MASARYK UNIVERSITY
FACULTY OF INFORMATICS


# Verification of Name Service Cache Daemon with DIVINE Model Checker

MASTER'S THESIS

**Milan Lenčo**

Brno, autumn 2014

## Declaration

Hereby I declare, that this paper is my original authorial work, which I have worked out by my own. All sources, references and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

Milan Lenčo

**Advisor:** doc. RNDr. Jiří Barnat, Ph.D.

## Acknowledgement

I would like to thank my thesis advisor doc. Jiří Barnat for giving me the opportunity to join the ParaDiSe laboratory and see how the real research is done. Even though I have decided to go a different direction in the end, it is only through experience that one can make a right decision about the future career path.
Also a very special thanks to my parents who were very supportive during the study period and gave me the motivation to finish this off in the darkest of times.

## Abstract

In this thesis we present an attempt to verify a number of important safety and liveness properties of GNU `nscd`, a name service cache daemon shipped alongside the GNU C Library, using the DIVINE model checker. We give a detailed description of all the steps needed to prepare and perform verification of a non-trivial C/C++ program with DIVINE. In our approach we keep `nscd` unmodified and wrap it around with a virtual environment simulating interactions between the daemon, clients and directory services from a black-box perspective. Model checking is then performed for different configurations of the complete system. Another contribution of this thesis is a verification-ready implementation of an in-memory file system, supporting all common low-level I/O interfaces as defined by POSIX family of standards. Based on the results obtained and the effort that was required, we draw conclusions on feasibility of model checking real-world concurrent C/C++ programs in general.

## Keywords

DiVinE, `nscd`, glibc, NSS, model checking, LLVM, software verification

# Contents

# 1 Introduction

A key challenge in software development is the difficulty of finding and eliminating implementation errors. Testing is by far the most common approach used to tackle this problem. Some types of software bugs, however, do not appear unless some intricate sequence of steps is executed. Such errors are triggered rarely but often with fatal consequences and are hard to reproduce. For example, behaviours of concurrent programs do not depend only on user and environment inputs, but also on the interleaving of its execution units. In practice, this means that testing alone often leaves many errors undetected, which manifest only after days or weeks of execution.

Several authors have previously suggested model checking as a promising means to detect concurrent bugs and other types of security vulnerabilities. Its idea is to prove that a system under verification has the specified property by traversing all its reachable configurations, using techniques to explore vast state spaces efficiently.

Unfortunately, model checking of software is almost never done in practice. Even state-of-the-art model checkers are limited in use when they report an overwhelming number of false positives, counter-examples difficult to interpret for humans, or when their lengthy running time slows down other software development processes. Moreover, model checking fails to impress outside the scope of research laboratories. Apart from safety critical systems, formal verification is still lagging behind testing based methods in correctness evaluation. Industry constantly overlooks model checking as a viable method for software quality assurance for a number of reasons.

The most prominent problem is the so-called *state space explosion*, referring to the fact that the size of the state space is exponential to the size of the model description. Later in the thesis, we will briefly survey the existing methods used to fight the state space explosion problem. Since this problem is inherent to model checking, it cannot be fully eliminated and is the limiting factor for complexity of systems that can be verified using this method efficiently.

Traditionally, model checking has been based on maintaining a separate model alongside the actual implementation, providing formally exact, high-level description of the system under verification. Not only this approach requires significant effort of a specialist, but can also miss some compiler and programming language specific errors, plus the model often gets quickly outdated as the development progresses or as design is revised. Recently, this problem has been addresses and several formal verification tools

can now directly operate on the source code or on the level of a compiler intermediate representation (IR).

Many advances have been made towards making the model checking practical and easier to use, yet only few studies of real-world software model checking have been published. We believe that the lack of use cases and guidelines contributes to the prevalent omission of model checking in software development as much as its theoretical limitations.

In this thesis we present an attempt to verify a number of important safety and liveness properties of `nscd`, a name service cache daemon shipped alongside the GNU libc, using the DIVINE model checker [1]. Entire process of verification, consisting of system analysis, environment simulation and the actual model checking is described in detail. Although most of the steps and information provided here are specific to DIVINE, overall the thesis can serve as a guideline for performing model checking of (unmodified) programs. Based on the results obtained and the effort that was required, we draw conclusions on feasibility of model checking real-world concurrent C/C++ programs in general.

This thesis is organized as follows. Chapter 2 overviews model checking of software and briefly introduces the LLVM intermediate representation, on top of which DIVINE operates for model checking C and C++ code. Chapter 3 is the user guide to verification of unmodified C/C++ programs using DIVINE model checker. In Chapter 4 we provide a detailed description of `nscd` and its implementation, which is followed by the system analysis from the model checking point of view in Chapter 5. Based on this analysis, we implemented some OS and libc facilities, mostly file-system related, to emulate the environment in which `nscd` typically executes, while providing a complete description of the system so that the state space can be generated. As we describe in Chapter6, this work resulted in implementation of all common low-level I/O functions as defined by POSIX family of standards, which in turn makes it applicable for verification of other systems as well. Results of the actual model checking (`nscd` and our environment) are presented in 7. Finally, we conclude the thesis in the last Chapter 8.

# 2 Preliminaries

## 2.1 Model Checking

### 2.1.1 Introduction

In its full generality, Model checking refers to a formal verification method, that for a given model exhaustively and automatically checks whether this model satisfies a given specification. Due to the undecidability theorem [Turing, 1936], it is impossible to provide a sound and complete algorithmic solution for any sufficiently powerful programming model, therefore it is required that the state space of the model is finite, or meets some other constraints depending on the approach used.

Model is typically a transcription of a system's design, written in special formally-defined language. Nowadays, however, multiple model checkers support direct model checking of programs implemented in general-purpose languages such as Java or C++.

Specifications about the system are usually expressed as temporal logic formulas. Most commonly used logics to describe properties are LTL and CTL (with their variants). For concurrent programs, the list of frequently verified properties includes: assertion safety, deadlock-freedom, livelock-freedom, progress, etc.

### 2.1.2 Linear Temporal Logic

The purpose of the Linear Temporal Logic (LTL) in model checking is to describe behavior of a system in time. LTL is often chosen as it is convenient for expressing liveness (*something good eventually happens*) and fairness (*all possible/enabled actions are chosen infinitely often*), two very important classes of properties especially in the concurrent programming. We will see applications of both later in the thesis.

For safety properties, such as a deadlock-freedom and an absence of assertion violations, introducing LTL into model checking is superfluous and a simple reachability analysis is sufficient. Many model checkers actually omit support for temporal logics altogether, typically those which are oriented towards verification of single-threaded programs.

For the remainder of the thesis, we will concern ourselves with the *state based* LTL. Properties of a single state are reflected by validity of atomic propositions in the state and LTL formulae are interpreted over behaviours of the system represented by sequences of sets of valid atomic propositions.

Formally, the syntax of an LTL formula is defined as follows:

$$\varphi ::= \top \mid a \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid \mathcal{X}\varphi \mid \varphi_1 \mathcal{U} \varphi_2 \tag{2.1}$$

where $\top$ stands for *true* and $a$ ranges over the countable set of *atomic propositions* $AP$.

For the model checking purposes, we will interpret LTL formula on a variation of the transition system known as a *Kripke structure* [2].

Let $S$ be the set of all states, $I$ the set of initial states and $\rightarrow \subseteq S \times S$ a total transition relation. Then the Kripke structure is a triple $(s_0, S, \rightarrow)$, where $s_0 \in I$. Next we define the *Labeling function*, assigning a set of valid atomic propositions to states, as $L : S \rightarrow 2^{AP}$.

A run $\pi$ in a Kripke structure is an infinite sequence of states, $s_0 s_1 \ldots$, such that for every $i \in \mathbb{N}$, $s_i \rightarrow s_{i+1}$. Additionally, by $\pi_i$ we denote the suffix of $\pi$ starting with $s_i$.

The *validity* of an LTL formula $\varphi$ for $\pi = s_0 s_1 \ldots$ and a labeling function $L$, written as $(\pi, L) \models \varphi$, is then inductively defined as:

$$(\pi, L) \models \top \tag{2.2}$$
$$(\pi, L) \models a \iff a \in L(s_0) \tag{2.3}$$
$$(\pi, L) \models \neg\varphi \iff (\pi, L) \not\models \varphi \tag{2.4}$$
$$(\pi, L) \models \varphi_1 \wedge \varphi_2 \iff (\pi, L) \models \varphi_1 \wedge (\pi, L) \models \varphi_2 \tag{2.5}$$
$$(\pi, L) \models \mathcal{X}\varphi \iff (\pi_1, L) \models \varphi \tag{2.6}$$
$$(\pi, L) \models \varphi_1 \mathcal{U} \varphi_2 \iff \exists i \in \mathbb{N} : (\pi_i, L) \models \varphi_2 \wedge \forall j < i : (\pi_j, L) \models \varphi_1 \tag{2.7}$$

Furthermore, a number of abbreviations and derived operators are typically defined for convenience:

$$\bot \equiv \neg\top \tag{2.8}$$
$$\varphi_1 \vee \varphi_2 \equiv \neg(\neg\varphi_1 \wedge \neg\varphi_2) \tag{2.9}$$
$$\varphi_1 \implies \varphi_2 \equiv \neg\varphi_1 \vee \varphi_2 \tag{2.10}$$
$$\mathcal{F}\varphi \equiv \top \mathcal{U} \varphi \qquad (\textit{eventually}) \tag{2.11}$$
$$\mathcal{G}\varphi \equiv \neg\mathcal{F}\neg\varphi \qquad (\textit{always}) \tag{2.12}$$

Finally, we say that an LTL property holds for a given system if it holds for all its runs starting in the initial state.

### 2.1.3 Explicit Model Checking of Software

The current trend in model checking is to progress towards direct verification of the implementation, targeting most commonly used general-purpose languages and effectively avoiding the burden of providing a separate design-level description of the system. This is a significant step forward in making the method *practical* and *approachable* for the industry. Direct model checking of programs has the potential to complement testing and become part of the software development process.

From the theoretical point of view, formal definitions and algorithms used for traditional LTL model checking can be reused for direct verifications of software without substantial modifications. If we consider an imperative programming language – in practice, the paradigm used in vast majority of cases – representing the state space of a program as a Kripke structure is a straightforward process. States can be represented by the contents of memory locations accessible to the program at a given time. The transition relation is then given by the program itself, $s_i \to s_j$ if and only if by executing the instruction/statement pointed to by the program counter in $s_i$ (in the context of any active thread), we can get to the state $s_j$.

Bringing all the powerful constructs of general-purpose languages into model checking, however, has its side effects in the form of new and difficult problems.

First of all, constructing state space on the level of the implementation is overly precise and aggravates the already difficult state space explosion problem even more. Given the large code size of industrial software, numerous methods for mitigating this problem are required if model checking is to be feasible in practice.

Secondly, semantics of common programming languages are not formally defined, instead a textual specification is provided, which is often imprecise in places or overly complex for model checking. This may lead to different interpretations or omission of some of the important features of the language. As a radical solution, model checkers could interpret programs on the level of the machine code. While this is an option, it would result in construction of the state space on an extremely fine-grained level and deteriorate the possibility of applying abstraction techniques efficiently. Instead, multiple research groups found the intermediate representation (IR) of a particular compiler as a viable compromise between the language complexity and the level of abstraction.

Introducing LTL into model checking of software is not free of challenges either. The major problem lies in formulating the validity of atomic

propositions. While it is desirable to define atoms as predicates over variables, there is no clear solution how it should be implemented. Due to the lexical scoping, a common feature of programming languages, variables may become temporary inaccessible, potentially resulting in impossibility to evaluate the set of valid atomic propositions.

Lastly, programs are almost always *open* in the sense that their implementation is not fully-contained, but calls to external libraries and system facilities are made. Unavailability of the external code or its non-applicability for verification means, that an extra effort has to be made to provide replacements and simulate the environment solely for verification purposes.

### 2.1.4 State Space Explosion

As it has been already mentioned, the state space explosion problem is the most prominent difficulty and blocker in making the model checking part of the development cycle.

The source of the blowup is twofold: *data nondeterminism*, which, in purely explicit model checking, causes the state space to branch for every possible input value. This means, that reading even only a few 32 bit integer values makes the traversal of all accessible configurations infeasible. Moreover, concurrent programs suffer from *control-flow nondeterminism* (also referred to as *path explosion*) – every state can have as many successors as there are active threads at the given time, which leads to an exponential growth of the state space in relation to the code size and the number of execution units.

A numerous approaches have been suggested and successfully implemented to minimize this problem. Here we list and briefly describe some of the main techniques:

#### *Symbolic execution*

The key idea is to represent input values symbolically, instead of operating with the concrete data, and expressing the program variables as functions of the symbolic input values [3]. This technique is thus primarily used to cope with the *data nondeterminism*, but in its pure form cannot handle the path explosion problem very well and is not suitable for verification of complex parallel systems. While symbolic execution is a method on its own, separate from model checking and often combined with testing to produce high coverage, multiple model checkers have adopted its principles to symbolically store path conditions and input-dependent variables. This approach

enables to collapse individual states into sets of states, using a suitable set-based data representation, such as *Binary Decision Diagrams (BDD).*

*Bounded model checking*

Instead of using an execution-based approach and exploring all the behaviours separately, a Boolean formula is constructed to represent all possible execution paths violating a given property, with the length restricted by a fixed bound [4]. Any assignment satisfying the formula is a witness for the violation of the property, which can be find using a satisfiability solving program. In practice, the bound is progressively increased and the process repeats for longer and longer traces. As the performance of SMT solvers keeps improving, so does the popularity of BMC raises in the software verification community.

*Reductions*

A class of techniques especially efficient for complex irregular models with a low-level description, therefore a primary pick for (direct) model checking of parallel programs. The most established method is *Partial Order Reduction* (*POR*) [5], which exploits the commutativity of concurrently executed transitions. While POR is used to detect symmetries in execution traces, other techniques, such as the *Heap symmetry* [6, 7], focus on eliminating symmetries induced by data objects and memory configurations.

Completely different category of reductions is based on decreasing the frequency of thread interleavings, aiming to mitigate the path explosion problem. $\tau$-reduction, introduced in [8] and the improved $\tau+$reduction [7], are based on the principle, that it is sufficient to interrupt thread execution and create a state snapshot only before instructions with a *visible* effect, such as `Load` and `Store`, to preserve all the behaviours of the system. The only exceptions are control-flow loops, in which an interrupt must be enforced, otherwise the transition may never be finalized due to an infinite execution path with no observable effects (e.g. active waiting for an event to occur).

*Compressions*

Techniques used to decrease the memory requirements without actually reducing the number of visited states.
For model checkers storing processed states into a hash-table, the *Hash compaction* may be an option [9, 10, 11]. This compression is lossy, however, and

can miss some counter-examples.
Fortunately, some of the lossless compression methods also proved to be effective for model checking, while maintaining the completeness of the verification, like a well-known *Huffman compression* (supported for example by SPIN [12]), *Automata representation* [13] or the *Tree compression*, presented in [14, 15].

*Abstractions*

These methods compute an approximation of a program, which is then easier to verify. The model can be either under-approximated, potentially missing some witnesses of the property violation, or more preferably over-approximated, with a risk of reporting false alarms. A non-valid counterexample, however, can be verified, therefore the procedure is often wrapped in a refinement loop which ends only after no spurious violations are reported or when it runs out of the allocated time. This approach is known as *Counterexample guided abstraction refinement* (*CEGAR*) [16].

*Distributed memory*

An orthogonal approach to all the techniques described above is to harness the combine power and memory resources of network-connected workstations. Most of the traditional model checking algorithms, however, have a sequential character and thus novel designs are needed for this type of environments. SPIN was among the first tools capable to (efficiently) perform distributed state space exploration [17], while DIVINE introduced new sophisticated algorithms for parallel (and distributed) LTL model checking [18]. A more recent example is PREACH[11], a distributed explicit state model checker based on Mur$\varphi$.

## 2.2 Low-Level Virtual Machine

The Low-Level Virtual Machine (LLVM)[1] [19], is a compiler framework providing a modern compilation strategy, initially designed for transparent optimizations and analysis of arbitrary programs. The major contribution of the LLVM project is its well-specified code representation, known as the *LLVM intermediate representation* (*LLVM IR*). It is an assembly-level machine-independent *Static Single Assignment* (*SSA*) based language. LLVM operates in the middle-layer of a typical compiler scheme – in between the compiler

1. `http://www.llvm.org/`

frontend and backend – which makes it independent on both the input language and the target architecture.

Moreover, the LLVM project hosts a number of sub-projects, aiming to provide a wide-range of tools and support libraries to ease the manipulation with LLVM and to increase the scope of applications beyond what was originally intended. An attractive design and well-developed interface for programmers sparked interest in LLVM for model checking purposes as well. Nowadays, multiple model checkers support LLVM IR as the input language for direct verification of programs [1, 20, 21, 22].

## 2.3 Related Work

In this section, we give a brief overview of some of the success-stories of applying software model checking for verification of complex real-world systems at the level of source code. An exhaustive survey, however, is out of the scope of this thesis. Instead, we focus our attention to use cases where the initiative was not just to promote a particular model checking tool, but to actually eliminate intricate errors in the system under verification.

One of the first model checkers with a record in the area of industrial software analysis is VeriSoft [23]. Fully integrated into the development cycle, it was used to analyse several software products developed in Lucent Technologies, a former American telecommunications equipment company. VeriSoft is a *state-less* model checker, meaning that the exploration of the state space is performed without maintaining a queue of visited states. In order to prevent the state-less search from looping forever in cycles, the depth of the search is limited. The most outstanding use case of VeriSoft that has been published is a verification of the call-processing library running on Lucent's CDMA base-stations [24], a complex system involving many concurrent components, implemented by millions lines of code. For the exploration to be feasible, authors opted for a *black-box* approach – only the processes inside the testing environment were controlled, while the non-determinism induced by the processing of the CDMA library was not visible to VeriSoft, meaning that the verification couldn't guarantee completeness.

NASA has had a long standing involvement in the research and development of formal verification methods and tools, primarily carried out by the NASA Ames Research Center[2] and the Jet Propulsion Laboratory[3].

2. `http://www.nasa.gov/centers/ames/home/`
3. `http://www.jpl.nasa.gov/`

One of the first tools to al least partially enable verification at the implementation level was the SPIN model checker [12], developed at JPL. As of the version 4.0, SPIN is able to interpret C code fragments embedded into PROMELA models. Most of the published SPIN use cases, however, did include the modelling step. The most prominent story is a verification of a multi-threaded plan execution programming language used for the Deep-Space 1 mission, which helped to reveal several safety-critical errors [25]. Later at JPL, a tool named *pancam* was created, a virtual machine for executing programs in LLVM IR, used as a frontend for SPIN to enable direct verification of multi-threaded C programs [22].

Meanwhile in the NASA Ames Research Center, Java PathFinder was developed, an explicit-state mode checker used for analysis of Java programs on the bytecode level [26]. JPF has been used on various NASA projects, including the Deep-Space 1 fault protection, Shuttle ground control software, Mars Rover control as well as on products from companies such as Honeywell and Fujitsu [26, 27]. Another project from the same laboratory has led to the development of MCP – an explicit model checker providing capabilities to verify C/C++ code, built on the top of the LLVM framework [20]. It addresses the ever-growing complexity of flight systems, which are mostly written in C or C++. A unique feature of MCP is that is does not assume any particular threading model (such as POSIX threads), instead a generic threading subsystem emulator is provided onto which a user-supplied threading model can be attached. For example, this scheme enabled precise verification of the ARINC-653 flight code with API slightly different from POSIX threads [28].

Another large application area for software verification methods is the Linux kernel development. The Linux kernel is currently one of the most important software systems in our society. From embedded systems up to supercomputers, IT infrastructures heavily rely on its correctness. Most of the Linux verification projects target device drivers which have been identified as the primary source of critical errors. The major obstacle, however, is that Linux lacks a strict and uniform driver framework, which makes the transformation of a driver code into a unit test a difficult task to automatize [29]. A great deal of effort of these projects is therefore oriented around the development of tools for preprocessing and transforming drivers into verifiable programs, while the actual verification is carried out by a particular model checker.

As an example, the Linux Driver Verification Project[4] [30] is an initiative

4. `http://linuxtesting.org/project/ldv`

to define potential hazards that may occur in Linux device drivers and to implement special-purpose verification tools for their automatic detection. All the methods that have been developed use either CEGAR-based BLAST model checker [31] or CPAchecker [32], a tool for configurable software verification, as their backend. In April 2014 it was reported that more than 150 patches had been submitted as a result of the verification effort[5].

Other notable verification projects targeting Linux device drivers are Avinux [29] and DDVERIFY [33], both using bounded model checker CBMC [34] as their primary backend.

Microsoft[6] has also identified the device drivers as the most important source of failures in their operating systems. Consequently, the company has significantly increased the reliability of the Windows OS by integrating the Static Driver Verifier (SDV) into the production cycle. The foundations were developed in the SLAM research project [35]. The authors reported that SDV had helped to reveal many non-trivial bugs and to improve the overall stability of the OS.

For other companies actively using formal verification for correctness assurance of their products, look for Intel[7] [36], Airbus[8] [37] or Honeywell[9] [38].

Lastly, we would like to mention the *Competition on Software Verification* (*SV-COMP*)[10] [39], a solid effort to provide a systematic comparative evaluation of the performance and efficiency of the state-of-the-art in software verification. The benchmark repository of SV-COMP is a large collection of verification tasks which, albeit being relatively simple in complexity, cover the current scope of abilities required from software verification tools. The primary goal of the competition is to accelerate the transfer of new verification technology to industrial practice.

5. `http://linuxtesting.org/results/ldv`
6. `http://www.microsoft.com`
7. `http://www.intel.com/`
8. `http://www.airbus.com/`
9. `http://www.honeywell.com`
10. `http://sv-comp.sosy-lab.org`

# 3 Model Checking C/C++ Programs with DIVINE

## 3.1 Introduction

DIVINE[1] [1] is an explicit-state LTL model checker developed in the Parallel and Distributed Systems Laboratory (ParaDiSe)[2]. The original initiative behind DIVINE was to exploit parallelism in both shared memory and distributed memory environments in order to address the state space explosion problem and the high requirements of model checking in general [18]. In recent years, however, the primary focus has been on language support, resulting in modernization of the DVE interpreter [40], support for LTL model checking and deadlock detection for real-time systems designed in UPPAALL [41], and what is the major highlight – the ability to model-check LLVM bitcode [8, 7], which in turn enables direct verification of C/C++ programs using an LLVM-enabled compiler. This is an important milestone for DiVinE. The requirement to represent systems in DIVINE's own specifically designed language has always been the major turn-off whenever the tool was presented to academic community or industrial partners.

Apart from sophisticated parallel algorithms and multiple language interpreters, DIVINE implements a number of techniques to minimize the state space explosion problem, such as the Partial-Order Reduction [42], Hash compaction [9], Tree compression [15] and the LLVM interpreter specific $\tau$+reduction, `store` visibility and Heap symmetry [7]. Furthermore, a research to introduce semi-symbolic model checking methods is being pursued with the objective to handle the data non-determinism efficiently [43, 44].

In the rest of the chapter, we will focus our attention on the LLVM interpreter from the user perspective and provide a guideline to model-checking C/C++ programs with DIVINE. The remainder of the thesis then describes a real-world use case of this application – our attempt to verify the Name service cache daemon (`nscd`).

## 3.2 From Implementation to Correctness Evaluation

While the support for model checking LLVM bitcode permits to skip the modelling step, in practice the process of verification still requires a great deal of human effort and guidance.

1. `http://divine.fi.muni.cz/`
2. `http://paradise.fi.muni.cz/`

First of all, the verified program must have all the symbols defined for DIVINE to be able to fully construct the state space (with the exception of DIVINE *traps*, which are listed and explained in section 3.6). We refer to such programs as *closed*. As we discuss in section 3.7, verifiable implementations of some of the system libraries are already shipped with DIVINE. The rest of the symbols must be defined by the user himself. A pure computational fragments of external libraries can be supplied without modifications, provided that they are actually available and not overly complicated. On the other hand, system calls and I/O facilities must be simulated.

Additionally, certain operations produce a high-level of non-determinism, such as the random number generator. In theory, this can be interpreted by simply branching the state space for each possible outcome, but in practice it is necessary to provide a constrained replacement or even assume a fixed return value for model checking with explicit-state representation to be feasible. The same rule applies to user-provided substitutions of input operations. For model checking task to fit within the available memory, it is thus crucial to inspect the program before verification and to localize and simplify all the sources of high uncertainty.

Lastly, the counter-example analysis may also be a gruelling task to do manually. Even with all the reductions enabled, an error trace of a medium-sized program can consist of thousands of transitions. As each state is displayed as a dump of all accessible memory locations, the complete textual representation of a counter-example can easily exceed 100 MiB. The problem of counterexample explanation has already been explored for DVE in the past [45]. At the moment, DIVINE does not implement any method for improved error localization. In a close future, however, it is planned to provide a GUI-enhanced tool with debugger-like features. It would enable user to follow the error trace more conveniently and with better focus on important steps. Moreover, unlike the traditional debugger, the exploration could also be performed in the reverse direction of the execution.

## 3.3 Typical Workflow

To summarize the previous section, a typical approach to model-checking C/C++ programs with DIVINE would consist of the following steps:

1. Overall analysis of the system under verification; identification of all external symbols and verification-wise expensive operations, followed by evaluation of their minimal set of properties/behaviours as required by the system, simply by tracking and inspecting all their

occurrences in the source code.

2. Design and implementation of a closed virtual environment, comprised by a set of simplified replacements for fragments of the original code which are not available or not applicable for direct model-checking. Ideally, the environment should have a minimal impact on the size of the state space.

3. Optionally, complementing the expressive power of assertion statements with definitions of some LTL properties, directly embedded into the test suite.

4. Model checking of the environment in order to eliminate programming errors introduced in the second phase.

5. Verification of the complete system or individual units of the source code using the prepared test suite.

6. Analysis of the verification outcome. It may lead to a further redesigning and simplification of the tests and the environment if DIVINE reports failure for a lack of available memory.

We believe that a more efficient approach, however, is to incorporate verification into the development cycle. This would mean that the first three steps are performed regularly, without much need for a cumbersome retrospective analysis. While at first it may seem a too time-costly addition to a rapid test-driven development – especially the first two steps – in practice this is already a common process in testing and the simplified replacements are known as *test stubs*.

Conversely, the model-checking procedure would have to be performed less frequently than the natively-executed tests for its high computational complexity. Conventional testing is still expected to detect the majority of errors, while model checking could be applied just before important milestones to significantly decrease the chance of missing intricate bugs before the product is released and sold to customers.

## 3.4 LTL Specification

For the purposes of convenience, DIVINE offers means to embed definitions of LTL properties directly into the source code, instead of keeping them in separate files. But as neither C nor C++ provide reasonable and optimization-free constructs for expressing LTL, properties are stored into

the constant global memory as arrays of characters (with the syntax as defined in subsection 2.1.2), instead of being part of the program data. The variable which references such represented LTL property has a user-defined name prefixed with `__divine_LTL_`, so that it is recognised and parsed by the LLVM backend.

To be precise, LTL properties are expressed using the macro `LTL(name, prop)`, provided by DIVINE and defined as:

```
#define LTL( name, prop ) \
    extern const char * const __divine_LTL_ ## name = #prop
```

Atomic propositions are referenced by integer values wrapped in an enumerated type of the source language, identified as `APs`. A validity of a proposition for a *single* state is signalled by invocation of a DIVINE provided trap `__divine_ap(a)`, with the proposition ID as its argument. Again, a macro is provided to hide this implementation detail:

```
#define AP( a )  __divine_ap( a )
```

To summarize using an example, an excerpt of a program containing definitions of LTL properties is included as Listing 3.1.

```
// Definitions of LTL and AP are automatically included by DiVinE.
LTL(progress, G(wait1 -> F(cs1)) && G(wait2 -> F(cs2)));
LTL(exclusion, G(!(cs1 && cs2)));

// The identifier must be "APs".
enum APs { wait1, cs1, wait2, cs2 };
...

void implement_critical_section( int thread_id ) {
  AP( thread_id == 1 ? wait1 : wait2 );

  // This operation may be blocking, but should finish eventually.
  enter_critical_section();

  // This is supposed to be executed exclusively.
  critical_section( thread_id );
  AP( thread_id == 1 ? cs1 : cs2 );

  // Allow the other thread to enter the critical section now.
  leave_critical_section();
}
...
```

Listing 3.1: An excerpt of a program implementing critical section for two threads, extended with definitions of some LTL properties for verification.

## 3.5 Safety Properties

Apart from LTL properties, DIVINE provides a wide variety of safety conditions to check, for which a simple reachability analysis is sufficient to determine the validity. Table 3.1 lists and describes all the available safety properties.

| *Property* | *Description* |
|---|---|
| `assert` | Verify that assumptions made by the programmer and expressed using the macro `assert` are satisfied. |
| `deadlock` | Deadlock freedom in the traditional model checking sense, i.e. satisfied iff all the reachable states have at least one outgoing transition (not very useful with the LLVM backend as discussed in section 3.9). |
| `pointsto` | Detect invalid use of pointers based on the points-to information, which is statically precomputed with one of the pointer analysis algorithms and stored into LLVM metadata. |
| `memory` | Reveal improper manipulation with memory objects. Includes bound checking and detection of invalid dereferences. |
| `arithmetic` | Check for the presence of the *division by zero* error. |
| `leak` | Detect memory leaks. |
| `user` | User or library defined safety problems (defined by the use of the `__divine_problem` buitlin, see section 3.6). |
| `guard` | Safety of compiler-defined guards (e.g. the non-reachability of the `unreachable` instruction). |
| `mutex` | Report violation if the resource allocation graph constructed for Pthread mutexes contains a cycle (a more useful version of `deadlock` for Pthread-based parallel programs). |
| `safety` | All the above except `deadlock` and `pointsto`. |

Table 3.1: Safety conditions supported by DIVINE.

## 3.6 Built-in Functions

Internally, the LLVM backend implements a virtual machine executing instructions of the verified program to gradually explore the complete configuration graph.

Certain programming concepts, however, cannot be facilitated solely by the language features of a pure C/C++, and an access to the system API is required. For example, this includes thread management, atomicity control and memory management. Additionally, LTL property specification requires special support as well. Therefore, a low-level interface of the machine is exposed through a set of DIVINE defined *builtins*, making the interaction between the user-supplied code and the execution engine feasible.

Some of these *builtins* are rather one-purpose, defined to implement a specific feature, now supplied by DIVINE. Here, we instead focus on the reusable subset of builtins, which users can exploit to implement virtual environments for their programs. Table 3.2 provides a summary with a short description for each of them.

## 3.7 Library Substitutions

To make the production of verification-ready programs easier, DiVinE supplies replacements or partial replacements of system libraries, which are statically linked into the program during the DIVINE-driven compilation of the source code into the LLVM bitcode. Collectively we call them the "*user-space*" to make a clear distinction between the interpreted code (model description) and the executed code guiding the process of interpretation (the LLVM interpreter, a.k.a. the "*system-space*").

Specifically, a slightly modified copy of the Public Domain C Library (PDCLib)[3] is shipped with DIVINE to provide almost complete implementation of the C standard library. The library uses small, but well-defined interface to the low-level file system API, making it easily attachable to user-defined virtual environments.

For C++ programs, DIVINE ships with *libc++abi*[4] to provide the run-time support and with *libc++*[5], implementing the C++ standard library. The run-time library had to be altered in places to enable support for model-checking C++ code with exception handling [46].

3. `http://pdclib.e43.eu/`
4. `http://libcxxabi.llvm.org/`
5. `http://libcxx.llvm.org/`

Moreover, a substantial subset of the POSIX threading API was implemented specifically for verification, including thread management (creation, joining, detaching and exiting), mutual exclusion (normal and recursive mutexes), thread-local storage, barriers, once-only execution and condition variables.

## 3.8 Command Line Interface

In this section, we briefly describe the usage of DIVINE for model checking C/C++ programs. A more detailed summary, with all the commands and available options, can be find in the user manual for DIVINE or displayed with "`divine --help`" and "`divine <cmd> --help`".

Firstly, DIVINE doesn't really support C or C++ as the model description language. Instead, C/C++ programs must be first compiled into the LLVM bitcode, which DIVINE can then interpret. Additionally, the obtained bitcode file must be statically linked with the verifiable variants of system libraries shipped with DIVINE.

To simplify this process for users, DIVINE provides a command named `compile`, which is a frontend to an actual LLVM-enabled compiler (such as *clang*), performing all these steps transparently.
For C/C++ programs, the usage is as follows:

```
divine compile --llvm [--cflags=<val>] [--precompiled=<val>]
                      [--output-file=<val>]  <input-file>
```

This will take longer that you would expect as all the system libraries must be compiled as well. To speed-up the process, you can use "`divine compile --libraries-only`" (without any input file) to obtain the archives of prebuild system libraries. The file path to these archives is then specified using the `--precompiled` option.
The output file will have an extension "`bc`" (unless defined otherwise) and for DIVINE it basically represents a model description, ready for verification.

To get a list of all available properties (and to check that all the symbols are actually defined), run:

```
divine info <bitcode-file>
```

To explore and measure the state space without actually verifying any property, invoke:

```
divine metrics <bitcode-file>
```

However, bear in mind that this operation is asymptotically as complex as the verification itself.

For further state space analysis, a debugger-like step-by-step exploration can be performed using `divine simulate` and a partial visualization of the configuration graph can be obtained with `divine draw`.

The actual model checking is executed by `divine verify`:

```
divine verify  [--property=<val>] [--d] <bitcode-file>
```

Where `-d` is a short-name alias for `--display-counterexample`, an option to print the counterexample if one was found. Once finished, the command reports whether or not the property is satisfied and potentially outputs an error trace.

## 3.9 Limitations

As it has been already pointed out, the approach used by DIVINE strictly requires full reproducibility of every execution step, meaning that no real I/O operations are allowed. Process can only access its own memory and all the interactions with the environment must be emulated. While some of the system facilities can be relatively easily simulated to almost full API compatibility, such as a file system, others may need to be constrained and simplified for model checking to be feasible. In order to maintain credibility of the verification, any constraint imposed during simplification must not interfere with the requirements of the system under verification. The implication is that DIVINE cannot supply substitutions for all the aspects of an execution environment and it is thus necessary for users to analyse the system before verification and customize the simulated environment accordingly.

Currently, the LLVM backend doesn't support fairness, meaning that LTL properties expressing progress are very likely to fail. Actually, for any Pthread-based parallel program it is almost guaranteed, as not even thread creation (`pthread_create`) counts as a wait-free operation. Unfortunately, sleeping of threads isn't supported either, hence the busy-waiting is an unavoidable construct for thread synchronization, producing non-fair execution runs due to the presence of self-loops in the configuration graph.

The expressive power of `__divine_ap` is yet another limitation for usability of LTL in model checking. Atomic proposition values can be configured only on a single-state basis. Therefore, it is impossible to make a

proposition valid for a consecutive sequence of states, meaning that properties of the form $\mathcal{G}a$ cannot be satisfied either.

In the traditional model checking, a deadlock is defined as a system state without any outgoing transitions. For C/C++ programs, however, a different definition and an approach for detection are required. First of all, programs, especially non-reactive ones, tend to terminate eventually, producing a finite-length execution run. Normally, we wouldn't want a termination to be treated as a deadlock. Instead, a real deadlock (even more so livelock) will appear as a loop in the state space with certain properties. Currently, DIVINE can detect both self-deadlock and circual deadlock, but only for Pthread mutexes.

For certain classes of programs, it is also desirable to consider relaxed memory model in order to detect errors caused by delayed memory operations. At the current version, however, DIVINE implements a sequentially consistent memory model disallowing any instruction re-ordering. An improvement in this area is planned for the future.

Lastly, a purely technical limitation is an absence of multiprocessing support. The state vector of the execution engine is designed to represent exactly one process with an unlimited number of threads. Consequently, systems composed of multiple processes have to be simplified by mapping processes to threads and resolving any interference that may occur as a result.

| *Builtin* | *Description* |
|---|---|
| `__divine_new_thread` | Start a new thread, with a supplied function as an entry point and a pointer-sized argument. |
| `__divine_get_tid` | Obtain a unique identifier of the calling thread. |
| `__divine_interrupt_mask` | Mask interrupts, i.e. disallow other threads to interrupt execution of the calling thread. The masking is bound to stack frames, meaning that a stack unwind leading to an originally unmasked function will automatically cause an unmask. |
| `__divine_interrupt_unmask` | Cancel the effect of any previous interrupt mask. |
| `__divine_interrupt` | Manually invoke interrupt. Useful for non-interruptible loops to prevent from introducing a transition with an infinite execution path. |
| `__divine_choice` | Implements non-deterministic choice. The state space is branched for every value from a given range. |
| `__divine_assert` | Raise assertion violation if the argument evaluates to zero. |
| `__divine_ap` | Signal that a given atomic proposition is valid in the current state. |
| `__divine_malloc` | Request fresh memory from the heap. This operation never fails, while the `malloc` family of functions is implemented to simulate potential failure using a non-deterministic choice. |
| `__divine_free` | An alias for `free`, i.e. invalidate the memory block pointed to be the argument. |
| `__divine_memcpy` | Copy a block of memory without destroying pointer maps, a data structure used by the `store` visibility reduction [7] for tracking pointers. |
| `__divine_problem` | Raise a user-defined violation of a given type and with a description of the problem. |

Table 3.2: A *re-usable* subset of DIVINE defined builtins.

# 4 Name Service Cache Daemon

The GNU Name Service Cache Daemon (`nscd`)[1] is a process, typically running in the background as a daemon, providing cache capabilities for the most common name service requests, including accesses to the `passwd`, `group`, `hosts`, `services` and `netgroup` databases through standard libc interfaces, such as `getpwnam`, `getpwuid`, `getgrnam`, `getgrgid`, etc.

When a new (name service lookup) request is received, first the associated cache is searched to see if a response to this request is already known. If that is the case, `nscd` answers immediately, without accessing the relevant service, thus saving some computer resources and decreasing the response latency. Otherwise, the service is accessed as normally and the response is forwarded to the client as well as stored into the cache with a bonding to the original request, so that future requests for the same data can be served faster.

GNU `nscd` is developed and maintained inside the glibc[2] repository, under the sub-directory `glibc-X.XX/nscd/`. It is written in the C programming language and the implementation consists of 34 source files and 5 headers with cca. 13500 lines of code in total (including comments and blank lines). For this thesis, we extracted `nscd` source code from the GNU C library, version 2.19 stable. As of this writing, a newer version 2.20 is available, but has no substantial changes or bug fixes related to `nscd` or other tightly-cooperating sub-systems.

## 4.1 Background

`nscd` is an optional component of an OS facility called Name Service Switch (NSS), a clean and extensible solution for accessing directory information – databases of people/groups/hosts/etc. – through a designated set of POSIX API functions. Traditionally, this type of information was obtained from files (e.g., `/etc/host`), using a simple, but hard-coded method provided by the C library.

With the rapid development of network technologies and continuous growth of shared computing infrastructures, file-based solutions stopped to scale and had to be replaced by directory services for most of the applications. A wide variety of name resolution services enabled different data to be stored at different places. A need for a common access mechanism,

1. `http://linux.die.net/man/8/nscd`
2. `http://www.gnu.org/software/libc/`

allowing run-time configuration while being extensible for future services, has led to the development of Name Service Switch.

Originally designed and implemented by Sun Microsystems for their Solaris operating system, but subsequently ported to many other operating systems, including FreeBSD, NetBSD, GNU/Linux, HP-UX, IRIX and AIX. As we discuss more closely in the next section, `nscd` has been inserted in between the application programming interface and NSS to provide caching for slow services like LDAP, NIS or NIS+.

In this thesis we focus solely on the GNU `nscd`, which, albeit fitting into the scheme as designed by Sun, has no common code with the Sun's version or any other. Therefore, the results we obtained from the verification are relevant only for the GNU-based operating systems.

## 4.2 The NSS Scheme

The basic idea behind NSS is to put the implementation of different services offered to access the databases in separate modules, loaded at the run-time using the dynamic linking loader. This approach offers several advantages over the static legacy solution:

1. Support for new services can be easily provided without changing the C library.
2. Modules can be loaded only as needed and based on a run-time configuration.
3. This scheme requires strict interface between the standard library and services, which, in a long run, can only be beneficial in terms of code maintainability.
4. The C library image is smaller and receives less changes from external contributors.
5. The modules can be updated separately.

To facilitate this idea, a simple naming scheme for modules and their external symbols was introduced. For service `serv`, the module (dynamic library) should be available as `libnss_serv.so`, i.e. prefixed with "`libnss_`" and with the extension as used by the system for dynamically loaded libraries. Function `fct` will be then searched for as `_nss_serv_fct` (NSS prefix + service name + function name) in the context of symbols defined with the external linkage.

The set of supported services and the order in which they should be accessed is specified individually for each database in the configuration file `/etc/nsswitch.conf`[3]. The file is plain ASCII text, consisting of columns separated by white-space characters. The first column specifies the database name, immediately followed by a colon. The remaining columns then describe the order of services to query and, optionally, also a limited set of actions to perform based on the lookup results.

For network based services, every query would translate to a TCP connection with handshake overhead, possibly over SSL introducing a further performance penalty, resulting in a very high latency for production use. Therefore, the caching daemon `nscd` is typically used in between API and NSS to speed-up consecutive requests. It is accessed via a UNIX socket, and as illustrated by the Figure 4.1, loads the nss modules itself in order to act as a hit-and-miss cache. Moreover, it can also be used as a client to contact another `nscd` running as a daemon, and obtain the current cache statistics or invalidate the cache content or even shut down the running daemon.

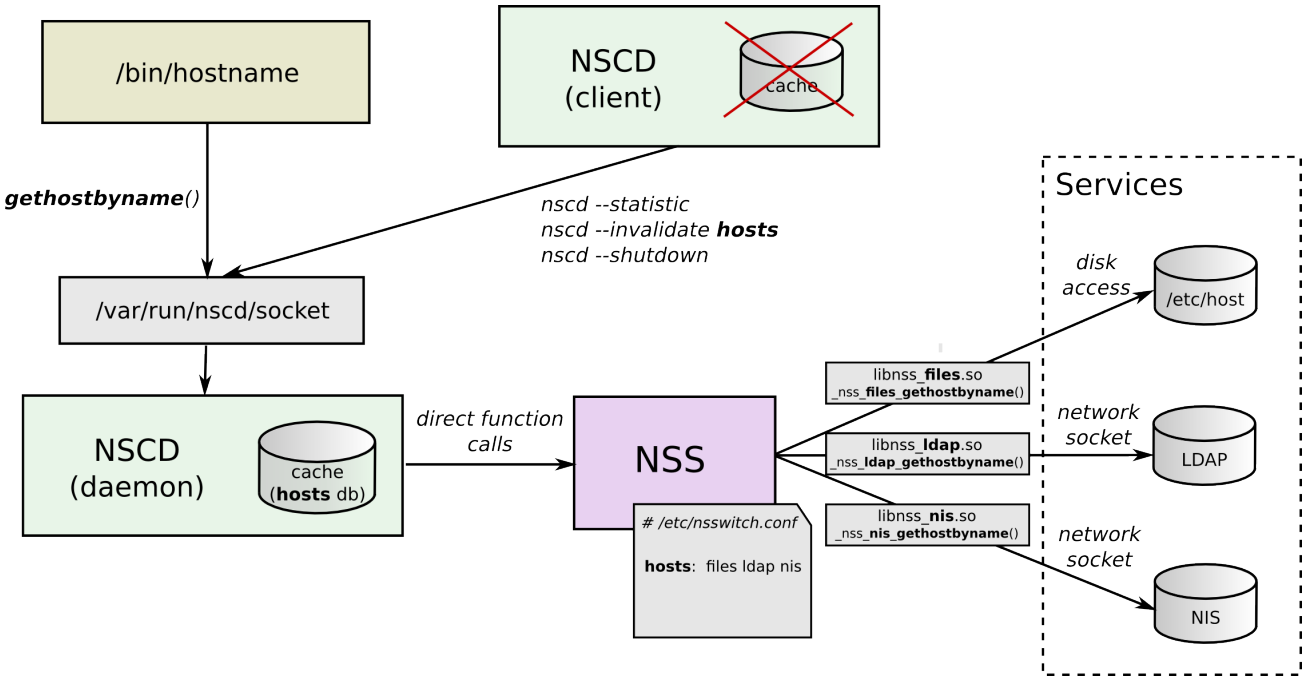


Figure 4.1: The visualization of the `gethostbyname()` control flow within the NSS facility.

3. `http://linux.die.net/man/5/nsswitch.conf`

## 4.3 Configuration

The GNU Name Service Cache Daemon can be configured via the ASCII-only text file */etc/nscd.conf*[4], read at the program startup.

Each line specifies either a global attribute and a value, or a database-specific attribute, database name, and a value. Fields are separated by white-space characters. Comments start with the hash character, #, and extend to the end of the physical line. Valid databases are `passwd`, `group`, `hosts`, `services` and `netgroup`.

An example of the `nscd` configuration file, with a short description for every attribute, is included as Listing 4.1.

## 4.4 Cache

Internally, `nscd` maintains a separate cache for every (enabled) database. The cache is represented by a pair of C structures – `database_dyn` and `database_pers_head`, connected through a pointer. The former stores the so-called *dynamic* attributes, data relevant only for the current execution, such as the file descriptors associated with the database file (see later), locks for concurrency control and a subset of the user configuration. The *persistent* cache data are wrapped by `database_pers_head`. If enabled, the content of this structure is mirrored onto a file (one for every database) using the POSIX facility for memory mapped files (*sys/mman.h*). This way, the cache content is not only preserved over server restarts, but can also be accessed externally by privileged processes.

Each database file is opened with two file descriptors, one in the full RW mode used by `nscd` to propagate changes from the cache into the file and vice-versa, while the other descriptor allows only read operations and can be acquired over a UNIX socket and used to search the cache without the assistance of `nscd`. For an external process to be able to access the cache directly, special privileges are required, including `SCM_RIGHTS`[5] for transporting open file descriptors.

In order to associate queries with responses and store them efficiently while minimizing the lookup cost, `nscd` implements the traditional hash table algorithm with buckets (*Separate chaining with linked lists*). The key for each entry is computed based on the request arguments without violating

4. `http://linux.die.net/man/5/nscd.conf`
5. `http://linux.die.net/man/3/cmsg`

```
# /etc/nscd.conf
#
#   <global-option>         <value>
#   <db-specific-option>    <db>       <value>

# Global options ------------------------------------------------------

logfile         /var/log/nscd.log  # Destination for debug messages.
threads         4          # The (initial) number of workers.
max-threads     32         # The maximum number of workers.
server-user     nobody     # Run nscd as this user.
stat-user       somebody   # This user is allowed to request stats.
debug-level     0          # Desired debug level. The default is 0.

# Enabling paranoia mode causes nscd to restart itself
# periodically. The default is no.
paranoia            no

# Set the restart interval in seconds if the paranoia mode
# is enabled. The default is 3600.
restart-interval    3600

# How many times a cache record is automatically reloaded without
# actually being used before it is removed.
reload-count        5

# DB attributes (here only for the "hosts" database) ------------------

enable-cache            hosts   yes  # Enable cache for this DB.
positive-time-to-live   hosts   3600 # TTL for positive responses.
negative-time-to-live   hosts   20   # TTL for negative responses.
suggested-size          hosts   211  # The number of hash buckets.

# Check for the changes in the associated files of the file-based
# services for this database (e.g. /etc/hosts for the "hosts" DB).
check-files             hosts   yes

# Keep the content of the cache over server restarts,
# by storing it into a memory-mapped file.
persistent              hosts   yes

# Allow clients to directly access and search the cache through
# the associated file descriptor, passed over a Unix domain socket
shared                  hosts   yes

# The maximum allowable size, in bytes, of the database file.
max-db-size             hosts   33554432
```

Listing 4.1: An example Name Service Cache config file with explanations.

the uniqueness property, while the hash of an entry is obtained using a further (lossy) transformation of the key, with the size also reduced by the modulo operation to fit into the range of available buckets.

As illustrated by Figure 4.2, the hash table starts with an array of buckets, each referencing a linked-list of associated hash entries (instances of `hash_entry`). Hash entry is merely a reference to `datahead`, a variable-sized structure storing the key and the response data alongside some additional fields, such as the Time to live (TTL) of the entry. The separation of hash entries from data entries is simply because the same data can often be obtained through different methods, such as `gethostbyname()` and `gethostbyaddr()` for retrieving the `hostent` structure of a given host, therefore the equivalent response values may fall into different buckets.

### 4.4.1 Memory Management

A closer look at the implementation reveals, that `database_pers_head` is in fact a header for a variable-length object with a zero-length array attribute at the end, referencing the start of the actual hash table storage. To facilitate the mapping between the memory and the database file, the storage for persistent data must be a continuous memory block and all the allocations for hash entries must necessarily come from this region. Furthermore, all the references inside the hash table must be represented as offsets from the base address, since the standard C pointers would become invalid after the server restart or when accessed from another process. Therefore, `nscd` implements its own memory management operating on the top of a pre-`malloc`-ed memory area (to a maximum allowable size), obtained during the initialization phase individually for every enabled database.

The allocation mechanism works simply by returning the offset of the first byte past the last used byte. This operation has a constant time complexity as `nscd` keeps the memory usage information updated inside the header area of `database_pers_head`. While trivial in design, this approach requires regular garbage collection, otherwise `nscd` would quickly run out of the available memory for cache data. Alternatively, a new memory could be made available by expanding the existing mapping using `mremap()`, but with this solution the size of the cache storage would grow infinitely.

Therefore, alongside the worker threads processing client requests, every database runs a special so-called *prune* thread, periodically walking through the hash table and removing (or reloading) all entries which lifetime ended. The clean-up procedure consists of three steps. First, all the hash entries are visited and the TTL field is compared with the current time

to determine whether the entry should be removed. In this phase, obsolete entries are only removed from the linked-list so that they are no longer referenced. The next step is the *mark* phase of the *Mark-and-Sweep* algorithm. A bit-mask representing the set of bytes used by the accessible subset of all objects is constructed by traversing the hash table once again. Lastly, the garbage collection is finished by moving the non-free memory areas over the unused regions towards the base address so that the internal fragmentation is completely eliminated.

## 4.5 Concurrency

In order to decrease the response latency as much as possible, `nscd` also leverages parallelism by processing client queries in separate threads – so-called *workers*. The main thread is primarily responsible to carry out the initialization phase, i.e. loads and parses the configuration file, pre-allocates memory for all the caches and starts other threads. Afterwards, it only waits for new connections, while file descriptors of accepted sockets passes directly to workers for processing.

Workers are not sorted by databases or any other criteria, everyone of them has full capabilities to handle any request. Most of the time these threads are idle, waiting on a Pthread conditional variable, and only when a receipt of a request is signalled, one of the workers is woken up to handle the request. Normally, there is at least as many workers as databases, but in the debug mode the number is allowed to be lower. But if a situation occurs that all the processing threads are busy and a request is received, a new worker is fired instead of delaying the query.

Moreover, as discussed in subsection 4.4.1, one prune thread is running for every enabled database, performing garbage collection for the associated cache. Cache pruning cannot be disabled but the time interval between subsequent passes is configurable via the configuration file. In addition, the timing is cleverly de-synchronized between databases so that prune threads do not wake all at once.

Concurrency control is facilitated using Pthread mutexes and RW locks. Every cache has associated one RW lock and 2 mutexes – one to avoid parallel execution of the garbage collection with the cache invalidation, and the other to serialize memory allocations performed over the same hash table storage. The purpose of the RW lock is to avoid collisions between workers and prune threads. For query processing the read lock is sufficient, whereas cache pruning requires full RW privileges.

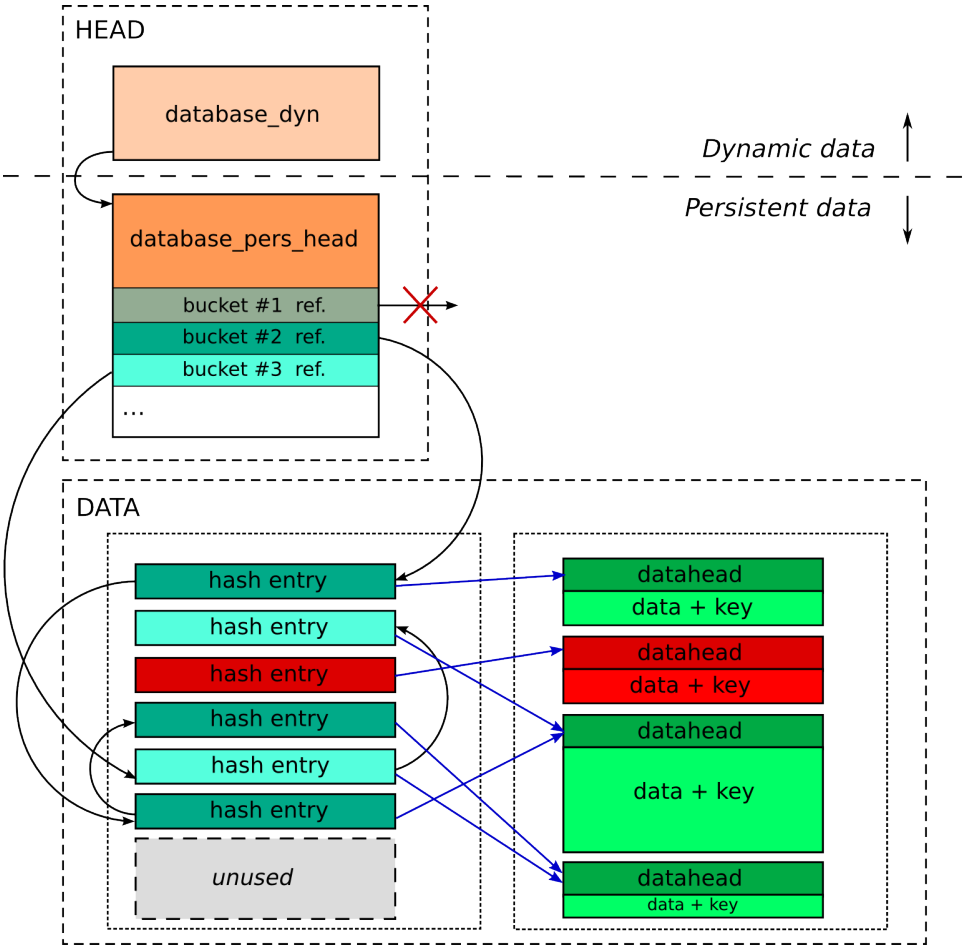


Figure 4.2: The `nscd` cache for one database, allocated using only two real `malloc`s – one for the dynamic data and one for the persistent data. This illustration represents the cache in the state after the first phase of pruning, with the non-referenced entries highlighted using a red background color. Also notice that the same data can be referenced by multiple hash entries, potentially residing in different buckets, as discussed in section 4.4. The distribution of hash and data entries is somewhat simplified here, because in reality these objects tend to be mixed among each other within the `DATA` region.

# 5 Model Checking the GNU NSCD

## 5.1 Motivation

For our real-world use case of model checking we chose The GNU Name Service Cache Daemon for a number of reasons.

First, in terms of size and complexity, `nscd` falls into the medium category. While certainly much more complex than a typical benchmark for software verification, it is not massive and thus should be a feasible target for the state-of-the-art explicit model checkers if this approach to quality assurance is ever to become widely applied in practise. Also for DIVINE itself this is a significant step from simple test programs to a proper performance evaluation.

Second, `nscd` exploits multi-threading a lot but introduces only a very little of data non-determinism as we will see later, making it a perfect fit for the *explicit-state* model checking.

In addition, the in-memory process-shared cache implemented by `nscd` initially appeared as an interesting target for the memory-safety verification. But as a closer system analysis revealed, the cache storage resides only in a single continuous block of preallocated memory with a custom memory management working on the top of it (non-transparent for model checker), meaning that (real) invalid dereferences and memory leaks are less likely to be present.

Last but not least, `nscd` is not just another benchmark tailor-made to showcase the strengths of formal verification, but a real-world software used in all GNU-based operating systems, making it a legitimate target for a critical evaluation of the feasibility of model checking for industrial use.

## 5.2 System Analysis

As explained in section 3.2, not even direct model checking can work without any human assistance and a great deal of manual effort is needed to invest into preparation of programs for verification. In case of DIVINE, systems need to be thoroughly analysed and all non-reproducible or highly data non-deterministic actions need to be localized and simulated or simplified. `nscd` is no exception to this, as it extensively interacts with the environment through various communication facilities.

### 5.2.1 Decomposition

Following our description of the NSS Scheme from section 4.2, let us first describe our approach to the system decomposition and environment emulation, which is visually summarized in Figure 5.1. Since we weren't able to find any existing `nscd` unit tests, we decided to perform the verification of the system as a whole and from the user point of view, exactly like *Acceptance testing* does. For a credible correctness evaluation at the level of individual units, we would desperately need an access to the original intentions of developers, otherwise the verification outcome would merely be a justification of our (mis)understanding of the system's inner workings.

The major achievement is that we have managed to perform the model checking while keeping the `nscd` source code almost untouched. Only the SELinux access controls were disabled and omitted from the verification (by un-defining the macro `HAVE_SELINUX`). For this thesis we have simply decided to target the concurrency bugs and memory safety violations, while an analysis of security vulnerabilities is left for the future work. In addition, we were also able to *copy-and-paste* the NSS modules and some other purely-computational dependencies from glibc.

In our approach, every other component participating in the NSS scheme is viewed to the system as an interaction with the environment which had to be simulated. External storage (a filesystem), used by `nscd` to store the cache data persistently as well as to log debug messages and load the user configuration, was emulated in-memory to a full API compatibility as a part of our *Closed Virtual File System* (*CVFS*), described in chapter 6. CVFS also supports UNIX domain sockets with `SCM_RIGHTS` (if enabled), for passing open file descriptors to other processes using ancillary data. Therefore, the communication between clients and `nscd` is covered as well.

Since we have decided to copy the NSS modules as they are, it was also necessary to implement the programming interface to dynamic linking loader. This couldn't be simulated to a full generality without an additional support from the LLVM backend, therefore we designed only an imitation of dynamic linking that requires further customization individually for every test. Our environment provides two internal functions – `_dl_newlib(name)` and `_dl_newsym(lib, sym_name, sym_pointer)`, which can be used by the programmer to define the list of available dynamic libraries and their symbols, while pointers to these symbols (as returned by `dlsym()`) have to point to objects already present in the verified bitcode file. For example, if we wanted to simulate the LDAP service for retrieving the `hostent` structure of a given host, we would implement test

stubs for `_nis_ldap_gethostbyname()` and `_nis_ldap_gethostbyaddr()`, and then register them inside a (faked) `libnss_ldap.so` library (before any client requests are issued).

Finally, the end-points of this scheme are clients and services themselves. In our test suite, both of these components together constitute individual (acceptance) tests, i.e. every test implements the behaviour of a client and a service (or possibly more), and then checks using assertion statements if the responses obtained from `nscd` match the values returned by the simulated service(s). Since the LLVM backend doesn't provide multiprocessing support, clients run only in separate threads of the same process as `nscd`.

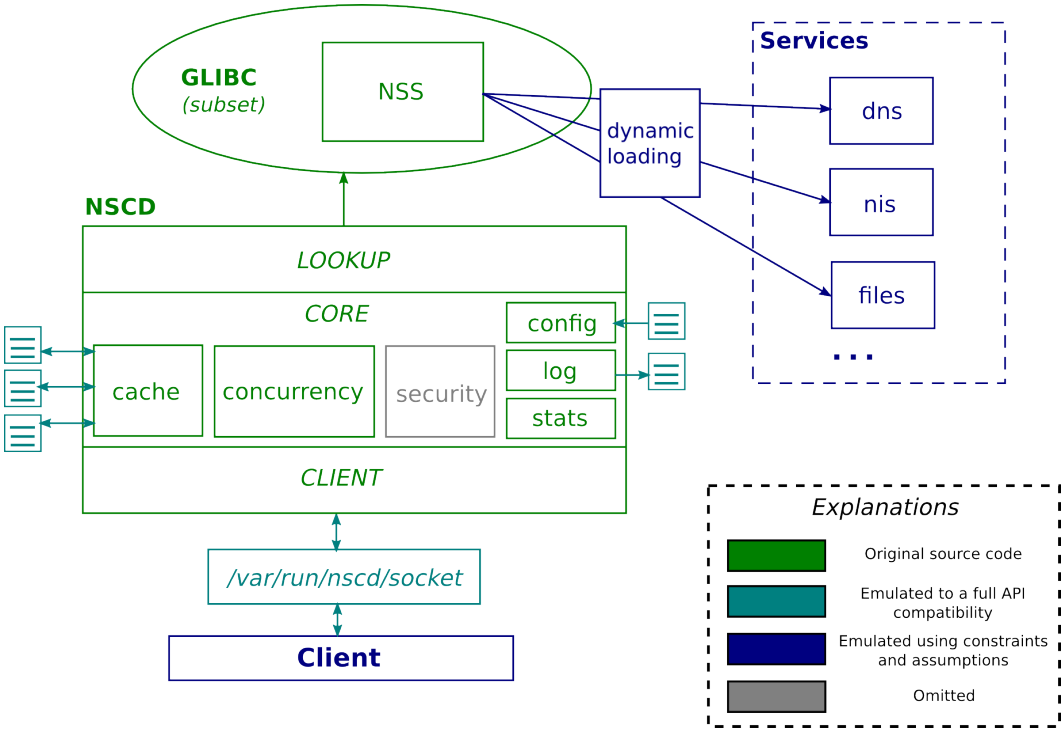


Figure 5.1: Our approach to the decomposition of the NSS Scheme for model checking purposes.

### 5.2.2 Complexity

The outcome of the work described above was a fully self-contained (*closed*) implementation of the GNU NSS facility (one instance per acceptance test),

with most of the source code being original. While this is theoretically sufficient to hand the task over to model checker, in reality it is very unlikely that we would have obtained any results without further simplification due to the notorious state space explosion problem.

*Data non-determinism*

As explained in subsection 2.1.4, the two major sources of the blowup are the *data non-determinism* and the *path explosion*. In terms of inputs, `nscd` reads the configuration file, argument values of client requests and data returned from services. Additionally, `nscd` uses time to schedule certain events, such as the cache pruning, and assigns a TTL value to every new hash entry based on the current time obtained from the system clock.

In our test suite, we opted for a fixed-content configuration file. However, each option can be easily adjusted to produce new instances of a single test, using a set of macros defined near the top of the source file. The behaviours of clients and services are also deterministic, with fixed argument and return values, and we do not even intend to make them (easily) customizable. Instead, clients typically issue multiple queries with different values to gradually test all the available interfaces of a specific database.

For the representation of time we have implemented multiple types of clocks, accessible through the interface `time()` from the standard C library. Most of these clocks are heavily constrained as it would be infeasible to consider every possible timing of events. For all of these clocks, user must first define a maximum time interval that can elapse in between two successive calls of the function `time()` (from any thread), as well as the minimum interval that the clocks can measure (or as we call it – the *clock tick*). The most non-deterministic clock then works by randomly choosing the number of ticks that have elapsed since the last call not exceeding the maximum interval. On the other hand, the most constrained (and fully deterministic) clock simply assumes one clock tick with each measurement. For our experiments, however, we ended up using the so-called *manual* clock. When used, the time doesn't advance unless explicitly requested. Using the internal function `_env_clock_tick()`, one can manually trigger one clock tick as needed. Since the LLVM backend doesn't support model checking with time constrains, this approach was needed to eliminate all unrealistic timings that would otherwise produce highly improbable time-outs and the related safety violations.

*Control-flow non-determinism*

During the system analysis, we identified the thread interleaving as the major source of non-determinism. Without the prune thread, however, the situation would be significantly different. If we assume at most one client interacting with `nscd` at any time and no garbage collection running in the background, the processing of client queries is basically serialized.

When a request is obtained, first the main thread is woken up to establish a new socket-based connection, while workers still remain inactive. Only when the main thread passes a newly accepted socket to a global list of ready connections, one of the workers is activated and starts the request-handling procedure. The main thread no longer participate in the processing of this request and immediately goes back to the (busy-)waiting state. Meanwhile, the client thread (in a true NSS scheme it would be a separate process) is busy-waiting for a response without performing any actions with external effects, thus only producing self-loops in the configuration graph. The worker thread performs the cache lookup and potentially accesses the relevant service, but only if the response is not already known. Either way, the execution is sequential up to the point when a response is sent back to the client. For the worker, the task may not be finalized yet as a new cache entry has to be added or an existing one reloaded in case the data have changed. This happens after the response is returned so that clients are not unnecessarily delayed. During that time, the client may already do some post-processing of the obtained data, which would thus interleave with the finalization phase of the worker.

If we add the pruning thread into the formula, however, the situation becomes much more complex. Depending on the time representation, the garbage collection can trigger almost at any time and interleave with the processing of current queries. Since the cache clean-up procedure operates over a thread-shared cache storage, almost all the instructions are visible and require a state snapshot. Moreover, the procedure is relatively long, consisting of three phases as described in subsection 4.4.1. Therefore, we decided to make the garbage collection optional and provided a configuration switch which can be used to disable the cache pruning. The results of our experiments proved that this was a wise decision (see chapter 7).

*Memory usage*

Furthermore, the memory consumption of a model checker is not only given by the number of reachable states, but also by the average size of a state

snapshot. As expected, the most space-complex data structure is the cache storage. Since `nscd` shares the cache data through database files with other processes, it has to reside in a single continuous memory block. Moreover, the cache storage must be pre-allocated to a maximum possible size as an implication of the following statement from the Linux Programmer's Manual (mmap(2)): "*The effect of changing the size of the underlying file of a mapping on the pages that correspond to added or removed regions of the file is unspecified*".

By default, `nscd` assumes 32 MiB for every database, but this parameter can be customized through the configuration file. Given that a complete snapshot of every enabled database is present in each state, we believe that the default value would make the model checking infeasible. Fortunately, for our experiments we have never needed to set this parameter to more than 512B.

## 5.3 Build system

It is no wonder, that by extracting `nscd` and NSS from glibc and placing them into our own environment, we have introduced new bugs not present in the native variant of the system. Most of these bugs were common programming errors, while others resulted from API incompatibility between our implementation and glibc, primarily caused by different interpretations of the standard. Moreover, since GNU `nscd` is strictly tight to one specific implementation of the standard C Library, it even assumes implementation-defined behaviours of OS facilities as given by glibc, which oftentimes does not conform to POSIX or is not defined by the standard at all.

For this reason, it was desired to have the capability to execute tests natively before performing the model checking. The majority of non-original bugs were easily reproducible and the standard testing approach turn out to be a more suitable and a less time consuming method for their detection.

In order to facilitate different build modes of a relatively large codebase and make them easily customizable, we have decided to manage the compilation and the execution/verification process using CMake[1], a cross-platform and open-source build system. For the native mode, we provide replacements for DIVINE builtins, implement certain non-pure C concepts differently, hide internally defined symbols so that they are not called from the system environment (e.g. `pthread_create()` calls `mmap()`) and include a copy of the PDCLib library in our repository for compilation. For the verification mode, we have written a special toolchain file reflecting the

1. `http://www.cmake.org/`

DIVINE-driven build process. At the time, the `divine compile` subcommand used to use the `ld.gold` linker[2] together with the LLVM gold plugin[3] for effective linking of LLVM assembly files. Recently, a new version of DIVINE was released – 3.2.2, which is stripped of these dependencies and implements its own linker.

Using CMake we were also able to define a long list of options for different parts of the system. For example, CVFS can be customized to simulate different file system configurations. The time representation of our environment can also be easily adjusted per build directory. Certain `nscd` parameters are configurable as well, including the maximum allowable size for a `nscd` database and the absence/presence of pruning threads.

Additionally, we have also implemented a simple testing framework consisting of python scripts and custom CMake targets, allowing to perform the correctness evaluation simply from the build system. For example, all tests available for both `nscd` and our environment can be run using `make check`, while one specific test can be evaluated with `make check-<test>`. The expected outcome of a test is specified in a separate XML file (one for every test). It is possible to define the expected (in)validity of every available property, optionally including the file name and the line number where a violation is supposed to be detected, as well as the expected textual output and the return value for the native execution. Based on these data, the build system automatically runs all the selected tests (in the given mode), parses the output from DIVINE or a test itself and prints the results of the evaluation.

## 5.4 Acceptance Tests

Our acceptance test suite consists of 6 basic tests, each targeted to cover all interfaces of a specific database (`passwd`, `hosts`, `group`, ...) by sending a few requests and verifying progress of operations and correctness of results, and 4 specialized tests, examining the extra features of the cache sub-system.

All these tests share the same skeleton of the control flow (implemented in `atests/common.c`), which could be summarized as:

1. Initialize the Closed Virtual File System (must be run explicitly).
2. Create and initialize the contents of `nscd` and NSS configuration files.

2. `http://www.gnu.org/software/binutils/`
3. `http://llvm.org/docs/GoldPlugin.html`

3. For all accessed services, register the associated (fake) NSS libraries and their respective functions implemented as stubs.
4. Create sub-directories used by `nscd` to store the database files and the UNIX domains socket for communication with clients (`/var/db/nscd` and `/var/run/nscd`).
5. Run `nscd` in a separate thread.
6. Create a number of clients (given by the `NUM_OF_CLIENTS` macro), each running in a separate thread, sending some requests and testing responses.
7. Wait for clients to finish.
8. Send the `shutdown` request to `nscd`.
9. Wait for `nscd` to finish.

The specialized, cache-targeted tests (briefly described in Table 5.1), additionally obtain and consult the cache statistics. Most of the properties captured by these tests wouldn't be possible to verify without the support for full LTL model checking.

| *Test* | *Expected behaviour* |
|---|---|
| `gc` | With the reloading disabled and the cache not being used anymore, all the cache entries should be eventually garbage collected. |
| `reload` | With the `reload-count` set to `unlimited` and the cache being non-empty, the cache memory usage never reaches zero. |
| `cache_stats` | The cache subsystem correctly maintains the statistics of cache hits and misses. |
| `invalidate` | When the `invalidate` request is received, `nscd` explicitly triggers the garbage collection but pretends that the current time is infinity, therefore all the hash entries are unconditionally marked as obsolete. |

Table 5.1: Acceptance tests examining the extra features of the cache subsystem.

# 6 Closed Virtual File System

Most programs need to do either input (reading data) or output (writing data), or most frequently both, in order to do anything useful. However, DiVinE interprets programs in a closed execution environment, where the process is not allowed to see outside its own memory segments so that every transition is fully reproducible. Therefore, I/O operations cannot be interpreted directly but have to be appropriately simulated.

One option is to emulate input values using non-deterministic choice, but this is not suitable for a purely-explicit-state representation. Luckily, if properly approached, many applications of I/O facilities have easily predictable behaviours, with almost no inherent uncertainties, such as a use of a secondary storage device to save data persistently or an inter-process communication performed through local sockets. External storage can be easily emulated in-memory without introducing additional non-determinism (apart from random failures). Likewise, if all the interacting parties are included in the system description, then the communication through I/O channels is reproducible as well.

In this chapter, we present our implementation of an in-memory file system, called *Closed Virtual File System* (or *CVFS* for short), suitable for verification with DiVinE. It provides all common low-level I/O features as defined by POSIX, including basic I/O functions, directory handling, UNIX domain sockets, file statistics and memory mapped files. Supported are all types of files – directories, regular files, symbolic links, sockets (but only AF_UNIX), FIFOs and even device special files. Additionally, it can attach to the PDCLib library to enable access through the stream interface (PDCLib implements the I/O buffering and the semantics of streams).

CVFS goes even beyond the current DiVinE capabilities and provides a multi-processing support. For the time being, this is compatible only with the native execution and for the model checking the user must select the thread-only mode. In order to make a clear separation between thread shared data and process shared data, while still maintaining the extensibility for the future development of DiVinE, we hide memory allocations and primitives for concurrency control (locks, barriers, ...) behind a set of macros, which translate differently based on the selected mode.

In terms of conformance, we aimed for a full POSIX compliance, but where the standard is unclear or does not impose a specific behaviour we instead followed the GNU C Library Reference Manual [47]. It was important for us to maintain the API compatibility between CVFS and glibc for

the sake of `nscd` verification.

In the build system, CVFS is a target on its own and compiles into a separate static library. Therefore, it is easy to extract the file system from our repository and use it also for other projects. Furthermore, CVFS can be configured through a set of CMake options to meet the requirements of different applications.

## 6.1 Design

Development of an in-memory file system is relatively a straightforward process. Normally, the work of a file system engineer is mostly oriented around designing and implementing solutions for data organization and free space management, with a goal of finding the most appropriate balance between disk space efficiency, performance and security, depending on the application area.

In our case, the task was narrowed down to selecting the most suitable data representation (i.e. data structures) and implementing the methods for concurrency control. Since we were developing the file system for the primary memory, we were able to just leverage the heap and the shared memory management systems provided by the C run-time in order to facilitate the storage capabilities. In addition, the performance or security weren't our primary concerns; instead, we had to focus on minimizing the complexity of the solution in terms of reducing the number of internal states that the file system produces. For example, we provide an option (`CVFS_SERIAL_EXECUTION`) to mask the interrupt everywhere inside CVFS except for sections where threads need to communicate.

### 6.1.1 Concurrency Control

Access to shared resources is controlled by acquisition and release of exclusive locks. Mutexes provide synchronization services for objects allocated on the heap, while semaphores help to maintain the consistency of process-shared data (currently only for the native mode). However, the lock-based synchronization may give rise to deadlocks if used carelessly. Therefore, we impose a partial ordering of all resource types, and require that each process/thread requests resources in an increasing order of enumeration. This method is commonly used to avoid the *circular wait*, a necessary condition for a deadlock situation to emerge.

Threads are mandated to acquire resources in the following order:

1. file description
2. file without a hard-link
3. file with a hard-link
   (a) for multiple *hard-linked* files: always acquire at the same branch and in the increasing distance from the `root` directory (i.e. predecessors first)
4. File descriptor table

It is not allowed to acquire multiple locks at the same level (with the exception of 3a) or out of this order.

Separately from the lock-based access control, we also maintain a reference counter for each shared object, which is only manipulated using atomic instructions (CAS), or inside a non-interruptible block of code (verification mode). The reference counting was used specifically to implement a simple garbage collection mechanism.

### 6.1.2 Basic Data Structures

At the top-most level of abstraction, CVFS represents the shared data using only 5 different data types, all declared as `struct`s and visually illustrated in Figure 6.1.

- *File* (`_cvfs_file_t`)
  Object type that is used to store the header (timestamps, protection, file type identification) and the content of a single file. Internally, a `union` is used to store file-type-specific data. For example, directories contain a list of entries – associations between file names and pointers to other instances of this structure, whereas regular files store all data inside a continuous memory block, reallocated on demand. Files are organized in a tree, reflecting the Unix file naming scheme, with the root directory at the top.

- *File Descriptor* (`_cvfs_fd_t`)
  An abstract thread-shared indicator for accessing a file. It is merely a pointer to an associated file description and exists only to store the file descriptor flags separately (currently only `FD_CLOEXEC`).

- *File Description* (`_cvfs_fdn_t`)
  The process-shared portion of an open file identification data. Points

to the associated file, maintains the current read/write file offset (for *random-access* files only), specifies the access mode and stores the file status flags.

- *File Descriptor Table* (`_cvfs_fd_table_t`)
  File descriptors (the numbers) are indices into the process-specific file descriptor table. Inside, integers get translated into references, which when followed lead through all the structures described above up to the actual (open) file.

- *Table of Drivers* (`_cvfs_driver_table_t`)
  A container of all available drivers. A driver represents a custom operational semantics of the basic I/O operations and is accessed through device special files. We designed this mechanism to implement the standard streams and the `null` device in a clear non-intrusive manner.

## 6.2 Correctness

For correctness evaluation, we have written 15 unit tests to gradually cover all supported interfaces of the file system. Tests were designed to execute at least a portion of the code from multiple threads running in parallel, so that there are actual benefits of applying the model checking approach and a realistic change of detecting and reproducing concurrency bugs.

In the native mode, it is possible to execute these tests against both our environment and the system-provided implementation of the file system. We have written the unit tests in advance, before we even started designing the file system, and ran them inside a GNU-based operating system so that we could learn how glibc behaves in various situations and maintain the API compatibility.

Results of the model checking (presented in chapter 7) are very promising. Not only we were able to perform the verification of every unit test in a relatively short time and using at most 12 GiB of memory, but the majority of concurrency bugs were identified and eliminated only after we have performed the verification. It is yet another consolidation of the fact that the (explicit-state) model checking thrives where the standard testing fails – in the area of parallel systems.

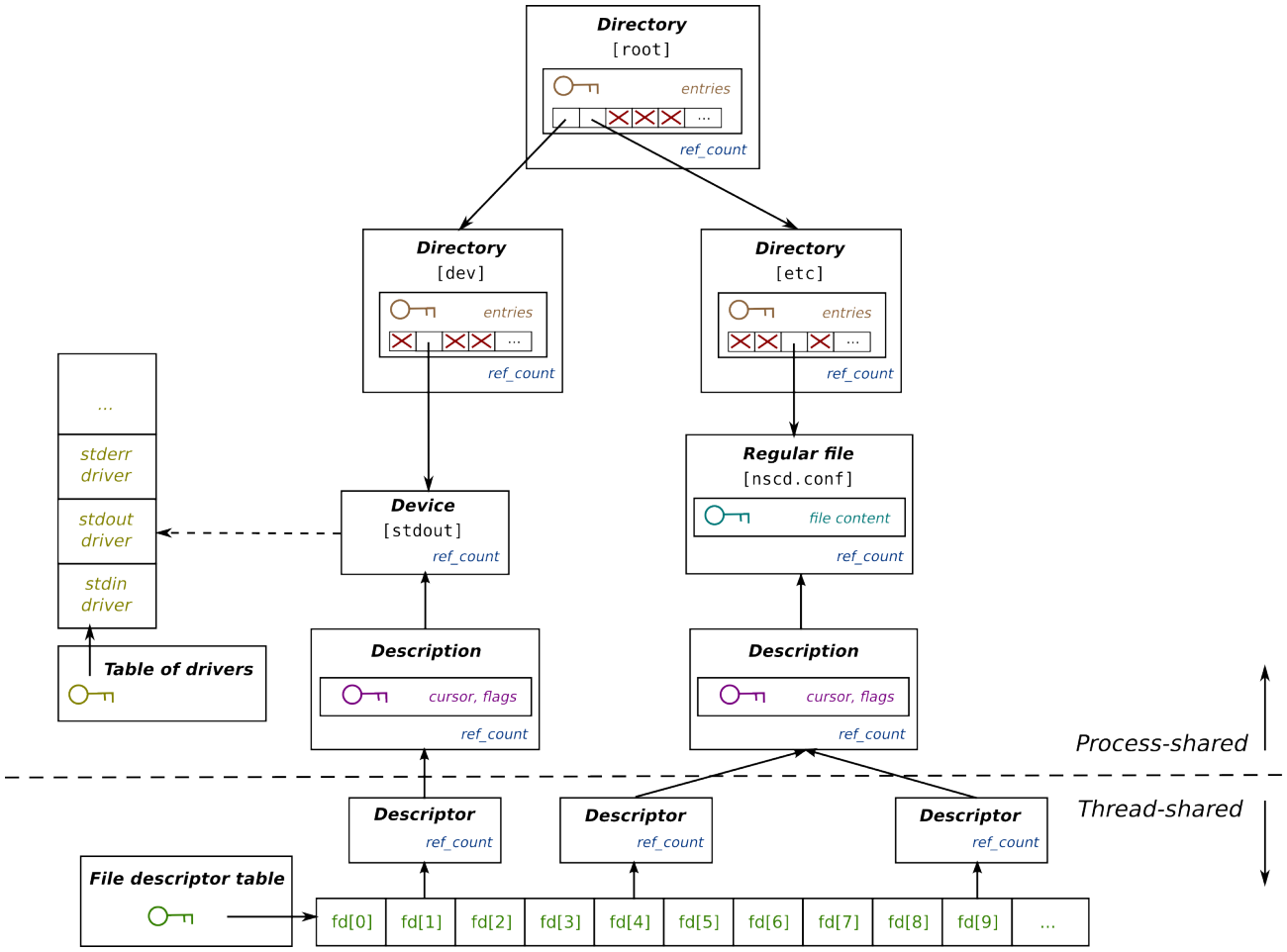


Figure 6.1: Closed Virtual File System.

## 6.3 Limitations

Naturally, the file system has its limitations. While some of them are just products of decisions made towards simplified solutions, others cannot be easily overcome due to a lack of special support from the interpreter.
Here we list the major drawbacks:

- Pointers stored into files may cause some memory leaks to remain undetected.
- `lseek()` with the argument `SEEK_HOLE` always returns the offset of the end of the file and for `SEEK_DATA` simply returns the given offset.

If a hole is to be created (when writing past the end of a file), bytes inside the hole are allocated anyway and are filled with zeroes. This is probably the simplest implementation possible while still maintaining the conformance with the specification.

- Only sockets for local interprocess communication are supported (`AF_UNIX`/`AF_LOCAL`).
- Read/Write from/to a memory-mapped file is always in-sync with the file content. Actually, we do not model delayed/buffered stores and loads anywhere in the file system.
- Memory permission flags are ignored as any protection mechanism would necessarily require an additional support from the interpreter.
- Writing beyond the boundaries of a memory mapped area is treated as an invalid dereference, i.e. the underlying file is not automatically resized nor new pages get allocated on demand (this is not required by the standard anyway).
- In order to limit the impact of the file system on the size of the state space, we do not model any non-deterministic I/O errors.

# 7 Experiments

## 7.1 User-space Modifications

First of all, we had to apply our assumptions into the DIVINE's user-space to make it compatible with our environment. Moreover, we quickly realized that it is essential to reduce the complexity of the support-code for the model checking to be feasible with the resources at our disposal.

With the LLVM linker used at the time (ld.gold + LLVM gold plugin) it was impossible to link modules with multiple definitions of the same symbol. Therefore, we had to remove the DIVINE provided stubs for I/O interfaces so that we could use our own file system instead.

Next, we have made the `malloc` family of functions deterministic by no longer assuming that a memory allocation may fail. While CVFS is robust enough to handle allocation failures, unit tests as well as acceptance tests are much simpler in design and always expect the no-random-failure scenario. With this simplification we have basically restricted ourselves from one category of bugs, but in practise they are unlikely to pose a threat for `nscd` anyway, as the typical target platform for the caching subsystem is usually a server with an abundance of hardware resources.

In addition, we have also customized the implementation of the Pthreads API. Specifically, the spurious wake-ups of `pthread_cond_wait()` were disabled for simplicity's sake and the time depended actions were made aware of our custom clocks.

Lastly, we have made all functions from `string.h` (provided by PDCLib) non-interruptible. These functions are implemented to operate on a char-by-char basis and produce a large number of internal states.

## 7.2 Configuration

The verification was performed using DIVINE 3.2.0–development snapshot from 1.11.2014. At the time, the newest release version was quite outdated but the development version was relatively stable. We enabled all available reductions ($\tau+$, `store` visibility, Heap symmetry) as well as the (lossless) Tree compression for the best memory-usage efficiency. Apart from that, DIVINE was run with default settings, which also means that the state space exploration was performed by two parallel workers.

Tests were compiled into LLVM IR using `CLANG` with `-O0`, following

our observation that the reductions perform better when the compiler optimizations are disabled.

For the emulated environment we chose the *manual clocks* as the least expensive alternative for the time representation in terms of data nondeterminism. Time progresses only explicitly and at certain places of each acceptance test. Similarly, to mitigate the path explosion problem, we have configured CVFS to always disable interrupts unless it would lead to an infinite transition in the configuration graph (i.e. only "waiting states" are allowed to interrupt the execution).

For acceptance testing, we have configured `nscd` to use the hash table with exactly 3 buckets, each initially reserved to 96 bytes. Fortunately, it was sufficient to set the maximum allowable size of every database to only 512 bytes (compare it to the default 32 MiB).

Our correctness evaluation covers only a model checking of the `safety` property (explained in Table 3.1) using the reachability analysis. All defined LTL properties are guaranteed to fail without fairness assumptions.

## 7.3 Platform

Experiments were performed on `aura.fi.muni.cz`, with 448 GiB DDR3 RAM and eight Intel Xeon X7560 2.27 GHz processors (64 cores total). The limit for the execution time was set to 1 month and each test was allowed to use at most 200 GiB of virtual memory.

## 7.4 Results

### 7.4.1 Environment Verification

All environment unit tests were verified for only two parallel workers (defined by the `NUM_OF_WORKERS` macro individually for each test). A comprehensive study on real-world multi-threading programming bugs [48] showed, that majority (96% according to their evaluation) of concurrency bugs are guaranteed to manifest if certain partial order between 2 threads is enforced.

Model checking helped us to localize various bugs that the native execution failed to detect or reproduce frequently enough. Majority of violations resulted from our own programming errors, but others were due to bugs already present in the user-space, mostly in PDCLib. Table 7.1 lists state space metrics of all env. unit tests as well as time and memory requirements of the reachability analysis as performed by DIVINE after we have eliminated all

the errors. Similarly, statistics for CVFS unit tests are recorded separately in Table 7.2.

| *Unit test* | *States* | *Transitions* | *Wall time (seconds)* | *RAM (MB)* |
|---|---|---|---|---|
| `resolv` | 93 084 | 277 403 | 619 | 1061 |
| `time` | 20 239 | 59 784 | 127 | 632 |
| `conf` | 278 | 659 | 2 | 631 |
| `error` | 290 201 | 868 440 | 1768 | 1254 |
| `herrno` | 4235 | 12 360 | 22 | 631 |
| `libintl` | 3054 | 8834 | 15 | 631 |
| `getenv` | 188 | 401 | 1 | 631 |
| `in` | 530 | 1391 | 3 | 631 |
| `dlfcn` | 84 867 | 252 592 | 681 | 1239 |

Table 7.1: Environment unit tests – reachability analysis.

### 7.4.2 NSCD Verification

Due to the high time requirements of the model checking, we decided to focus solely on the `hosts` database, which is accessed through `gethostbyname()` and `gethostbyaddr()` interfaces. Black-box testing of the `nscd` cache associated with the `hosts` database is performed by `atests/test-host.c`.

In the default configuration, the test obtains a `hostent` structure for a simulated `localhost` using both interfaces. The number of lookups is doubled as data are requested separately for IPv4 and IPv6 addresses (i.e. 4 `nscd` queries in total). However, it is possible to reduce the complexity and select only one of these calls. The control flow of the client is further wrapped in a loop with a fixed number of iterations, set by the macro `NUM_OF_ITERATIONS`. Moreover, it is possible to configure multiple clients using the macro `NUM_OF_CLIENTS` and send the queries simultaneously.

The upper half of Table 7.3 shows statistics for the simplest possible instances of the test; that is no `SCM_RIGHTS`, disabled cache pruning and

| *Unit test* | *States* | *Transitions* | *Wall time (seconds)* | *RAM (MB)* |
|---|---|---|---|---|
| `close` | 5 785 193 | 17 350 220 | 36 708 | 2944 |
| `dgram_socket` | 299 286 | 897 066 | 3774 | 2017 |
| `stream_socket` | 60 743 | 181 988 | 706 | 1083 |
| `directory` | 4 257 392 | 12 764 332 | 54 001 | 4757 |
| `fifo` | 26 499 | 78 112 | 488 | 1071 |
| `glue` | 12 942 809 | 38 810 216 | 138 532 | 11 647 |
| `link` | 1 133 963 | 3 392 492 | 14 837 | 3072 |
| `lseek` | 109 576 | 326 171 | 1219 | 1088 |
| `mknod` | 1 218 150 | 3 652 807 | 16 289 | 5002 |
| `mmap` | 142 870 | 424 601 | 6664 | 2690 |
| `open` | 1 641 818 | 4 915 251 | 21 613 | 5480 |
| `pipe` | 5805 | 17 252 | 103 | 905 |
| `poll` | 21 941 | 65 716 | 305 | 911 |
| `readwrite` | 8 708 377 | 26 124 438 | 131 292 | 10 606 |
| `scm_rights` | 64 462 | 190 973 | 586 | 1073 |

Table 7.2: CVFS unit tests – reachability analysis.

only one active client invoking single request without repetitions. DIVINE did not report safety violations for any of the interfaces.

However, moving only one step in complexity was enough to reach the specified time limit in 3 out of 4 cases. When the Tree compression is enabled, the rate of the memory consumption keeps decreasing as the exploration progresses and more similarities emerge, meaning that for large models the execution time becomes the major limitation.

We picked a single interface (`gethostbyname()` for IPv4) and went

from the simplest instance to gradually increase/enable one complexity-affecting parameter at a time. Within the execution time limit, we were only able to verify the interaction between the client and `nscd` with `SCM_RIGHTS` enabled. In this model, the client first asks for a (read-only) file descriptor associated with the `hosts` database and then performs the cache lookup himself, without the assistance of `nscd`. Since no data are stored in the cache at the first iteration, the client has to send a full query to `nscd` anyway. Unsurprisingly, `nscd` searches the cache again before contacting the associated service as a last resort. The benefits of `SCM_RIGHTS` would show up only if we increased the number of iterations or the number of clients. Unfortunately, no bugs were identified using the model checking approach. The results of this partial success are summarized in the second half of Table 7.3.

| ***Acceptance test*** | | | | | | ***States*** | ***Transitions*** | ***Wall time*** | ***RAM*** |
|---|---|---|---|---|---|---|---|---|---|
| db | interface | clients | iters | SCM | GC | | | ***(seconds)*** | ***(MB)*** |
| host | by-addr/IPv4 | 1 | 1 | no | no | 729 203 | 2 749 166 | 15 934 | 9474 |
| host | by-addr/IPv6 | 1 | 1 | no | no | 866 927 | 3 281 714 | 19 679 | 10 971 |
| host | by-name/IPv4 | 1 | 1 | no | no | 709 008 | 2 692 782 | 16 240 | 9399 |
| host | by-name/IPv6 | 1 | 1 | no | no | 801 727 | 3 030 114 | 17 672 | 10 039 |
| host | by-name/IPv4 | 2 | 1 | no | no | n/a | n/a | n/a | n/a |
| host | by-name/IPv4 | 1 | 2 | no | no | n/a | n/a | n/a | n/a |
| host | by-name/IPv4 | 1 | 1 | yes | no | 40 882 947 | 196 364 832 | 1 196 141 | 109 598 |
| host | by-name/IPv4 | 1 | 1 | no | yes | n/a | n/a | n/a | n/a |

Table 7.3: NSCD acceptance tests – reachability analysis.

# 8 Conclusion

We have shown model checking to be a viable and useful technology to include into the software development cycle for an improved quality assurance. We have extracted the GNU Name Service Cache Daemon from glibc, built an emulated environment around it, prepared a suite of acceptance tests of moderate complexity and performed a verification of the safety property using DIVINE model checker for a few instances of the system.

The goal was not to exhaustively verify `nscd` in all available configurations and for all plausible scenarios, but instead to demonstrate the current capabilities and limitations of DIVINE and provide a guideline for the implementation-level model checking of real-world software in general.

We conclude with the following observations on using software model checking for industrial applications:

- By attacking the correctness evaluation from a different angle, software model checking complements traditional testing and can significantly increase the confidence that a software product is ready to ship.
- In theory, (explicit) model checking is a simple verification strategy based on exhaustive state space exploration. If used naively, the chances of getting valuable feedback are rather small. Used properly, it can be extremely effective in increasing test coverage, detecting intricate bugs and reducing the overall cost of the product.
- Writing useful tests and properties, i.e. those that actually help to reveal unknown bugs, while minimizing the state space explosion problem requires training, experience and a basic knowledge of how model checking works, as well as tenacity and ingenuity.
- Retrospective analysis and verification of an already developed system is a costly and a time demanding process. Instead, we recommend to incorporate the manual pre-verification steps into the development cycle and perform the model checking on a regular basis, e.g. prior to public releases.
- Model checking is most effective in the area of parallel systems and especially when combined with unit testing. From our experience with DIVINE, the impression is that model checking is perfectly practical method for verification of individual units of a complex parallel

program. A high-level model checking of complete medium-sized or larger systems, corresponding to acceptance or integration testing, is currently feasible only for very simple instances and thus looses its major benefits.

# Bibliography


[1] J. Barnat, L. Brim, V. Havel, J. Havlíček, J. Kriho, M. Lenčo, P. Ročkai, V. Štill, and J. Weiser, "DiVinE 3.0 – An Explicit-State Model Checker for Multithreaded C & C++ Programs," in *Computer Aided Verification (CAV 2013)*, vol. 8044 of *LNCS*, pp. 863–868, Springer, 2013.

[2] C. Baier and J.-P. Katoen, *Principles of Model Checking*. The MIT Press, 2008.

[3] J. C. King, "Symbolic Execution and Program Testing," 1976.

[4] E. Clarke, A. Biere, R. Raimi, and Y. Zhu, "Bounded Model Checking Using Satisfiability Solving," 2001.

[5] D. Peled, "Ten years of partial order reduction," Springer Berlin Heidelberg, 1998.

[6] R. Iosif, "Exploiting heap symmetries in explicit-state model checking of software," IEEE, 2001.

[7] P. Ročkai, J. Barnat, and L. Brim, "Improved State Space Reductions for LTL Model Checking of C & C++ Programs," in *NASA Formal Methods (NFM 2013)*, vol. 7871 of *LNCS*, pp. 1–15, Springer, 2013.

[8] J. Barnat, L. Brim, and P. Ročkai, "Towards LTL Model Checking of Unmodified Thread-Based C & C++ Programs," in *NASA Formal Methods Symposium*, vol. 7226 of *LNCS*, pp. 252–267, Springer, 2012.

[9] J. Barnat, J. Havlíček, and P. Ročkai, "Distributed LTL Model Checking with Hash Compaction," *Electr. Notes Theor. Comput. Sci.*, vol. 296, pp. 79–93, 2013.

[10] V. Y. Nguyen and T. C. Ruys, "Incremental Hashing for Spin," Springer Berlin Heidelberg, 2008.

[11] B. Bingham, J. Bingham, F. M. de Paula, J. Erickson, G. Singh, and M. Reitblatt, "Industrial Strength Distributed Explicit State Model Checking," IEEE Computer Society, 2010.

[12] G. Holzmann, *The Spin Model Checker: Primer and Reference Manual*. Addison-Wesley Professional, 2003.

[13] G. J. Holzmann and A. Puri, "A minimized automaton representation of reachable states," 1999.

[14] A. Laarman, J. van de Pol, and M. Weber, "Parallel Recursive State Compression for Free," Springer Berlin Heidelberg, 2011.

[15] V. Štill, "State space compression for the DiVinE model checker," Master's thesis, Masaryk University, 2014.

[16] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Counterexample-Guided Abstraction Refinement," Springer Berlin Heidelberg, 2000.

[17] F. Lerda and R. Sisto, "Distributed-Memory Model Checking with SPIN," Springer-Verlag, 1999.

[18] J. Barnat, L. Brim, I. Černá, P. Moravec, P. Ročkai, and P. Šimeček, "DiVinE – A Tool for Distributed Verification," Springer Berlin Heidelberg, 2006.

[19] C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation," IEEE Computer Society, 2004.

[20] S. Thompson and G. Brat, "Verification of C++ Flight Software with the MCP Model Checker," in *Aerospace Conference, 2008 IEEE*, pp. 1–9, March 2008.

[21] F. Merz, S. Falke, and C. Sinz, "LLBMC: Bounded Model Checking of C and C++ Programs Using a Compiler IR," in *Verified Software: Theories, Tools, Experiments*, vol. 7152, pp. 146–161, Springer Berlin Heidelberg, 2012.

[22] A. Zaks and R. Joshi, "Verifying Multi-threaded C Programs with SPIN," in *Model Checking Software*, Lecture Notes in Computer Science, pp. 325–342, Springer Berlin Heidelberg, 2008.

[23] P. Godefroid, "Software Model Checking: The VeriSoft Approach," 2005.

[24] S. Chandra, P. Godefroid, and C. Palm, "Software Model Checking in Practice: An Industrial Case Study," ACM, 2002.

[25] K. Havelund, M. Lowry, and J. Penix, "Formal Analysis of a Space-Craft Controller Using SPIN," 2001.

[26] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda, "Model Checking Programs," 2003.

[27] W. Visser and P. Mehlitz, "Model Checking Programs with Java PathFinder," Springer Berlin Heidelberg, 2005.

[28] S. Thompson, G. Brat, and A. Venet, "Software Model Checking of ARINC-653 Flight Code with MCP," in *Proceedings of the Second NASA Formal Methods Symposium (NFM 2010), NASA/CP-2010-216215*, pp. 171–181, NASA, April 2010.

[29] H. Post, C. Sinz, and W. Küchlin, "Towards automatic software model checking of thousands of Linux modules—a case study with Avinux," 2009.

[30] A. Khoroshilov, V. Mutilin, A. Petrenko, and V. Zakharov, "Establishing Linux Driver Verification Process," Springer Berlin Heidelberg, 2010.

[31] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre, "Software Verification with BLAST," Springer Berlin Heidelberg, 2003.

[32] D. Beyer and M. Keremoglu, "CPAchecker: A Tool for Configurable Software Verification," in *Computer Aided Verification*, vol. 6806 of *Lecture Notes in Computer Science*, pp. 184–190, Springer Berlin Heidelberg, 2011.

[33] T. Witkowski, N. Blanc, D. Kroening, and G. Weissenbacher, "Model Checking Concurrent Linux Device Drivers," ACM, 2007.

[34] D. Kroening and M. Tautschnig, "CBMC – C Bounded Model Checker," in *Tools and Algorithms for the Construction and Analysis of Systems*, vol. 8413, pp. 389–391, Springer Berlin Heidelberg, 2014.

[35] T. Ball, B. Cook, V. Levin, and S. K. Rajamani, "SLAM and Static Driver Verifier: Technology Transfer of Formal Methods inside Microsoft," in *Integrated Formal Methods*, vol. 2999, pp. 1–20, Springer Berlin Heidelberg, 2004.

[36] L. Fix, "Fifteen Years of Formal Property Verification in Intel," in *25 Years of Model Checking*, vol. 5000, pp. 139–144, Springer Berlin Heidelberg, 2008.

[37] J. Souyris, V. Wiels, D. Delmas, and H. Delseny, "Formal Verification of Avionics Software Products," in *FM 2009: Formal Methods*, vol. 5850, pp. 532–546, Springer Berlin Heidelberg, 2009.

[38] J. Barnat, J. Beran, L. Brim, T. Kratochvíla, and P. Ročkai, "Tool Chain to Support Automated Formal Verification of Avionics Simulink Designs," Springer Berlin Heidelberg, 2012.

[39] D. Beyer, "Status Report on Software Verification," in *Tools and Algorithms for the Construction and Analysis of Systems*, vol. 8413, pp. 373–388, Springer Berlin Heidelberg, 2014.

[40] J. Kriho, "Enhanced parser for DVE modelling language," Master's thesis, Masaryk University, 2013.

[41] J. Havlíček, "Untimed LTL Model Checking of Timed Automata," Master's thesis, Masaryk University, 2013.

[42] J. Barnat, L. Brim, and P. Ročkai, "Parallel Partial Order Reduction with Topological Sort Proviso," in *Software Engineering and Formal Methods (SEFM 2010)*, pp. 222–231, IEEE Computer Society Press, 2010.

[43] J. Barnat, P. Bauch, and V. Havel, "Model Checking Parallel Programs with Inputs," in *22nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP)*, pp. 756–759, IEEE, 2014.

[44] V. Havel, "Generic Platform for Explicit-Symbolic Verification," Master's thesis, Masaryk University, 2014.

[45] R. Plášil, "Counterexample explanation in DiVinE model-checker," Master's thesis, Masaryk University, 2011.

[46] P. Ročkai, J. Barnat, and L. Brim, "Model Checking C++ with Exceptions," *Electronic Communications of the EASST, Proceedings of 14th International Workshop on Automated Verification of Critical Systems (AVoCS 2014)*, vol. 70, 2014.

[47] S. L. with Richard M. Stallman, R. McGrath, A. Oram, and U. Drepper, *The GNU C Library Reference Manual*. [for version 2.19].

[48] S. Lu, S. Park, E. Seo, and Y. Zhou, "Learning from Mistakes — A Comprehensive Study on Real World Concurrency Bug Characteristics," in *Proceedings of Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, (Seattle, WA), March 2008.

# A Content of the attached archive

A snapshot of the repository is made available in the attached archive `nscd-mc.tgz`. Table A.1 briefly explains the directory layout.

| *Directory* | *Description* |
|---|---|
| `atests` | `nscd` acceptance tests. |
| `cmake` | CMake modules, including the toolchain file for the DIVINE-driven compilation. |
| `cvfs` | Complete source code of Closed Virtual File System. |
| `diffs` | A record of all changes made in glibc for the verification purposes. |
| `env` | Emulated environment; i.e.: clocks, a fake dynamic linking loader, a mock DNS resolver and stubs for various system interfaces. Strictly speaking, CVFS is also part of the environment, but unlike the rest it is simulated to a full API compatibility, therefore we put it into a separate top-level directory. |
| `experiments` | A textual output from DIVINE as obtained from all the experiments. |
| `glibc` | A subset of the GNU C Library tightly-coupled with `nscd` (except `nscd` itself). |
| `native` | A support code for the native execution mode. |
| `nscd` | Complete source code of GNU Name Service Cache Daemon. |
| `pdclib` | A copy of the PDCLib library needed for the compilation in the native mode. |
| `primitives` | Basic building blocks of parallel programming, interpreted differently between the threads-only and the multiprocessing mode. |
| `tools` | Python and bash scripts used by custom targets of the build system. |

Table A.1: Directory layout of the repository.