And the Truth will make you Spin*

*Independent Research

Mathew, Cherry G.
NetBSD Developer (since 2006).
Retired FreeBSD Developer (retired 2014).

abc2024@bow.st

***Abstract*—Design Driven Development using the Spin Verifier**

a) The Problem: **Currently, the NetBSD sources are inherited from historic research, and maintained by a small, but committed group of volunteers. Often, the role of the NetBSD developer is to implement or port the design and implementation of a research project, and this involves buildup of design understanding and interpretation within the memory of a few "experts" within the community. This poses a few problems, especially for newcomers to the project:**

1) **Documentation and reading code alone, is insufficient to capture design nuances.**
2) **The "experts" can become a point of gated access or failure.**
3) **Design discussions with new developers often occur informally in the form of socratic questioning, or Q&A with an "expert" oracle.**
4) **Design and implementation may drift, simply due to the lack of a canon for design, and more importantly a way to mechanically check it for consistency.**

b) The Proposal: **I propose that the spin verifier [?] be used as the mechanical verifier, over a formal design model written in spin's promela language. This model then serves as canon, which can be logically queried via propositional logic in spin's ltl{} section.**

For architects, this takes off the burden of having to rely on their memory, random tests and bespoke models to keep state. For developers, it provides a clear "source of truth" (canon), and a process based mechanism to verify their implementation based on this source. And for project managers, there's much better visibility over the entire Q&A pipeline, analogous to, but much more powerful than a simple TDD style development process.

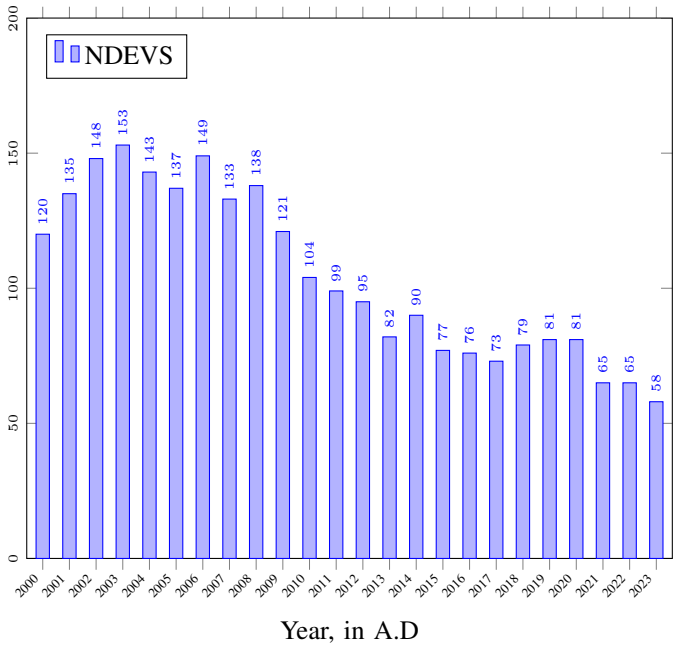
***Index Terms*—Software Design, Formal Verification, Model Checking, correctness, TDD, AI Safety**

I. Introduction

A. Motivations: The NetBSD Community - a small community developing a large codebase

The NetBSD kernel sources contain 3983983 (slightly under 4 million) Lines of Code (LoC) in the C language, as of 5th February, 2024. As we add and borrow more features and ports to the code-base, this number is bound to increase.

This research was self funded over a year long Sabbatical, from February 2023 to 2024. I am now looking to collaborate with people or organisations, for whom the ideas presented in this paper are of interest.



Year, in A.D

However, as the NetBSD kernel codebase naturally increases, the number of kernel developers in the community is shrinking. Here is a bar-chart of active Developers based on commit log data from anoncvs.NetBSD.org, from 2003 to 2023. This data shows a 62% attrition over that decade. In 2003, there were 153 active kernel developers - while by 2023, this number had shrunk to 58.

From the above, we notice that kernel code is managed by a steadily declining number of developers. This means that the burden of maintainance in terms of responsibility for lines of code per developer, steadily increases.

While this may seem like a challenging situation, clean organisation, management and automation on code can reduce the challenge significantly. This would naturally lead to a higher quality system, improving the attraction to the system, for new developers.

B. Design discovery and Documentation: Separating the "What" (Design) from the "How" (Implementation)

When a new developer joins the project, or an existing developer needs to understand the (design of) the code in

an unfamiliar area, there are a few methods to access design choices, typically as follows:

1) System documentation: Developers have an assortment of system manual (man) pages to reference the design and exported API of specific modules and components of the kernel. While many of these man pages are excellently documented, it is often the case that there may be a lag to catchup with the latest code changes. In rare cases, interfaces may not be documented at all. Finally, source code itself may have comments describing the design envelope, and particular implementation choices made.

2) Online documentation: Developers can consult design papers by the original designers of an algorithm, protocol or subsystem used in the kernel, as these are often presented at conferences such as USENIX or this one. These papers generally present design in the abstract, and any code that may be available as reference in the paper are seldom *"drop-in"* usable within the kernel code-base, as code evolves over time. Furthermore, in adopting and implementing a particular design, kernel developers may have made appropriate modifications to the design, to fit to the specific environment or requirements. For eg: Cf. notes in the OpenZFS implementation code https://github.com/openzfs/zfs/blob/zfs-0.6.3-stable/module/zfs/arc.c#L31 of the Adaptive Replacement Cache algorithm, wrt. the Original ARC paper by Meggido et. al. [**?**]

This sort of situation implies that developers have to finally resort to reading the implementation code, in order to infer final design canon.

3) Inference: Eventually, developers end up reading Implementation code, to Infer design via "Pattern recall" from their coding experience.

Most NetBSD developers have decades of experience, and while reading new and unfamiliar code, can easily recall patterns that we have been familiar with over time.

For example, consider the following C code snippet:

Listing 1
LOOP C CODE FRAGMENT

```
int j, i, array[10];

void
printarray(void)
{
  for (j = 0; j < 10; j++) {
    i = j;
    printf("array[%d] == %d\n", i, array[i]);
  }
}
```

An experienced C programmer would only need to briefly glance at these lines to quickly build the following design in their head:

"Given an array of integers of length 10, iterate through every element, and print it".

This is an inferred design, and the developer may want to cross check it via scrutiny.

4) Scrutiny: Once the design has been inferred, a developer would generally have it verified, by having it scrutinised possibly in the following ways:

1) Scrutiny via Q&A with subsystem experts. A dialogue, somewhat like the following could possibly ensue:

   Novice:
   Dear Greybeard of wisdom, my algorithm limits array size to a fixed size of 10. Is this how it's done ?

   Greybeard:
   Ah, but my innocent little novice, had you been around back in the day when we ran complex algorithms by counting with our fingers, your algorithm would have been correct. But nowadays we use these amazing things called computers which have large amounts of memory to hold large arrays. Please make the array able to hold an arbitrary number of elements.

   Novice:
   Thank you great sage of wisdom. My algorithm now looks like this: *"Given an array of integers of arbitrary length, iterate through every element, and print it".*

   Greybeard:
   But you do see that this could lead to an adversary misusing your algorithm, don't you ? They could make it allocate large amounts of memory, thus risking crashing the system - have you checked for what might be a reasonable upper limit to the array size ?

   Novice:
   I now see the constraints more clearly, great sage. I shall document them as follows:
   "Given an array of predetermined maximum size ARRAYMAX, iterate through its every element, and print it. The size of the array shall always be equal to ARRAYMAX. Similarly, the index value to the array, shall always be between 0 and (ARRAYMAX - 1). Eventually, once the algorithm runs to its completion, the index value will be equal to (ARRAYMAX - 1)"

   Greybeard:
   I now see that you have understood how to describe your algorithm more precisely. Your understanding of the algorithm itself is correct.

   Novice:
   But, but, my good sir! I do have a question: the code seems to imply a maximum array size of '10' ! How could it possibly be the case that the size is determined by ARRAYMAX ?

   Greybeard:
   Excellent question, you are looking at auto-generated code, which uses the value in

ARRAYMAX defined at $OBSCURELOCATION_WITH_OBSCURE_SHIM_SCRIPT

Congratulations! For having discovered this esoteric hidden wisdom, you are now allowed at the table of greybeards, to serve soup and clear the tables.

Novice: Such an honour, good sire!

Notably, this exchange has brought in several design considerations that would not have been obvious by just reading the code snippet, such as the potential DoS vulnerability, or the dynamically generated code situation. At this point, the Novice has caught up quite a bit with the design. They have managed to explain the design succinctly and accurately, both to themselves and to others - but only in natural language.

2) Scrutiny via comparing with VCS commit log descriptions. An example commit log for the above algorithm in code might look as follows:

Listing 2
RCSLOG LISTING

```
revision 1.13
date: 2024/02/5 03:46:49; author:
    greybeard; state: Exp; lines: +0 -1
Summary: This function iterates through a
    fixed global array and prints its
    contents.
```

Recall that the array size, although appearing to be a constant of value '10', is actually dynamically generated and pasted into code by $OBSCURE_FANCY_MECHANISM - and that this fact is not clarified in the commit log.

3) Scrutiny via running Unit tests as probes into the design space and filling the gaps through inference.
While the first two ways are not rigourous (as humans can generally be assumed to be fallible), Unit testing, while rigourous, fails to exhaustively cover all possibilities to comprehensively understand the design. It merely serves as probes into specific scenarios in the design, at specific time/state snapshots during the execution of the implementation code. Furthermore, the "big picture" is left to the inference of the reader.
For eg: a typical test for the above code might be:

Listing 3
PROPOSITIONAL LOGIC PROBES APPLIED TO SYSTEM STATE

```
void
test_printarray(void)
{
  assert(i >= 0);

  printarray();

  assert(i < 10);
}
```

Note that the probes for sane index values are only at two sample points: one before, and one after printarray() runs. From the test's expressivity point of view, there is no way to check for the "big picture"; ie; what printarray() might do to the index variable 'i' during its execution.

Furthermore, note how the constant size '10' is used as the array size. If the Unit tests had been written by a developer who did not understand the inferred design, per Section **??**., they would merely look at the source code, and naturally use the constant value '10' as the maximum value of the index, completely missing the situation that if the code were autogenerated, their test cases would not cover any ARRAYMAX that were greater than 10. This problem would only be revealed, if the autogenerating code were to use a value of ARRAYMAX greater than 10, to generate a new constant assignment for the index variable, 'i', since the `'assert(i < 10)'` in the unit test would fire.

Worse still, someone trying to infer the design from the tests, would completely miss the need for ARRAYMAX. If they went back to the code, and the developer were thorough enough to mention the entire design in comments, this would simply bring them back to the situation at Section **??**, above. However, there would be further doubts:

- Could the writer of the unit test have missed the fact that the need for ARRAYMAX is documented elsewhere ? Could the developer of the implementation code have overlooked the unit test, and missed the inconsistency ? Is the comment in the source code up-to date ? Are there other commits where design decisions have been recorded ?

- In an inconsistent situation like this, which design is to be believed ?

This is the question of Canon. There needs to be a single, agreed upon location that fully and canonically describes the design of the code.

We thus see that merely using the above four methods to discover, understand and document design is inadequate to correctly capture, record and communicate it. Furthermore, there is no current mechanism within the codebase, to mechanically verify the correctness of any inferred design, apart from the sparse state space probes via unit testing.

II. Proposed Solution:

A. Design Driven Development (DDD)

(Pronounced **"D Cubed"**)

I introduce DDD as a potential solution to the problem of design discovery, and documentation, described above.

Inspired from its cousin "Test Driven Development (TDD)", DDD allows for clean separation of Design and Implementation in the following ways:

1) Model, as Explicit Design: Given a suitably expressive language, a design can be written down explicitly, and with verifiable logical consistency. [**?**]

We call such an explicitly written design, a "model".

Developing a model has two advantages:

- Developers can use the model as a rigourous reference design and write a high fidelity implementation in the C language.
- Implementations can have context specific flavours, not limited to the specific OS environment eg: Linux, *BSD, etc.

  Cf. patch posted to the LKML [**?**]: NetBSD/cbd.c and linux/cbd.c for an example where the model anchors the implementations of a toy block driver for both NetBSD and Linux.

2) Model, as Canon: Once the model is written out, it becomes canonical, ie; the ultimate source of truth, for the intended design.

As with the current process described above in Section **??**., the newly written model needs Scrutiny. Fortunately, this falls in the domain of Formal Specification and Verification by Model Checking. When a model is written in a particular tool's expressive language, its' operating envelope can be specified via a special kind of logic called "Linear Temporal Logic" (LTL). A specification in LTL is often called a "property", a "temporal assertion", a "Logical Invariant" (or "Invariant" in short).

When scrutiny is complete, design inputs will have been captured in an explicit model + invariants form, and we can now automate the verification of this form, using an algorithm developed by Amir Pnueli in 1977. [**?**]

This algorithm, which is now variously called "Formal Verification", or "Model Checking", confirms that there is no instance of model execution state, where the invariants are violated. It follows therefore that guaranteed "correctness" of model designs are only as good as the *model* + *invariant* pair provided by the designer.

3) Model Evolution: Kernel design, like any other software project, is not static, and needs to evolve with changing needs such as newer hardware and application requirements. When a change in the design is required, it is first made explicit through and via update of the Model, followed by Automated Scrutiny (Model checking/Formal Verification). Finally, the implementation code is updated to conform to the latest verified model.

While this process is more onerous than the current process of ad-hoc code updation (sometimes) followed by documentation and unit test updates, I make the argument that this is tolerable in aggregate because infrastructure such as kernel code, is mostly mature, and evolutionary changes would be, by their very nature, incremental and not disruptive at large.

4) Model Implementation: Once there are stable, verified and canonical models in our codebase, those models become the canonical source of design truth. All design implementations should conform to these models with high operational fidelity. (See below for more on fidelity).

5) Model $\Leftrightarrow$ Implementation Fidelity: Once a C-language implementation is available (along with suitable documentation and Unit tests), a model extractor tool is run over it, to "extract" the underlying model. This extraction process requires guidance from someone who understands both the design, and the implementation.

Once the extracted model source is available, it can be Formally verified using the original invariants via "Automated Scrutiny" (in this case, verification by model checking). If the scrutiny passes, we infer that the extracted model, and the designed model satisfy the same operational constraints, and are therefore of high operational fidelity to each other.

Note that to the best of my knowledge, there is no rigourous technique of polynomial computational complexity, to formally verify the equivalence of two formal models, in the general case. While there are theoretical proofs to show that strongly unambiguous Buchi Automata, may be mechanically and formally proved to be equivalent [**?**] [**?**], it is unclear if any current programming languages can be converted exclusively to strongly unambiguous Buchi automata. We thus use the notion of "High Operational Fidelity", as follows: If two models satisfy the same invariant temporal logical constraints to their operational state space, they are deemed to be of "High Operational Fidelity" with respect to each other.

B. Prior art: MBSE

While DDD may superficially look like Model Based Systems Development/Engineering (MBSE), it is different in the following ways:

1) Scope: DDD has Narrower scope - it doesn't include business/product level information capture, like MBSE.

2) Codegen: Unlike DDD, State of the Art MBSE tools normally end with codegen, with no operational fidelity checking of the implementation (see below).

3) Fidelity checking: DDD does not end with modelling and invariant specification - there is a further step, where the corresponding implementation is parsed, and the design is extracted from it. This extracted design, is then compared with the original design for operational fidelity. This process ensures, that DDD models and implementations are in tight sync and of high fidelity with each other.

III. Implementing Design Driven Development (DDD)

Given the large source code base of the NetBSD kernel, implementing DDD would involve ascertaining the semantic layout of the source, and partitioning them into useful subsets of manageable size. We will call these subsets of source code, "hub"s. While it is likely computationally prohibitively expensive, to attempt to verify the entire kernel sources in their entirety, a verifier can be applied piecemeal to corresponding designs for sources partitioned into "hubs". We propose the spin verifier [**?**] as the verifier of choice, and its companion

tool, modex, as the model extractor. A spin kernel hub is defined as a collection of kernel source code that implements a consistent API or a well defined state machine. Hubs under scrutiny may be viewed similar to how atf(7) tests are currently organised in the NetBSD source base. They are easily integrable into the build system, since the spin tools are command line driven, and thus easy to hook into the Makefile based build tree, as well as the "autobuilds" Continuous Integration system.

Let us now look at what a trivial DDD implementation of a spin hub would look like in practice. There are two approaches based on whether the hub is being designed and implemented anew, or if a hub is being setup for "spin"-ing from existing source code.

A. New Designs:

1) Capture formal model from description/idea: Let us start with the informal requirement per the novice description in Section **??**:
"Given an array of predetermined maximum size ARRAYMAX, iterate through its every element, and print it. The size of the array shall always be equal to ARRAYMAX. Similarly, the index value to the array, shall always be between 0 and (ARRAYMAX - 1). Eventually, once the algorithm runs to its completion, the index value will be equal to (ARRAYMAX - 1)"

We now proceed to write this requirement more formally, in spin's promela language. We begin with the first part of the requirement:

"Given a predetermined maximum array size value, ARRAYMAX, iterate through every element, and print it."

Listing 4
LOOP PROMELA CODE FRAGMENT

```
#define ARRAYSIZE ARRAYMAX

int j, i, array[ARRAYSIZE];

active proctype printarray()
{
  for (j : 0 .. (ARRAYSIZE - 1)) {
    i = j;
    printf("array[d] == %d\n", i, array[i]);
  }
}
```

2) Capture invariant constraints to model operation: Next we shift our attention to the remaining description: *"The size of the array shall always be equal to ARRAYMAX. Similarly, the index value to the array, shall always be between 0 and (ARRAYMAX - 1). Eventually, once the algorithm runs to its completion, the index value will be equal to (ARRAYMAX - 1)"*

We see that this informal description contains some formalisable invariant properties - temporal logical assertions about the system state which are expected to hold in all states of the model.

Promela syntax allows us to group these under a special heading called "ltl", as follows:

Listing 5
INVARIANT ASSERTIONS IN LINEAR TEMPORAL LOGIC

```
ltl
{
  true
  && (always (ARRAYSIZE == ARRAYMAX))
  && (always ((i >= 0) && i <= (ARRAYMAX - 1))
      )
  && (eventually always (i == (ARRAYMAX - 1)))
}
```

3) Automated Model Checking: With these snippets suitably in a source file called printarray.pml (See: Appendix A, below, for listing), the spin verifier can be run over the model to verify it by model checking. See the spin documentation for details: [**?**]

This step allows the designer to refine the model iteratively, by going back to steps 1) and 2) above if needed, until the model+invariants are satisfactory.

4) Model Implementation: A reader with familiarity with the C language, would be able to inspect the promela model in Listing **??** and the C code snippet in Listing **??** to get intuition on how the promela model may be implemented in C. While out of scope of this paper, it would be important to understand that while the promela code looks similar to C in this instance, its' "execution model" is very different from that of C. We haven't come across this situation, because the "for (: ...) ..." construct has hidden the "guard statement" concept that is critical to understanding how the promela code flows. Please refer to the spin manual for further detail: https://spinroot.com/spin/Man/for.html

Once we have implemented the model in C language, as in Listing **??**, we are ready to harness the real power of DDD.

5) Model extraction: Spin comes with a companion tool called "Modex" (short for "Model Extractor"). Modex is able to (barely) parse modern C files, using a guided mechanism called a "Harness".

Below is a harness listing, to tell modex what C source file to parse, and which function to attempt to extract the model from.

Listing 6
MODEX HARNESS LISTING

```
// Spin model extractor harness for printarray
    .c
//
%F printarray.c
%X -n printarray

%P
#define ARRAYSIZE ARRAYMAX

active proctype printarray()
{
#include "_modex_printarray.pml"
}

ltl
{
  true
  && (always (ARRAYSIZE == ARRAYMAX))
  && (always ((i >= 0) && i <= (ARRAYMAX - 1))
      )
  && (eventually always (i == (ARRAYMAX - 1)))
}

%%
```

Note the ltl{} subsection listing within the harness. The LTL invariants in the extracted model, and the design based model need to be identical, for operational fidelity to be verified. A better way to arrange for this would be to have the ltl{} section have its own file, which can then be directly included in both models.

Below is the result of applying the harness in Listing **??** above, to modex, when extracting a formal model from the C function in Listing **??** .

Listing 7
MODEX EXTRACTED MODEL FROM PRINTARRAY.C

```
// Generated by MODEX Version 2.11 - 3
    November 2017
// Wed 07 Feb 2024 10:42:26 PM IST from
    printarray.c

int j;
int i;
int array[10];
active proctype printarray()
{
      c_code { now.j=0; };
L_0:
      do
      :: c_expr { (now.j<10) };
          c_code { now.i=now.j; };
c_code { now.j++; };
      goto L_0;
c_code { now.j++; };
      :: else; -> break
      od;
;
}
ltl
{
  true
  && (always (ARRAYMAX == ARRAYMAX))
  && (always ((i >= 0) && i <= (ARRAYMAX - 1))
      )
  && (eventually always (i == (ARRAYMAX - 1)))
}
```

If either model fails to satisfy their respective verification runs, DDD mandates that both the modelling and the implementation steps above be re-iterated, until verification succeeds. At this point, we say that the design in "printarray.pml" and the implementation in "printarray.c" have high operational fidelity with each other.

See Appendix B below ("DDD in action"), for some illustrative examples, in action.

B. Existing Designs: "Model Driven Reverse Engineering"

Let us assume that our source "hub" consists of the C stub code in Listing 1, above **??**.

1) Discover the design: Manual inference of design, as described in Section **??**, "Design discovery and Documentation" would give us an informal description of the design implemented in "hub" code. We re-print it below for clarity:

"Given an array of predetermined maximum size ARRAYMAX, iterate through its every element, and print it. The size of the array shall always be equal to ARRAYMAX. Similarly, the index value to the array, shall always be between 0 and (ARRAYMAX - 1). Eventually, once the algorithm runs to its completion, the index value will be equal to (ARRAYMAX - 1)"

2) Model the design: Make design explicit as in Section **??**. "Proposed Solution: Design Driven Development(DDD)"

At this point, we are ready to proceed per **??** "New Designs", above. Obviously, since we start with the hub source, we can skip **??** "Model Implementation".

The one constraint we have is that, unlike in a new design, retrofitting an inferred design model will have to minimise changing the hub source, until our confidence in the verification of the model+ltl pair (recall that the ltl section formally expresses the invariants in the design) by model checking, is strong enough to convince us that the existing implementation has incorrect or buggy flaws.

I expect that this process will unearth deep design bugs lurking in the kernel source. I am in the process of building a set of models as above, discovered from the entire set of NetBSD kernel source hubs. I call this set of models "SpinOS". [work in progress].

IV. The case for spin

The reader would have noticed that the spin verifier was rather abruptly dropped in, as the choice of verifier in this paper. Here we make a case for why this choice was made.

The case for spin, as the modelling language of choice for NetBSD includes:

A. Familiarity

Spin's model language is called Promela, and its syntax is easy to understand for C programmers.

B. C-toolchain

The spin toolchain has clean integration into the C-toolchain ecosystem. It is itself written in ANSI C, and is thus a natural fit for a C language codebase.

C. Embedding

spin provides direct support for embedded C code. [**?**]

V. Further work needed and help wanted

A. Model Extractor (Modex)

The spin model extractor, called "Modex" requires further development to be able to be used as a drop-in parser for NetBSD kernel sources, in order to extract models from them.

- The Modex parser needs overhaul to catchup with C99
- The Modex "Harness" language needs re-think/design in order to make it more expressive and powerful.

VI. Future research work

A. Scrutiny via LLM Q&A

While formal models+ invariants are rigorous and precise, understanding them requires reading and inferring semantics from them. This involves a form of scrutiny similar in spirit, to **??**, above. Similar to how the current process of design learning involves Q&A with subsystem experts, it is proposed that a suitably trained Large Language Model (LLM) might be able to answer factual questions based on the provided model+invariant Formal Specification pairs.

An LLM, can operate in slightly different variants:

1) Static: The "Static" variant, is where the LLM limits its answers based on the current model/invariant pair information. This limitation is about what the model covers at present, ie; questions of "what" in its currently implemented state.

2) Dynamic: A more interesting "Dynamic" variant, might use the "Retrieval Augmented Generation" (RAG) technique, where the student poses new invariants to the Formal system via an LLM→RAG pipeline. Here, the RAG can drive the model checker in the backend as part of "Retrieval", and then report back (via "Augmented Generation") on the question posed as invariants, based on the formal satisfiability of the question.

This provides the questioner with better semantic information about the operating envelope of the model, by asking "what if" questions, to the RAG, before even having to think about implementation.

For eg: a questioner could ask - "Can printarray() be used to discover the size of an array ?" - to which, given that we have an LTL invariant "eventually always (i == (ARRAYMAX - 1)" , the RAG has sufficiently precise information to answer this question, by running the model checker with the additional invariant "eventually (i == (ARRAYSIZE - 1))" and give a correspondingly truthful answer.

B. Document generation

Given sufficient model/invariant and implementation source code as training data, it is concievable, that Documentation can be auto generated directly from this data, using modern Generative AI techniques.

C. Code generation

It is also concievable that Code could be auto generated using a suitably trained LLM, simply given a model/invariant source input. The key difference with current state of the art, is that any generated code will have to pass the extraction→fidelity verification by model checking path, to be accepted into the codebase. We thus have built in "AI safety".

Acknowledgements

Much gratitude to Dr. Gerard Holzmann (Author of spin) for prompt Q&A responses on various aspects of Formal Verification and the Spin verifier in particular.

Shout-out to Matthew Green, Taylor R. Campbell and David Holland, for prompt and precise answers to Q&A around model+invariants in the implicit design of the NetBSD kernel. Big props to all the NetBSD developers who make the NetBSD developer community a fun, inclusive and safe online space to be in, and learn things at.

Finally, many thanks to Anuvrat Pareshar, Ashik Salahudeen, Biby Sam Varghese, Konarak Ratnakar and Markus Graeser for kindly reviewing drafts of this manuscript, and coming up with useful corrections and suggestions.

REFERENCES

[1] "ARC: A Self-Tuning, Low Overhead Replacement Cache", Nimrod Megiddo and Dharmendra S. Modha, 2nd USENIX Conference on File and Storage Technologies (FAST 03), 2003, San Francisco, CA

[2] "An automata-theoretic approach to automatic program verification", by Moshe Y. Vardi, and Pierre Wolper, Proc. First IEEE Symp. on Logic in Computer Science, 1986, pp. 322-331.

[3] Mathew, Cherry G. (2023-10-23). *"Formal models as source of truth for Software Architects."*. LKML.org Retrieved 2023-12-29.

[4] A. Pnueli. The Temporal Logic of Programs. In Proceedings of the 18th IEEE Symposium Foundations of Computer Science (FOCS 1977), pages 46-57, 1977.

[5] "A Unified Approach For Showing Language Containment and Equivalence Between Various Types of co-Automata." E.M. Clarke, LA. Draghicescu, R.P. Kurshan September 1989 CMU-CS-89-192

[6] "Equivalence and Inclusion Problem for Strongly Unambiguous Buchi Automata". Nicolas Bousquet ENS Chachan, France and Christof Loding RWTH Aachen, Informatik 7, 52056 Aachen, Germany

[7] G. J. Holzmann *"Verifying Multi-threaded Software with Spin"*. Spin Website, Retrieved 2023-12-29

[8] Mathew, Cherry G. (2023-09-02) *"ARC model specified in spin-root/promela"*. tech-kern@NetBSD.org mailing list, retrieved, 2023-12-29.

APPENDIX A: CODE LISTINGS

Listing 8
PRINTARRAY.C: C SOURCE CODE

```c
/*
 * This is companion source for the AsiaBSDCon
     2024 paper titled:
 * "And the Truth will make you Spin", by
     cherry@NetBSD.org
 */
#include <stdio.h>
#include <assert.h>

int j, i, array[10];

void
printarray(void)
{
        for (j = 0; j < 10; j++) {
                i = j;
                printf("array[%d] == %d\n", i,
                     array[i]);
        }
}

void
test_printarray(void)
{
        assert(i >= 0);

        printarray();

        assert(i < 10);
}

int main(int argc, char **argv)
{
        printf("Running test_printarray()\n");
        test_printarray();
        printf("Running printarray()\n");
        printarray();
        printf("All done. Exiting\n");
}
```

Listing 9
PRINTARRAY.PRX: MODEX EXTRACTION HARNESS

```
/*
 * This is companion source for the AsiaBSDCon
     2024 paper titled:
 * "And the Truth will make you Spin", by
     cherry@NetBSD.org
 */

// Spin model extractor harness for printarray
    .c
//
%F printarray.c
%X -n printarray

%P
#define ARRAYSIZE ARRAYMAX

active proctype printarray()
{
#include "_modex_printarray.pml"
}

ltl
{
  true
  && (always (ARRAYSIZE == ARRAYMAX))
  && (always ((i >= 0) && i <= (ARRAYMAX - 1))
      )
  && (eventually always (i == (ARRAYMAX - 1)))
}

%%
```

Listing 10
MODEX EXTRACTED MODEL VIA PRINTARRAY.PRX

```
// Generated by MODEX Version 2.11 - 3
    November 2017
// Thu 08 Feb 2024 02:14:07 AM IST from
    printarray.c

int j;
int i;
int array[10];
active proctype printarray()
{
      c_code { now.j=0; };
L_0:
      do
      :: c_expr { (now.j<10) };
          c_code { now.i=now.j; };
c_code { now.j++; };
      goto L_0;
c_code { now.j++; };
      :: else; -> break
      od;
;
}
ltl
{
  true
  && (always (ARRAYMAX == ARRAYMAX))
  && (always ((i >= 0) && i <= (ARRAYMAX - 1))
      )
  && (eventually always (i == (ARRAYMAX - 1)))
```

```
}
```

Listing 11
PRINTARRAY.PML: MODEL PROMELA CODE

```
/*
 * This is companion source for the AsiaBSDCon
     2024 paper titled:
 * "And the Truth will make you Spin", by
     cherry@NetBSD.org
 */

#define ARRAYSIZE ARRAYMAX

int j, i, array[ARRAYSIZE];

active proctype printarray()
{
  for (j : 0 .. (ARRAYSIZE - 1)) {
    i = j;
    printf("array[d] == %d\n", i, array[i]);
  }
}

ltl
{
  true
  && (always (ARRAYSIZE == ARRAYMAX))
  && (always ((i >= 0) && i <= (ARRAYMAX - 1))
      )
  && (eventually always (i == (ARRAYMAX - 1)))
}
```

APPENDIX B: DDD IN ACTION

Listing 12
SPIN VERIFIER RUN WITH ARRAY OF MAX ENTRIES 10

```
$ spin -DARRAYMAX=10 -search printarray.pml
ltl ltl_0: (((1) && ([] ((10==10)))) && ([]
    (((i>=0)) && ((i<=(10-1)))))) && (<> ([]
    ((i==(10-1)))))

(Spin Version 6.5.2 -- 6 December 2019)
        + Partial Order Reduction

Full statespace search for:
        never claim             + (ltl_0)
        assertion violations    + (if within
            scope of claim)
        acceptance   cycles     + (fairness
            disabled)
        invalid end states      - (disabled by
             never claim)

State-vector 28 byte, depth reached 87, errors
    : 0
      169 states, stored (250 visited)
      118 states, matched
      368 transitions (= visited+matched)
        0 atomic steps
hash conflicts:         0 (resolved)

Stats on memory usage (in Megabytes):
    0.009       equivalent memory usage for
        states (stored*(State-vector +
        overhead))
    0.286       actual memory usage for states
  128.000       memory used for hash table (-
      w24)
    0.534       memory used for DFS stack (-
      m10000)
  128.730       total actual memory usage


unreached in proctype printarray
        (0 of 11 states)
unreached in claim ltl_0
        _spin_nvr.tmp:30, state 44, "-end-"
        (1 of 44 states)

pan: elapsed time 0 seconds

$
```

Listing 13
MODEX EXTRACTION + SPIN VERIFICATION WITH ARRAY OF MAX ENTRIES 10

```
$ modex printarray.c;spin -DARRAYMAX=10 -
    search model
MODEX Version 2.11 - 3 November 2017
created: model and _modex_.run
ltl ltl_0: (((1) && ([] ((10==10)))) && ([]
    (((i>=0)) && ((i<=(10-1)))))) && (<> ([]
    ((i==(10-1)))))

(Spin Version 6.5.2 -- 6 December 2019)
        + Partial Order Reduction

Full statespace search for:
        never claim             + (ltl_0)
        assertion violations    + (if within
            scope of claim)
        acceptance   cycles     + (fairness
            disabled)
        invalid end states      - (disabled by
             never claim)

State-vector 68 byte, depth reached 67, errors
    : 0
      130 states, stored (192 visited)
       91 states, matched
      283 transitions (= visited+matched)
        0 atomic steps
hash conflicts:         0 (resolved)

Stats on memory usage (in Megabytes):
    0.012       equivalent memory usage for
        states (stored*(State-vector +
        overhead))
    0.284       actual memory usage for states
  128.000       memory used for hash table (-
      w24)
    0.534       memory used for DFS stack (-
      m10000)
  128.730       total actual memory usage


unreached in proctype printarray
```

```
        model:16, state 6, "␣now.j++;␣"
        (1 of 12 states)
unreached in claim ltl_0
        _spin_nvr.tmp:30, state 44, "-end-"
        (1 of 44 states)

pan: elapsed time 0 seconds

$
```