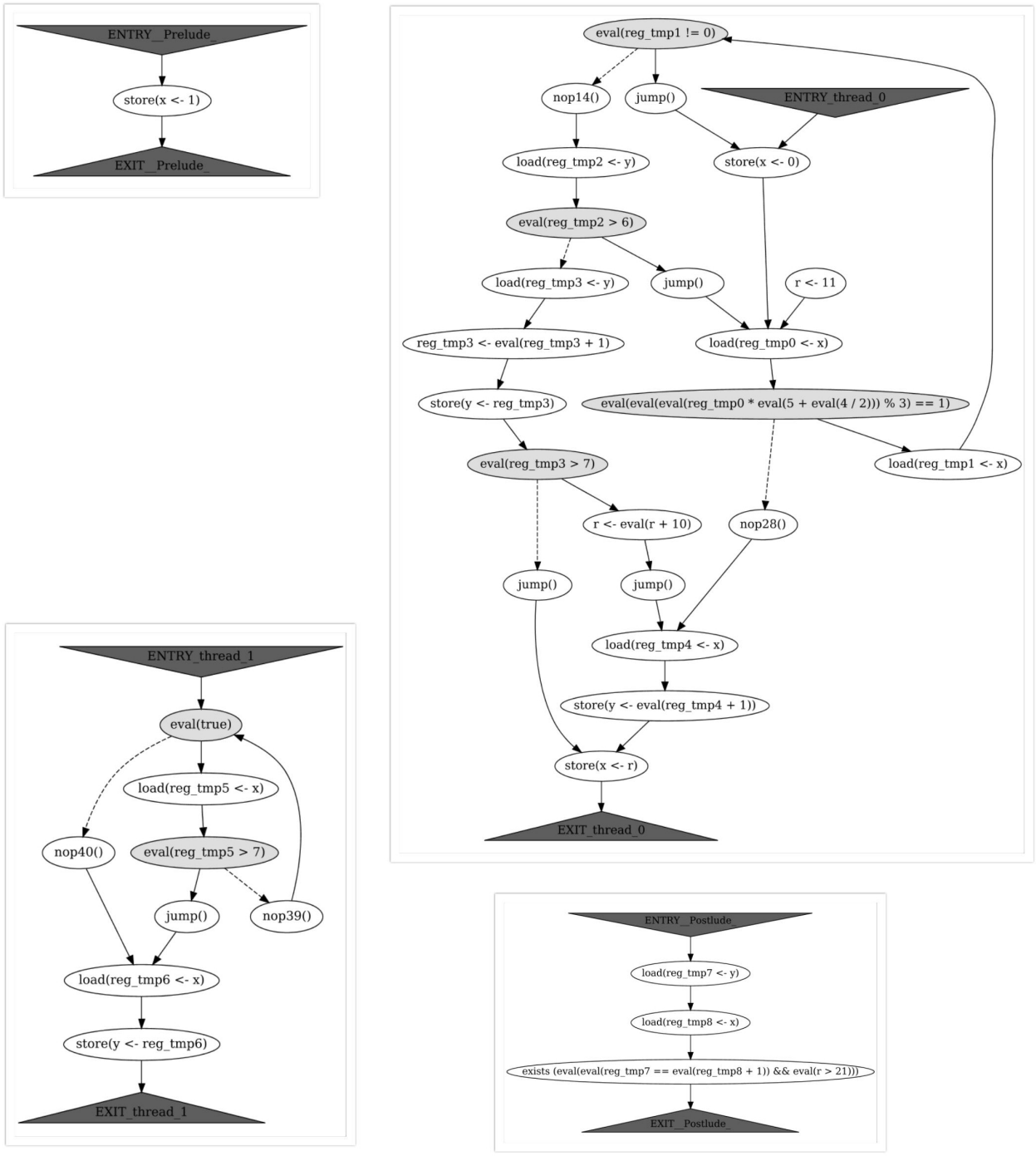


Figure 22 — Control-flow components of the event-flow graphs for the code in Figure 21

The four flow-graphs for the code in Figure 21 are shown in Figure 22. There are two small linear flow-graphs for *prelude* (litmus-specific variables initialisation and shared variables declaration) and *postlude* (assertion) definitions. As the initialisation statement assigns value only to one shared variable, its flow-graph has only a single event

(although variable declarations change the state of the interpreter while compilation, they do not produce events). The postlude has shared variables involved into the assertion, therefore they need to be copied into local registers beforehand. Note how the unconditional jump instructions `goto`, `continue` and `break` are compiled in the event-flow graph for processes `thread_0` and `thread_1`.

5.2.5 Extensible compiler mapping

Apart from redesigning the tool architecture and extending the input language, we added support for basic constructions used in *Kernel litmus tests* [9]. Although, current version of PorthosC only supports some basic macros of the Linux kernel (such as `READ_ONCE` and `WRITE_ONCE` for atomic load and store with relaxed memory ordering) as the support for kernel-specific memory barriers goes out of the scope of current thesis. Note that, comparing to the Porthos v1, adding support for new functions in PorthosC does not require changing the input language grammar, this is carried by invocation hooking mechanism described in Section 4.3.2.5.

As an example, consider the Store Buffering litmus test for the Linux kernel in Figure 23 (which is similar to the one presented in Figure 1). When verifying this litmus test by PorthosC in the state reachability mode, the test passes for TSO memory model and fails for SC model.

```
{ int x = 0; int y = 0;}

P0(volatile int& y, volatile int& x) {
  int r0;
  WRITE_ONCE(x,1);
  r0 = READ_ONCE(y);
}

P1(volatile int& y, volatile int& x) {
  int r0;
  WRITE_ONCE(y,1);
  r0 = READ_ONCE(x);
}

exists(x == 0 && y == 0)
```

Figure 23 — SB litmus test for the Linux kernel memory model

6 SUMMARY

6.1 Solved tasks and contributions

During the work on this thesis, we solved the following research and engineering problems:

1. We have studied existing weak memory model-aware analysis approaches, tools and frameworks to gather deep understanding of the problem (Sections 1.2 and 3.1 and Chapter 2).
2. Then, we explored existing implementation of the portability analysis tool *Porthos* and designed the new tool *PorthosC* that accepts as input the larger subset of the C language and supports its fundamental concepts by design, has modular architecture and disposes the extensible knowledge base of the domain-specific functions. The key aspects of the architectural design and implementations are the following:
    a. The first stumbling block in extension of the input language was the Porthos v1 program input parser, which performs the full semantic analysis of the program. Although it works for a small subset of the C language, the rich and expressive language such as C requires several stages of analysis before the compilation stage. To handle this, we implemented the processing units that analyse the AST of the program on the *pre-compilation* stage (see Section 4.3.2.4).

    As an example, the *variable kind analysis* (determining whether they are shared or local) is performed by Porthos v1 syntactically, depending on the operator or the function that uses the variable. In contrast, PorthosC traverses the program AST and collects information of all variables accessed in the program during the pre-compilation stage (see Section 4.3.2.4). Currently, PorthosC considers a variable to be shared if it was declared as a pointer, or its address is accessed at least once by the code of any process, or it was declared as a parameter of the function that

defines a process, or it was exported by `extern` keyword or an exporting function.

b. In addition, PorthosC supports *new syntactic constructions* of the C language: `break` and `continue` jumps, function invocations, multiple-variable declarations (as '`int a, b=2, z;`'), arbitrary expressions allowed by the C11 standard [32], and some others (see Section 4.2).

c. As the C language supports unconditional `goto`-jumps, the control-flow graph can have an arbitrary structure, which can not be modelled solely by the program AST, therefore PorthosC compiles the AST to the low-level hardware-agnostic program representation X-graph. For that, we implemented the *full C interpreter* discussed in Section 4.3.2.5.

d. Originally, the control-flow instructions were encoded directly into the SMT-formula [58]. In contrast, PorthosC encodes the low-level X-graph representation in accordance with the *new control-flow encoding scheme* that in general follows the one proposed in [28, Chapter 5.1.2] (see Section 3.2.1). As the new encoding scheme does not add new variables to every control-flow instruction, the number of variables in the result SMT-formula is expected to be smaller.

e. Since PorthosC compiles the program AST to the X-graph before encoding it into the SMT-formula, we decided to change the *unrolling algorithm* from unwinding all loops k times (where k is the user-specified unrolling bound) to the DFS-based algorithm that explores *all possible executions* that the program can produce within k steps (see discussion on the unrolling in Sections 4.3.2.6 and 5.1.1). On the other hand, it is shown in Section 5.1.2 that the implementation of the new unrolling algorithm does not produce exponentially many dummy `nop`-events as the implementation of the old unrolling algorithm does. Another benefit of the new unrolling scheme is that it is much more configurable (i.e., the number of events grows gradually as the unrolling bound is being increased, see Figure 19). Although the new unrolling algorithm produces considerably more executions than the old one and thus requires more time for analysis, PorthosC uses the new unrolling algorithm by default as it is complete (finds all possible executions).

f. For ease of adding support for new domain-specific functions (for example, the Kernel-specific atomic write function `atomic_store`), we implemented the *invocation hooking*, a flexible mechanism for intercepting the compilation process without changing the interpreter. The invocation hooking mechanism serves as a knowledge base for the program domain, that is to be filled and extended in future. Invocation hooks are defined in Java and thus are flexible, though their extension and modification requires some knowledge of the internals of the tool. We illustrate this mechanism with the basic support for Linux kernel litmus tests (see Sections 4.3.2.5 and 5.2.3).

3. As the tests show, the *performance overhead* of the new architecture is negligible (see Section 5.2), therefore we consider the applied architectural decisions to be acceptable.

6.2 Limitations and directions for future work

Current implementation of PorthosC has the following limitations, that might possibly define the direction of the future work.

– One of the major limitation of PorthosC as a software verification tool is its sensitivity to the *combinatorial explosion* of the state space. As it was shown in Section 5.2, the number of events of the unrolled program grow rapidly as the user increases the unrolling bound. One possible way to reduce the number of the program states might be applying some traditional techniques that target the state explosion problem (such as concrete execution as a part of *concolic execution* [43] before the unrolling stage). However, this must be done carefully and with taking into account the analysing weak memory model, because otherwise it can lead to the loss of states and thus to incorrect analysis results. Nonetheless, the small-size litmus test-like programs (containing hundreds of events after the unrolling) remain to be handleable by PorthosC within reasonable time.

– For justifying the correctness of the verification that bases on the result of the input program unrolling (via the new unrolling scheme discussed in Section 4.3.2.6), one might want to be aware whether the execution in the unrolled event-flow graph has run the loop a whole number or it was interrupted on non-last event of the loop. This can be modelled with *two kinds of sink events* (completed and uncompleted) of the event-flow graph as it was discussed in Sections 3.2.1 and 4.3.2.6.
– *Extending the knowledge base* of domain-specific functions to model synchronisation primitives (the compiler mapping) can be considered as the main future research direction. Thus, to be able to process most Linux kernel litmus tests, PorthosC needs to know the semantics of the barrier and memory operation functions that are specific for the Linux kernel in order to be able to compile them into the X-graph and later encode them into the SMT-formula. Due to modular architecture of PorthosC, the extension of the knowledge base can be done by modifying solely the invocation hooking mechanism. Note that the flip side of the flexibility of invocation hooks is their complexity: being written in Java, in addition they require from the user some knowledge of general architecture of the tool, the interface methods provided by the interpreter, the class hierarchy of the X-graph elements, etc.
– The wide range of existing litmus tests may require PorthosC to *support new input languages*. The first candidates for being supported are the LISA language (Litmus Instruction Set Architecture) [5] and assembly languages of most common architectures (x86, Power, Alpha, etc.). Note that the programs in low-level languages can be easily compiled to the X-graph representation by existing compiler architecture since low-level assembly-like languages can be considered as the subset of the C language, that restricts the set of non-linear control-flow instructions to the conditional and unconditional jumps (which are supported by the current X-graph compiler of PorthosC).
– Although PorthosC has an extended support for primitive data types (integers, reals, booleans), it still is not able to handle the *complex data types*. However, as the Z3 solver supports the theory of constant-size arrays, it might be relatively easy to extend the support for constant-size arrays (declared as ‘`int arr[10];`’), enumerations and structures. Nevertheless, pointers and variable-size arrays

(allocated in the heap by functions `malloc`, `calloc`, etc.) require a stronger pre-processing analysis before being encoded into an SMT-formula.

– Currently, PorthosC can operate only in the intra-procedural analysis mode by assuming that each function provided by the user represents a separate thread (so-called litmus test-mode). One future direction of the work could be adding the *inter-procedural analysis mode* for scanning a code project instead of a separate file (see the discussion in Section 4.1). Although, processing large programs may require serious memory optimisations (for instance, in storing the full unrolled graph data structure), which can possibly lead to worsening the overall performance. In addition, when processing large code projects, PorthosC might need the memory-guarding module that tracks the memory consumption of the tool and aborts its work if necessary.
– As it was mentioned in Section 3.2.1 devoted to the encoding for the control-flow of the program, the new encoding scheme used by PorthosC allows to analyse *partial functions*. Although the current PorthosC interpreter infrastructure is not configured to perform a partial analysis, it can be easily supported.
– During the implementation of PorthosC, we have only done limited *testing of the tool*. Thus, in order to increase the stability of PorthosC, we need to cover its code by unit- and functional tests.
– One might want to compare the performance of PorthosC in cases of usage the different SMT-solvers. For that, the Z-formula abstraction layer will be useful as new SMT-solvers can be easily added (the general way to support multiple SMT-solvers is to convert the Z-formula to an SMT-LIB formula [14], that can be passed as input to the SMT-solver run as an external process).

The new tool PorthosC with the extensible program domain knowledge base can be considered as a generalised framework for SMT-based memory model-aware analysis, which can not only perform the reachability and portability analysis, but also serve as a basis for other kinds of static analysis of concurrent programs.

ABBREVIATIONS

API	Application Programming Interface
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
SMP	Symmetric Multiprocessor architecture
SMT	Satisfiability Modulo Theories problem
SSA	Static Single-Assignment form
UMC	Unbounded Model Checking
UML	Unified Modeling Language
WMM	Weak Memory Model

LIST OF FIGURES

BIBLIOGRAPHY

1. Parosh Aziz Abdulla et al. “Stateless model checking for TSO and PSO”. In: *Acta Informatica* 54.8 (2017), pp. 789–818.

2. Sarita V. Adve and Kourosh Gharachorloo. “Shared memory consistency models: A tutorial”. In: *computer* 29.12 (1996), pp. 66–76.

3. Rishi Agrawal. *Linux Kernel Workbook*. Revision d9ac4e81. 2016. URL: http://lkw.readthedocs.io/en/latest/index.html.

4. Jade Alglave. “A shared memory poetics”. In: *La Thèse de doctorat, L’université Paris Denis Diderot* (2010).

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

6. Jade Alglave and Luc Maranget. *Simulating Memory Models with Herd.* Tech. rep. 2010.

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

8. Jade Alglave et al. “Don’t sit on the fence”. In: *International Conference on Computer Aided Verification*. Springer. 2014, pp. 508–524.

9. Jade Alglave et al. “Frightening small children and disconcerting grown-ups: Concurrency in the Linux kernel”. In: *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM. 2018, pp. 405–418.

10. Jade Alglave et al. “Software verification for weak memory via program transformation”. In: *European Symposium on Programming*. Springer. 2013, pp. 512–532.

11. Jade Alglave et al. *Static over-approximation for (mutually) recursive functions in the construction of the abstract event graph*. 2014. URL: http://www.cprover.org/wmm/musketeer/rec.pdf.

12. Jade Alglave et al. “The semantics of Power and ARM multiprocessor machine code”. In: *Proceedings of the 4th workshop on Declarative aspects of multicore programming*. ACM. 2009, pp. 13–24.

13. Roberto Baldoni et al. “A Survey of Symbolic Execution Techniques”. In: *ACM Comput. Surv.* 51.3 (2018).

14. Clark Barrett, Pascal Fontaine, and Cesare Tinelli. *The SMT-LIB Standard: Version 2.6*. Tech. rep. Available at `www.SMT-LIB.org`. Department of Computer Science, The University of Iowa, 2017.

15. Mark Batty et al. “Mathematizing C++ concurrency”. In: *ACM SIGPLAN Notices* 46.1 (2011), pp. 55–66.

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

17. Ahmed Bouajjani, Egor Derevenetc, and Roland Meyer. “Checking and enforcing robustness against TSO”. In: *European Symposium on Programming*. Springer. 2013, pp. 533–553.

18. Jerry R Burch et al. “Symbolic model checking: 1020 states and beyond”. In: *Information and computation* 98.2 (1992), pp. 142–170.

19. Alessandro Cimatti et al. “NuSMV: A new symbolic model checker”. In: *International Journal on Software Tools for Technology Transfer* 2.4 (2000), pp. 410–425.

20. Edmund M. Clarke, Orna Grumberg, and Doron Peled. *Model checking*. MIT press, 1999.

21. Edmund M. Clarke et al. “Model checking and the state explosion problem”. In: *Tools for Practical Software Verification*. Springer, 2012, pp. 1–30.

22. Edmund Clarke et al. “Bounded model checking using satisfiability solving”. In: *Formal methods in system design* 19.1 (2001), pp. 7–34.

23. Edmund Clarke et al. “Progress on the state explosion problem in model checking”. In: *Informatics*. Springer. 2001, pp. 176–194.

24. Vijay D'Silva, Daniel Kroening, and Georg Weissenbacher. "A Survey of Automated Techniques for Formal Software Verification". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)* 27.7 (July 2008), pp. 1165–1178.

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

26. Edsger W Dijkstra. "Over de sequentialiteit van procesbeschrijvingen". Trans. by Martien van der Burgt and Heather Lawrence. In: (1962). URL: http://www.cs.utexas.edu/users/EWD/translations/EWD35-English.html.

27. John Ellson et al. "Graphviz—open source graph drawing tools". In: *International Symposium on Graph Drawing*. Springer. 2001, pp. 483–484.

28. Javier Esparza and Keijo Heljanko. *Unfoldings - A Partial-Order Approach to Model Checking*. Monographs in Theoretical Computer Science. An EATCS Series. Springer, 2008. ISBN: 978-3-540-77425-9. DOI: 10.1007/978-3-540-77426-6. URL: https://doi.org/10.1007/978-3-540-77426-6.

29. David Gries. *The science of programming*. Springer Science & Business Media, 2012.

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

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

32. ISO/IEC. *ISO/IEC 9899:2011 Information technology – Programming languages – C*. Tech. rep. 2011.

33. Michalis Kokologiannakis et al. "Effective stateless model checking for C/C++ concurrency". In: *Proceedings of the ACM on Programming Languages* 2.POPL (2017), p. 17.

34. Daniel Kroening and Ofer Strichman. *Decision procedures*. Vol. 5. Springer, 2008.

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

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

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

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

39. William Landi. “Undecidability of static analysis”. In: *ACM Letters on Programming Languages and Systems (LOPLAS)* 1.4 (1992), pp. 323–337.

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

41. Daniel Lustig et al. “Automated synthesis of comprehensive memory model litmus test suites”. In: *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM. 2017, pp. 661–675.

42. Daniel Lustig et al. “Coatcheck: Verifying memory ordering at the hardware-OS interface”. In: *ACM SIGOPS Operating Systems Review* 50.2 (2016), pp. 233–247.

43. Rupak Majumdar and Koushik Sen. “Hybrid concolic testing”. In: *Proceedings of the 29th international conference on Software Engineering*. IEEE Computer Society. 2007, pp. 416–426.

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

45. William Mansky, Dmitri Garbuzov, and Steve Zdancewic. “An axiomatic specification for sequential memory models”. In: *International Conference on Computer Aided Verification*. Springer. 2015, pp. 413–428.

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

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

48. Paul E McKenney. "Memory barriers: a hardware view for software hackers". In: *Linux Technology Center, IBM Beaverton* (2010).

49. Paul E. McKenney et al. *A formal kernel memory-ordering model (part 1)*. 2017. URL: https://lwn.net/Articles/718628/.

50. Paul E. McKenney et al. *P0124R4: Linux-Kernel Memory Model*. Tech. rep. ISO/IEC JTC1 SC22 WG21 (C++ Language), Sept. 2017.

51. Paul E. McKenney et al. *WG21/P0868R0: Selected RCU Litmus Tests*. 2017.

52. Leonardo de Moura and Nikolaj Bjørner. *Z3 - a Tutorial*. 2011.

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

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

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

56. Terence Parr. *The definitive ANTLR 4 reference*. Pragmatic Bookshelf, 2013.

57. Hernán Ponce de León et al. "Portability Analysis for Axiomatic Memory Models. PORTHOS: One Tool for all Models". In: *CoRR* abs/1702.06704 (2017). arXiv: 1702.06704. URL: http://arxiv.org/abs/1702.06704.

58. Hernán Ponce de León et al. "Portability Analysis for Weak Memory Models. PORTHOS: One Tool for all Models". In: *Static Analysis - 24th International Symposium, SAS 2017, New York, NY, USA, August 30 - September 1, 2017, Proceedings*. 2017, pp. 299–320. DOI: 10.1007/978-3-319-66706-5_15. URL: https://doi.org/10.1007/978-3-319-66706-5_15.

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

60. Susmit Sarkar et al. "Understanding POWER multiprocessors". In: *ACM SIGPLAN Notices* 46.6 (2011), pp. 175–186.

61. Jaroslav Ševčík. "Program transformations in weak memory models". In: (2009).

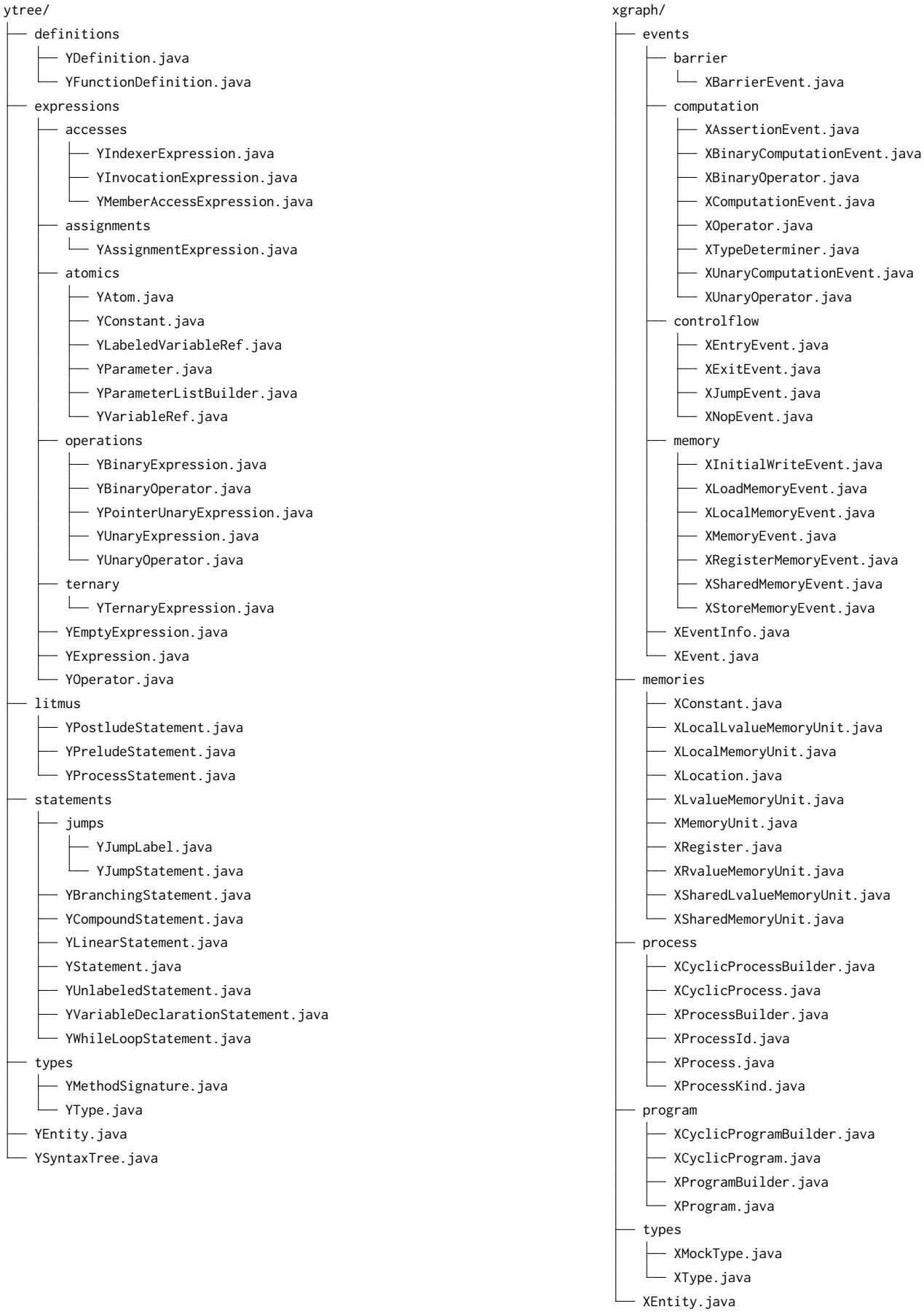
62. Ofer Shtrichman. "Tuning SAT checkers for bounded model checking". In: *International Conference on Computer Aided Verification*. Springer. 2000, pp. 480–494.

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

64. Viktor Vafeiadis et al. "Common compiler optimisations are invalid in the C11 memory model and what we can do about it". In: *ACM SIGPLAN Notices* 50.1 (2015), pp. 209–220.

65. John Wickerson et al. "Automatically comparing memory consistency models". In: *ACM SIGPLAN Notices*. Vol. 52. 1. ACM. 2017, pp. 190–204.

Appendix A.1 File trees of Y-tree and X-graph representations



Appendix A.2 Example of the invocation hook for intercepting the Linux kernel-specific functions

```
class XKernelInvocationHook extends XInvocationHookBase
       implements XInvocationHook {
 ...
 @Override
 public XInvocationHookAction intercept(String functionName) {
   switch (functionName) {
     case "WRITE_ONCE": {
       return new XInvocationHookAction((receiver, arguments) -> {
         if (arguments.length != 2 || receiver != null) {
           return null;  // do not intercept
         }

         XMemoryUnit destUnit = arguments[0];
         if (!(destUnit instanceof XSharedLvalueMemoryUnit)) {
           throw new XMethodInvocationError(methodName,
               "arg 1: not a shared l-value memory unit: " + destUnit);
         }
         XSharedLvalueMemoryUnit dest =
               (XSharedLvalueMemoryUnit) destUnit;

         XMemoryUnit srcUnit = arguments[1];
         if (!(srcUnit instanceof XLocalMemoryUnit)) {
           throw new XMethodInvocationError(methodName,
               "arg 2: not a local memory unit: " + srcUnit);
         }
         XLocalMemoryUnit src = (XLocalMemoryUnit) srcUnit;

         return program.emitMemoryEvent(dest, src);
       });
   }
   case "READ_ONCE": {
     ...
   }
 }
}
```

Appendix A.3 Excerpt from the Y-to-X converter for interpreting branching statements

```
public class Y2XConverterVisitor implements YtreeVisitor<XEntity> {
  ...
  @Override
  public XEvent visit(YBranchingStatement node) {
    program.startBlockDefinition(XInterpreter.BlockKind.Branching);

    program.startBlockConditionDefinition();
    XEntity conditionEntity = node.getCondition().accept(this);
    XComputationEvent condition = tryEvaluateComputation(conditionEntity);
    program.finishBlockConditionDefinition(condition);

    program.startBlockBranchDefinition(XInterpreter.BranchKind.Then);
    node.getThenBranch().accept(this);
    program.finishBlockBranchDefinition();

    YStatement elseBranch = node.getElseBranch();
    program.startBlockBranchDefinition(XInterpreter.BranchKind.Else);
    if (elseBranch != null) {
      elseBranch.accept(this);
    }
    else {
      program.emitNopEvent(); // needed for encoding
    }
    program.finishBlockBranchDefinition();

    program.finishNonlinearBlockDefinition();

    return null; //statements return null
    }
}
```

Appendix A.4 Public interface methods of the X-interpreter

```
public interface XInterpreter {

XProcessId getProcessId();
void finish();
XProcess getResult();

// emitting new events:
XEntryEvent emitEntryEvent();
XExitEvent emitExitEvent();
XBarrierEvent emitBarrierEvent(XBarrierEvent.Kind kind);
XJumpEvent emitJumpEvent();
XJumpEvent emitJumpEvent(String jumpLabel);
void markNextEventLabel(String jumpLabel);
XNopEvent emitNopEvent();
XAssertionEvent emitAssertionEvent(XBinaryComputationEvent assertion);
XLocalMemoryEvent emitMemoryEvent(XLocalLvalueMemoryUnit destination,
                                  XLocalMemoryUnit source);
XSharedMemoryEvent emitMemoryEvent(XLocalLvalueMemoryUnit destination,
                                   XSharedMemoryUnit source);
XSharedMemoryEvent emitMemoryEvent(XSharedLvalueMemoryUnit destination,
                                   XLocalMemoryUnit source);
XComputationEvent createComputationEvent(XUnaryOperator operator,
                                         XLocalMemoryUnit operand);
XComputationEvent createComputationEvent(XBinaryOperator operator,
                                         XLocalMemoryUnit operand1,
                                         XLocalMemoryUnit operand2);

// non-linear statements interpretation:
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);

}
```

Appendix A.5 Dekker's mutual exclusion algorithm 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);
        }
      }
    }
}
exists (turn == 10)
```

Appendix A.6 Sample of the verbose output of PorthosC in the portability analysis mode

```
$ java PorthosC --reachability --input benchmarks/Dekker.c --bound 27 --target TSO -v
0.400: Interpretation...
0.632: Unrolling...
0.805: Program encoding...
1.019: Program domain encoding...
36.945: Memory model encoding...
40.281: Solving...
{
  "result": "NonReachable",
  "options": {
    "inputProgramFile": {
      "path": "benchmarks/Dekker.c"
    },
    "sourceModel": "TSO",
    "unrollingBound": 27
  },
  "timerMain": {
    "elapsedTimeSec": 40.679
  },
  "timers": {
    "Interpretation": {
      "elapsedTimeSec": 0.222
    },
    "ProgramDomainEncoding": {
      "elapsedTimeSec": 35.925
    },
    "Solving": {
      "elapsedTimeSec": 0.397
    },
    "Unrolling": {
      "elapsedTimeSec": 0.088
    },
    "MemoryModelEncoding": {
      "elapsedTimeSec": 3.335
    },
    "ProgramEncoding": {
      "elapsedTimeSec": 0.214
    }
  },
  "processStatistics": {
    "_Prelude_": {
      "XBarrierEvent": 0,
      "XNopEvent": 0,
      "_XEdgePrimary": 4,
      "_XEdgeAlternative": 0,
      "XLocalMemoryEvent": 0,
      "XSharedMemoryEvent": 3,
      "XControlFlowEvent": 0,
      "XComputationEvent": 0,
      "XEvent": 4,
      "XMemoryEvent": 3
    },
    "_Postlude_": {
      "XBarrierEvent": 0,
      "XNopEvent": 0,
      "_XEdgePrimary": 3,
      "_XEdgeAlternative": 0,
      "XLocalMemoryEvent": 0,
      "XSharedMemoryEvent": 1,
      "XControlFlowEvent": 0,
      "XComputationEvent": 1,
      "XEvent": 3,
      "XMemoryEvent": 1
    },
    "t0": {
      "XBarrierEvent": 0,
      "XNopEvent": 0,
      "_XEdgePrimary": 22,
      "_XEdgeAlternative": 4,
      "XLocalMemoryEvent": 6,
      "XSharedMemoryEvent": 7,
      "XControlFlowEvent": 0,
      "XComputationEvent": 8,
      "XEvent": 22,
      "XMemoryEvent": 13
    },
    "t1": {
      "XBarrierEvent": 0,
      "XNopEvent": 0,
      "_XEdgePrimary": 22,
      "_XEdgeAlternative": 4,
      "XLocalMemoryEvent": 6,
      "XSharedMemoryEvent": 7,
      "XControlFlowEvent": 0,
```

```
      "XComputationEvent": 8,
      "XEvent": 22,
      "XMemoryEvent": 13
    }
  },
  "processStatisticsUnrolled": {
    "_Prelude_": {
      "XBarrierEvent": 0,
      "XNopEvent": 0,
      "_XEdgePrimary": 4,
      "_XEdgeAlternative": 0,
      "XLocalMemoryEvent": 0,
      "XSharedMemoryEvent": 3,
      "XControlFlowEvent": 0,
      "XComputationEvent": 0,
      "XEvent": 4,
      "XMemoryEvent": 3
    },
    "_Postlude_": {
      "XBarrierEvent": 0,
      "XNopEvent": 0,
      "_XEdgePrimary": 3,
      "_XEdgeAlternative": 0,
      "XLocalMemoryEvent": 0,
      "XSharedMemoryEvent": 1,
      "XControlFlowEvent": 0,
      "XComputationEvent": 1,
      "XEvent": 3,
      "XMemoryEvent": 1
    },
    "t0": {
      "XBarrierEvent": 0,
      "XNopEvent": 0,
      "_XEdgePrimary": 165,
      "_XEdgeAlternative": 33,
      "XLocalMemoryEvent": 43,
      "XSharedMemoryEvent": 52,
      "XControlFlowEvent": 0,
      "XComputationEvent": 46,
      "XEvent": 165,
      "XMemoryEvent": 95
    },
    "t1": {
      "XBarrierEvent": 0,
      "XNopEvent": 0,
      "_XEdgePrimary": 165,
      "_XEdgeAlternative": 33,
      "XLocalMemoryEvent": 43,
      "XSharedMemoryEvent": 52,
      "XControlFlowEvent": 0,
      "XComputationEvent": 46,
      "XEvent": 165,
      "XMemoryEvent": 95
    }
  },
  "errors": []
}
```