IEEE Newfoundland & Labrador Section

HARPO (HARdware Parallel Objects) Language

Static Program Verification Using Boogie Intermediate Verification Language

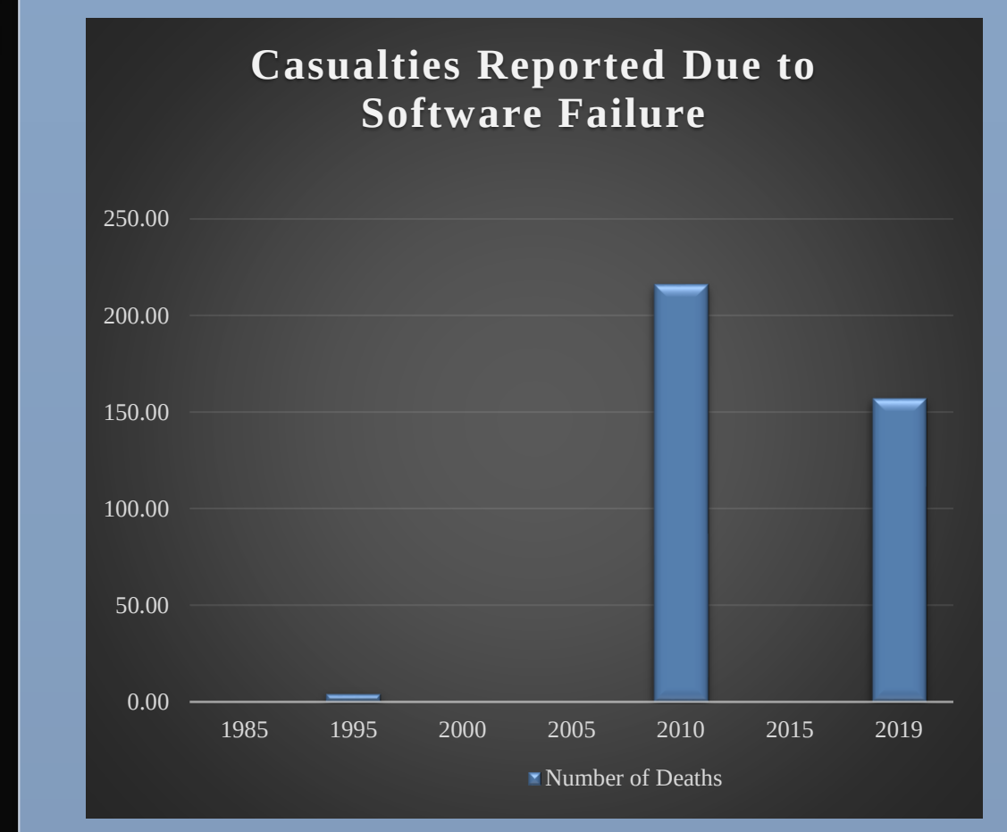
Introduction

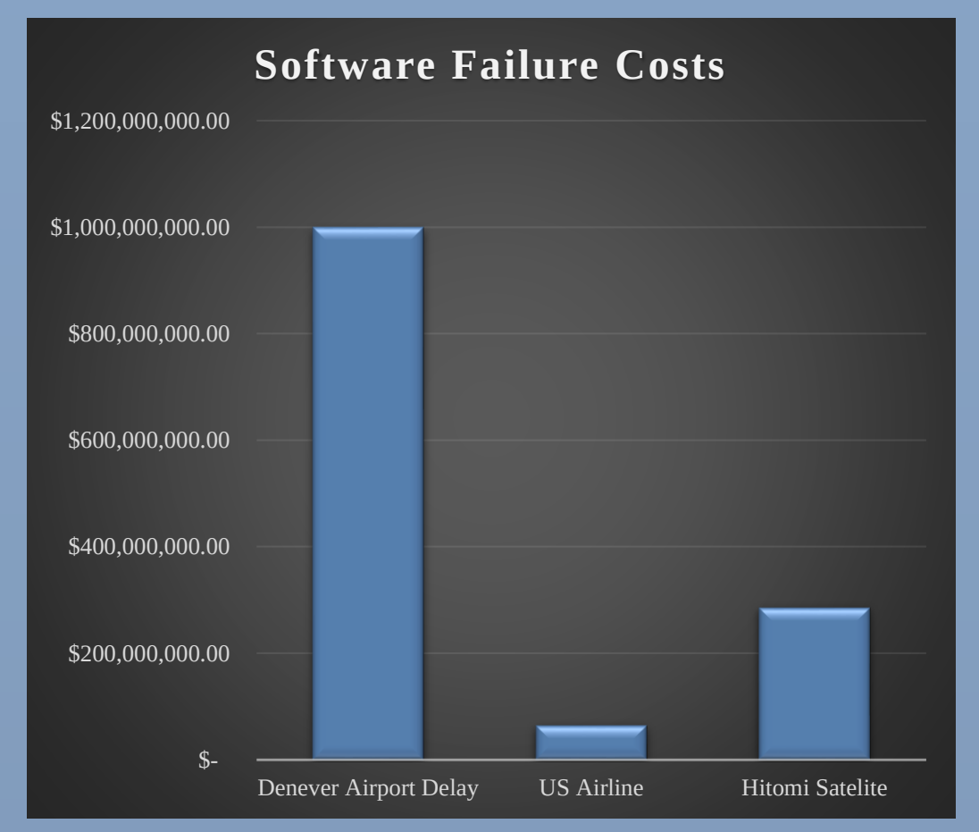
Errors in the software are hazardous. Testing software, before its deployment, may catch some errors present in the software. However, testing does not guarantee their absence. Therefore various forms of analysis can be used.

Historical Software Failures:

1. The erroneous angle-of-attack sensor reading in Ethiopian Airline Boeing 737 Max 8 triggered the anti-stall software system causing the plan to hit the ground at 575 miles per hour in 2019. Software requirements were partially blamed for 157 deaths reported in the accident. The software was not designed to behave correctly in the presence of sensor failure.
2. The loss of 286 million US dollars Hitomi satellite spun around due to altitude control failure in 2016.
3. On June 1, 2009, Air France Flight 447 from Rio de Janeiro to Paris crashed into the Atlantic Ocean, killing all 216 passengers and 12 crew members. Prior to the disappearance of the A330 aircraft, the automatic reporting system sent messages indicating disagreement in the airspeed readings, which led investigators to believe that the pilot probe sensors did not "accurately" measure airspeed and the autopilot may have automatically disengaged.
4. US Airline lost 65 million US dollars when airline reservation system reported flights as sold out.
5. In 1999, Arian 501 self exploded after 40 seconds of its launch due to specification and design errors in the program of inertial reference system.
6. Denver Airport opening delayed indefinitely in 1994, partly due to malfunctioning baggage sorting system delayed the opening and missing its fourth deadline $ 1 billion over budgeted.
7. Four patients died, and two left with life-long injuries due to erroneous software controlling the Therac-25 radiation machine. It was utterly relying on the computer for its security between 1985 and 1987.

We trust our most critical systems in ways they do not deserve by formally verifying both their design and implementation. We can prove correctness in systems where failure is unacceptable. The formal software verification approach can increase the productivity of a programmer and decrease the cost of dependable software production by reducing the cost of changing the software in and after development cycle. Formal verification has *Model Checking*, *Symbolic Simulation*, and *Interactive Theorem Proving*. A standard approach for formal program verification is to use the automated theorem-proving technique.





Research Motivation

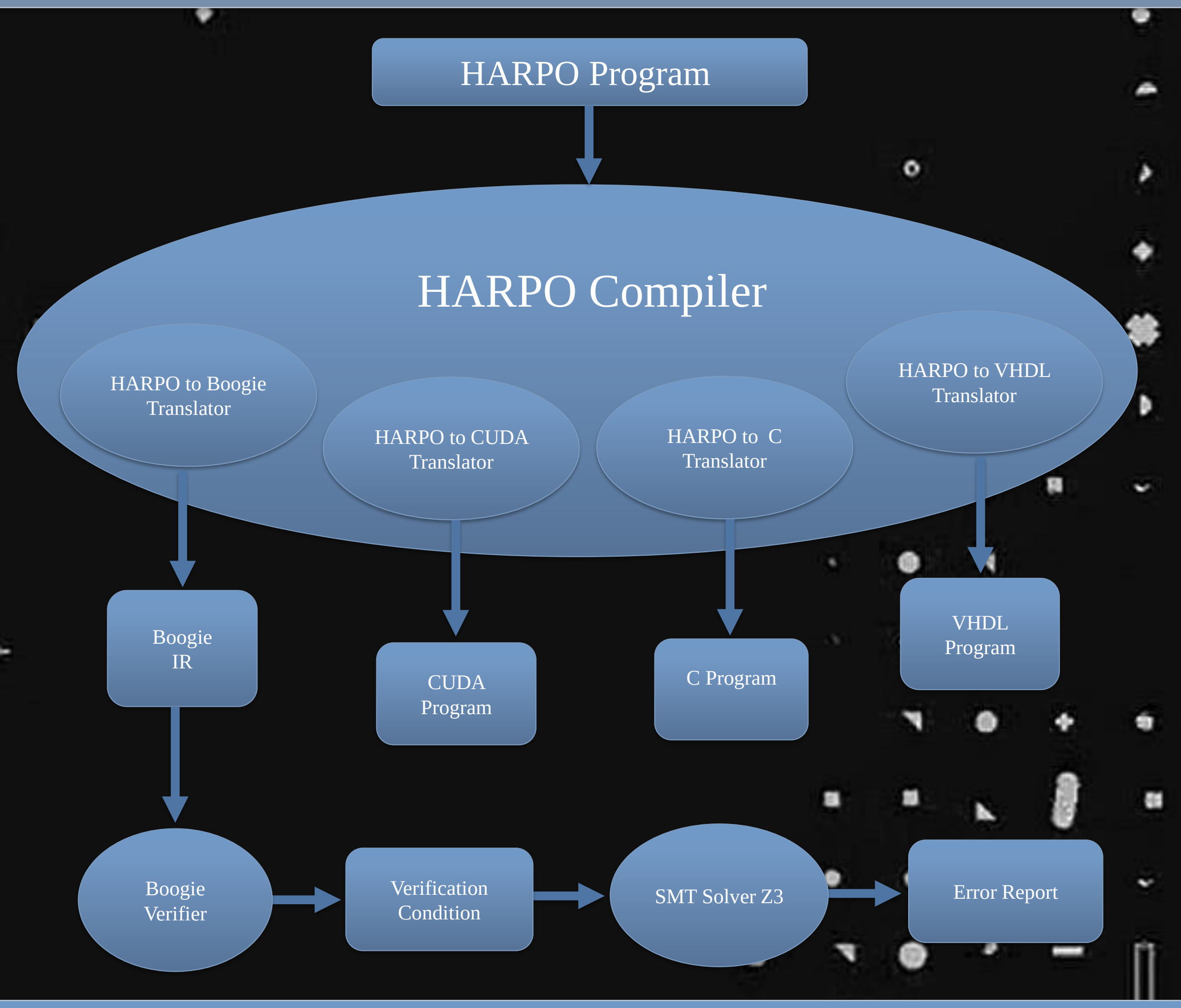
Software runs our world, becoming more sophisticated and infinitely complex from our cell phones to our cars and medical devices. We entrust our sensitive information even our lives, to a complicated maze of algorithms where safety and security are critical. Even with rigorous testing, there are countless ways for things to go wrong. So how can we make these systems safe? By going back to first principles and mathematically proving that software is correct. Just like a builder who relies on a blueprint for construction. Programmers rely on a critical system specifications outlining how it should and should not perform these specifications are translated into coded instructions, but often the specification and code do not match up. This causes flaws that lead to breakdowns malfunctions or cyber cyberattacks. With formal verification. We can analyze deep properties of the code to prove nothing is lost in translation and that the software does only what it's intended to do. We do this by converting both the code and the specification into mathematical representations, which are then checked against each other by mathematical proof. If they do not align, we isolate the cause of the mismatch and correct it.

Glimpse of Complexity: Imagine a labyrinth maze. Now multiply it by billions. The number of possible routes through the complicated maze of modern software is larger than the number of atoms in our solar system. With conventional testing it would take thousands of years to test each path individually for flight. Therefore critical routes are often missed. Formal verification allows us to evaluate all possible scenarios and the entire state space all at once. As a result we eliminate entire classes of flaws dramatically improving the safety and security of our critical systems and significantly reducing costs down the road. These days.

HARPO Programming Language

HARPO (HARdware Parallel Object) project started in 2006. The mission of the HARPO project is to develop an industrially viable concurrent language that supports the wide variety of reconfigurable processor architectures and GPUs with functional correctness properties using automated verification. This poster contains information on HARPO Verifier's methodology and development. Since HARPO language is intended target the systems with computation engines including Microprocessors, FPGAs, and GPUs the verification of software configuring and operating on these hardware components must be assured.

HARPO compiler consists of automated translators from HARPO program different variants. HARPO to Boogie translator generate a program in Boogie language(Boogie IR) to statically verify the HARPO program. Boogie Verifier generate the verification conditions (VCs) and pass on to *Z3* to determine their correctness. *Z3* generates the Error Report. The other three translators: HARPO to C, HARPO to CUDA, HARPO to VHDL, are intended to design generate the other programs in respectively mentioned languages. HARPO is future for design and verification of complex concurrent systems using uniprocessors FPGAs and GPUs.



Boogie Verifier

Several formal verification tools have been built and used for formal verification to ensure the correctness of software in different programming languages. The Boogie language is a common intermediate representation for static verification of programs written in several high-level programming languages. Boogie was developed by Hoare-Logic based program verifier. Boogie language is an intermediate verification language used as a common intermediate representation in the verification of other higher-level programming languages. Boogie can also be used as an input/output format for the abstract representation and predicate abstraction. Internally, Boogie performs a series of transformations of source program into verification conditions to error report. Boogie, previously known as BoogiePL, as a language has imperative and mathematical components. A challenge is to create verification conditions Boogie verifier uses SMT solvers, such as Z31, to determine the truth of verification conditions. In this paper, an eventful backend of a formal verification system called Boogie Intermediate Verification Language is discussed, and number of efficacious verifiers based on Boogie backend are reported as follows: Section II describes the Hoare-style automated verifier named Boogie and Boogie Verification Language. Later sections describe the verifiers based on Boogie.

Verifier Based on Boogie IVL

Dafny: The Dafny programming language is designed to write programs using built-in specification constructs. Dafny's treatment of locations is based on dynamic frame theory. Its compiler can produce executables for the .NET platform. Dafny attempts to determine the correctness of programs by checking the parts of the program for their own correctness and then infer the correctness of complete program based on smaller parts. The use of dynamic frames enables Dafny to prove the correctness when data-abstraction is used.

VCC: VCC (Verifier for Concurrent C) is layered on top of the C language for verification purposes. VCC was developed in Microsoft Hypervisor Verification Project (MHVP). The project was intended to provide verification of functional correctness properties of various software types, including commercial, system software, off-the-shelf software, and Microsoft Hyper-V.

Chalice: Chalice is an experimental language developed to allow the verification of concurrent programs. Language support various features, including dynamic objects and threads creation, locks, monitors, mutual exclusion, pre-conditions, and post-conditions. The language supports locking memory locations at very low-level using permissions. Chalice carries permissions and transfer of permissions approach to address the specification for verification of concurrent programs.

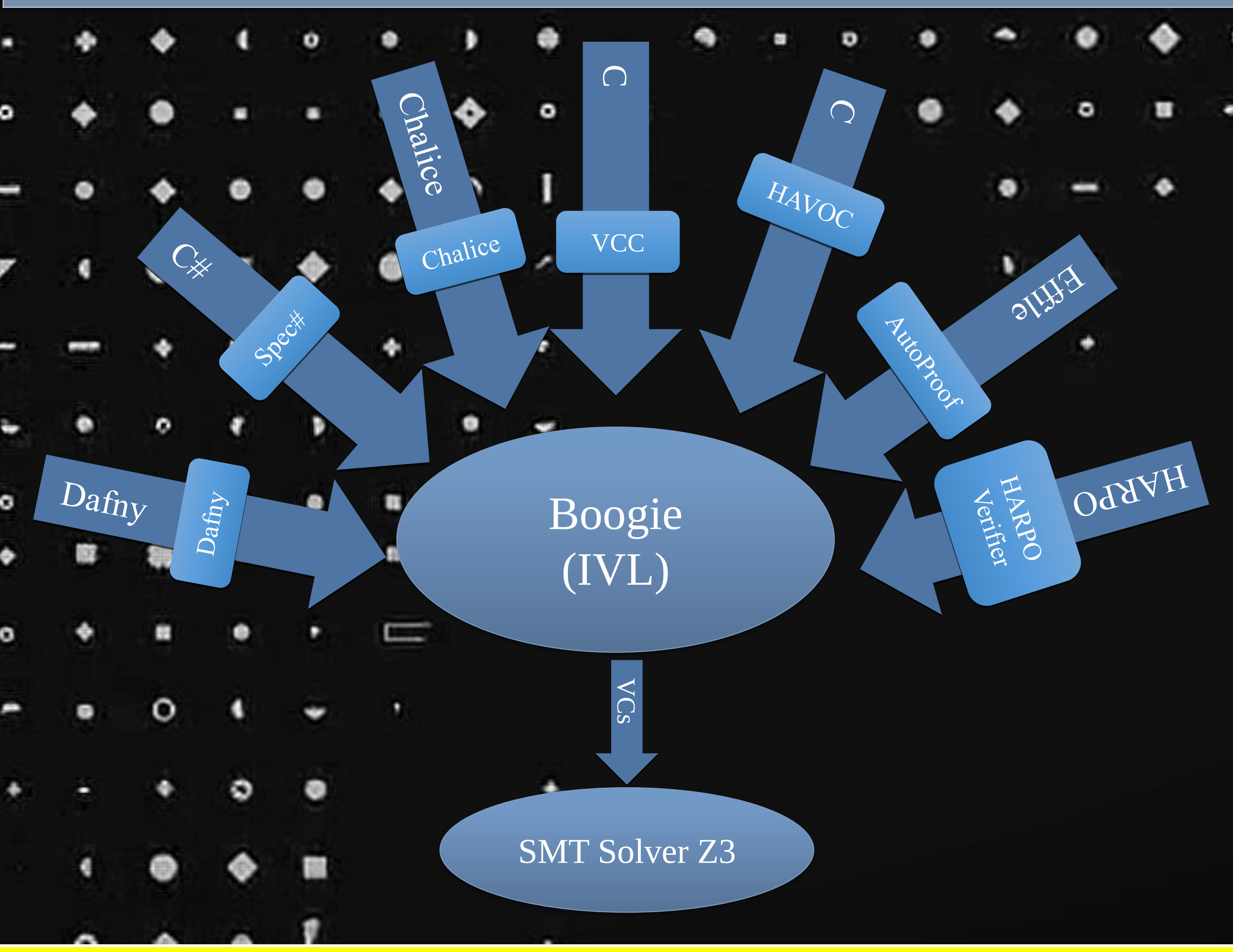
Verve: High-level computer applications are created on top of low-level operating systems and run-time language systems. The security and reliability of such lower-level software and system are critical. Errors can lead to system software crashes, data loss, and insecure hardware control. Verve is a formal verification system that uses Typed Assembly Language (TAL) with Hoare Logic to accomplish a highly automated verification. Verve primarily verifies the absence of many categories of errors in low-level code. Type and memory safety are verified with Verve, it has "Nucleus" which has access to hardware components such as memory. The "Nucleus" is implementation of memory allocation, garbage collection, interrupts and their handling, devices access control, and stacks. The kernel of OS is built above the "Nucleus" and applications run on top of kernel.

Spec#: Spec# is an object-oriented language designed by extending C# with specifications features. Spec#'s goal is to provide more cost-effective ways to produce high-quality software. The annotations allow the programmer to write specifications expressing the intention of programmer about data and methods being used. Spec# compiler performs run-time checks to assert the specifications.

HAVOC: HAVOC (Heap-Aware Verifier Of C) is a static verifier for C similar to ESC/Java, and Spec#, static verifiers for Java and C#, respectively. HAVOC is a distinguished verifier deals with lower-level details of C language. It provides the automatic update for reachability predicate which is explicitly designed to deal with pointers and their arithmetic operations. HAVOC addresses on of the sources of unscalability for automatic verifiers struggling with the imprecision caused by a heap allocated data structures. There are two fundamental correctness

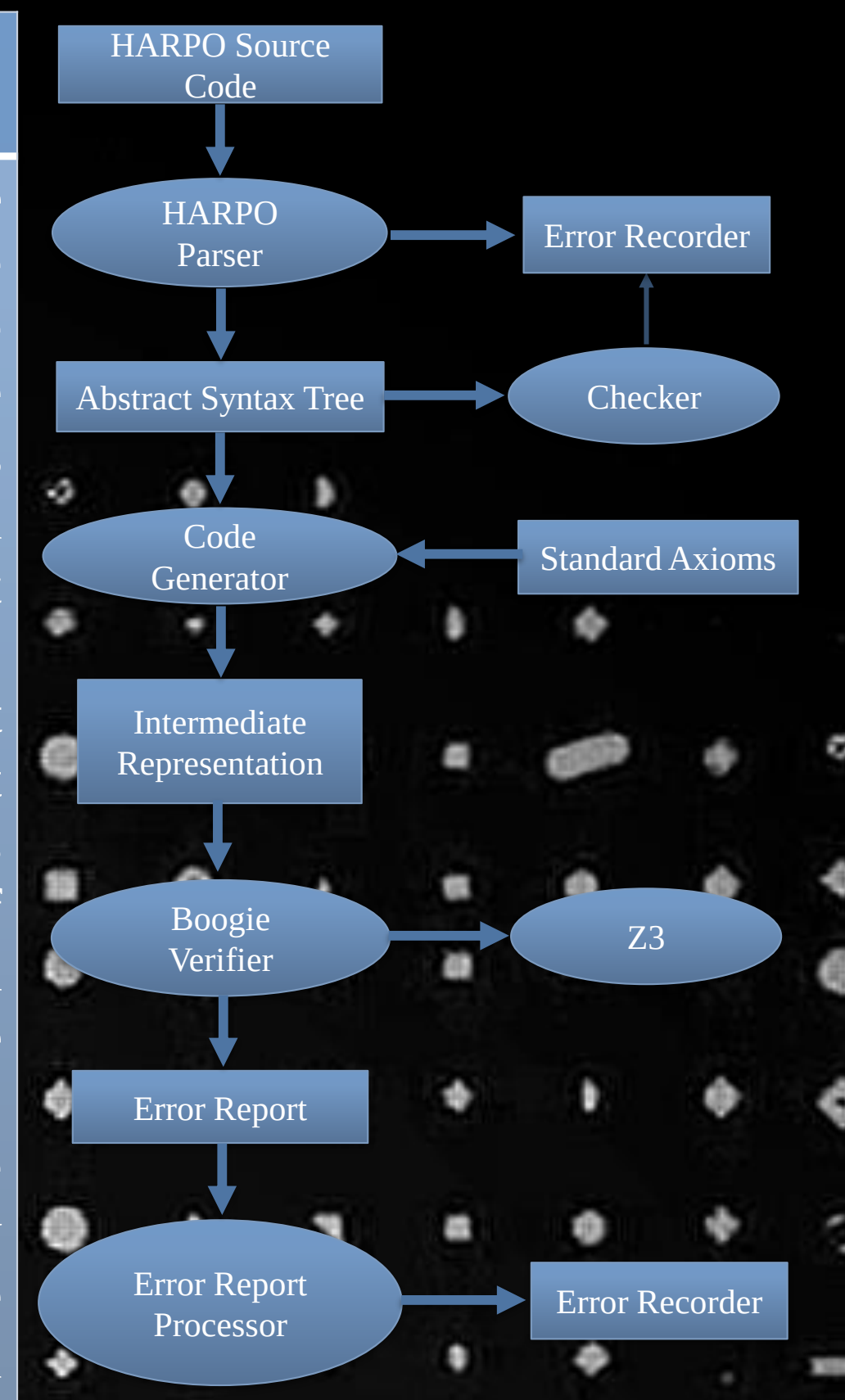
properties, control flow, and memory safety, which depend on the assertions pointing the contents of the heap data structure. These reachability predicates are required for specifying the properties of heap.

AutoProof: Programmers not comfortable with formal verification techniques experience difficulty while verifying the correctness of programs formally. Eiffel reduces the burden of writing enormous annotation specifications for program using Auto Proof approach. AutoProof is a static verifier for Eiffel programs and part of Eiffel Verification Environment (EVE) which encompasses numerous verification tool and utilize their synergy.



HARPO Verifier

HARPO Verifier is developed to verify the correctness of HARPO programs using the same methodology Listed Verifiers had been using to prove the correctness of their programming languages. The verification methodology of the HARPO verifier is significantly related to the previously mentioned verifiers. Since HARPO language is a concurrent language that is based of conditional critical sections and *rendezvous* between threads. It employs explicit transfer of permissions in order to verify concurrent programs. HARPO Verifier uses two different layers of operations while performing the verification of HARPO programs. Specifications are written in form of annotations in standard HARPO syntax. These annotations are ignored when compiling to C, VHDL, and CUDA. The HARPO Verifier transforms the annotated program into verification conditions by translating the program into Boogie. Later, Boogie Verifier transform the code into verification conditions and check for their correctness using SMT solver named *Z3*.



Conclusion

Some automated verifiers designed with on staging approach for generating verification conditions are reported and provide a motivation for HARPO Verifier. Each verifier targets different areas of software verification problems. However, all of them use the same underlying verifier for generating verification conditions and checking them with SMTs. Boogie is known and proved to be a beneficial resource for verification condition generation and checking.

We have tested the verifier on an integer counter design and number of other examples, as well as carefully unit testing each part of the system using HARPO Verifier.

Future Work

HARPO Verifier is addition to list of static verifiers targeting the verification of concurrent high-level programming language designed to target reconfigurable, GPUs, and microprocessors.

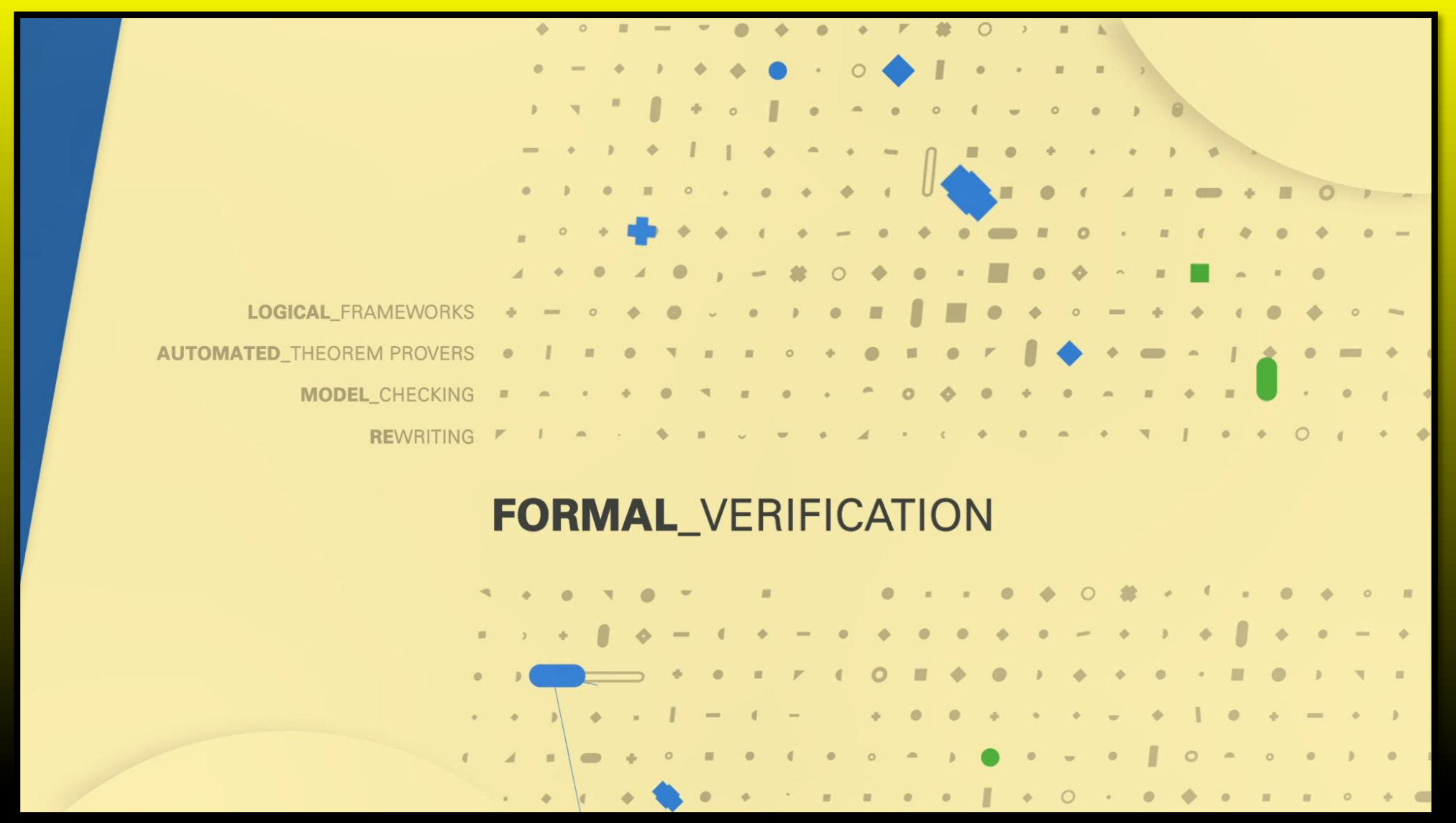
- Future work will include automatically inferring certain loop invariants so as to lessen the burden on the programmer in annotating their code.
- Developing user interface for HARPO Verifier.
- Develop the translation methodologies for VHDL and CUDA backend.

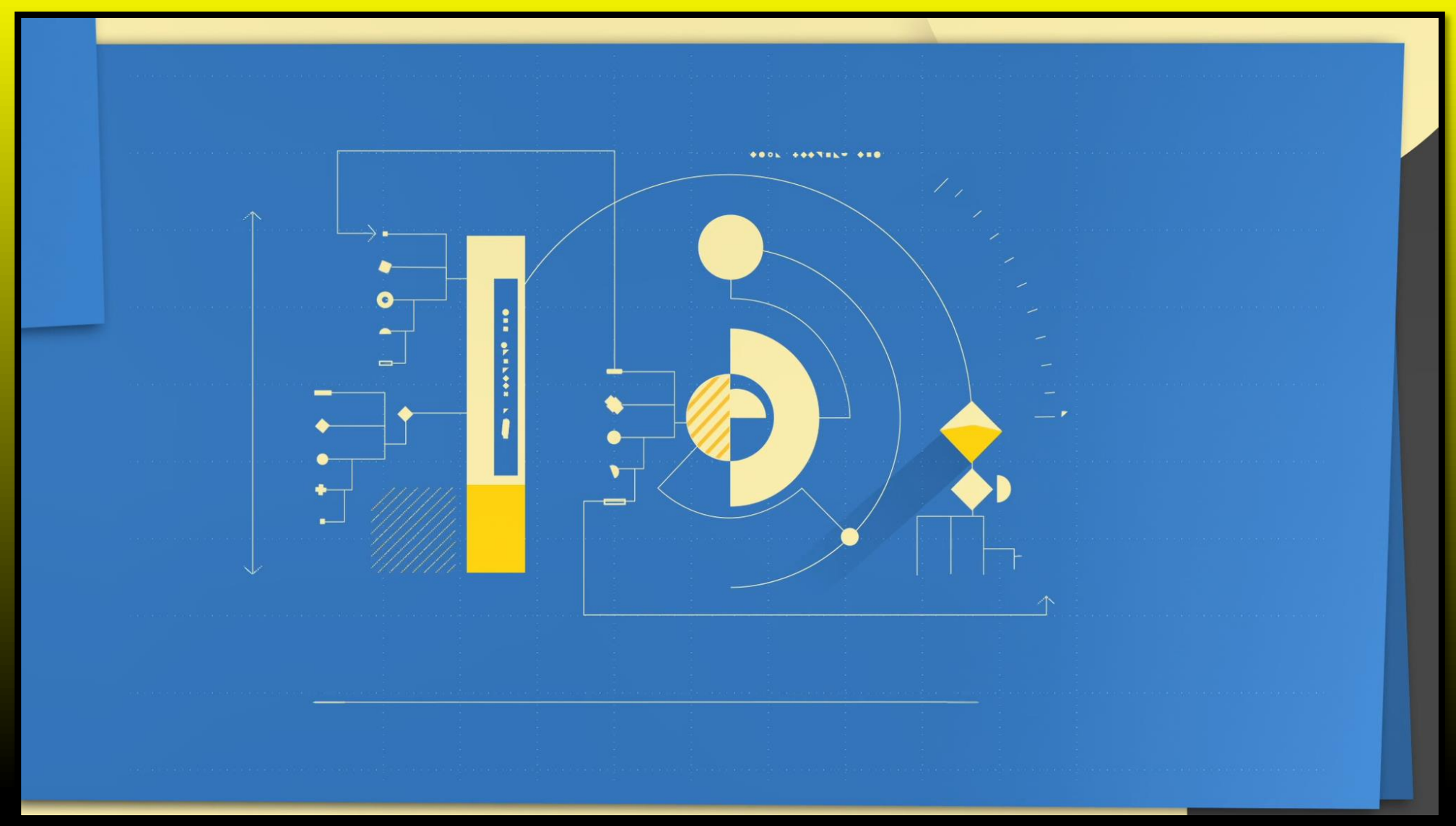
Acknowledgement

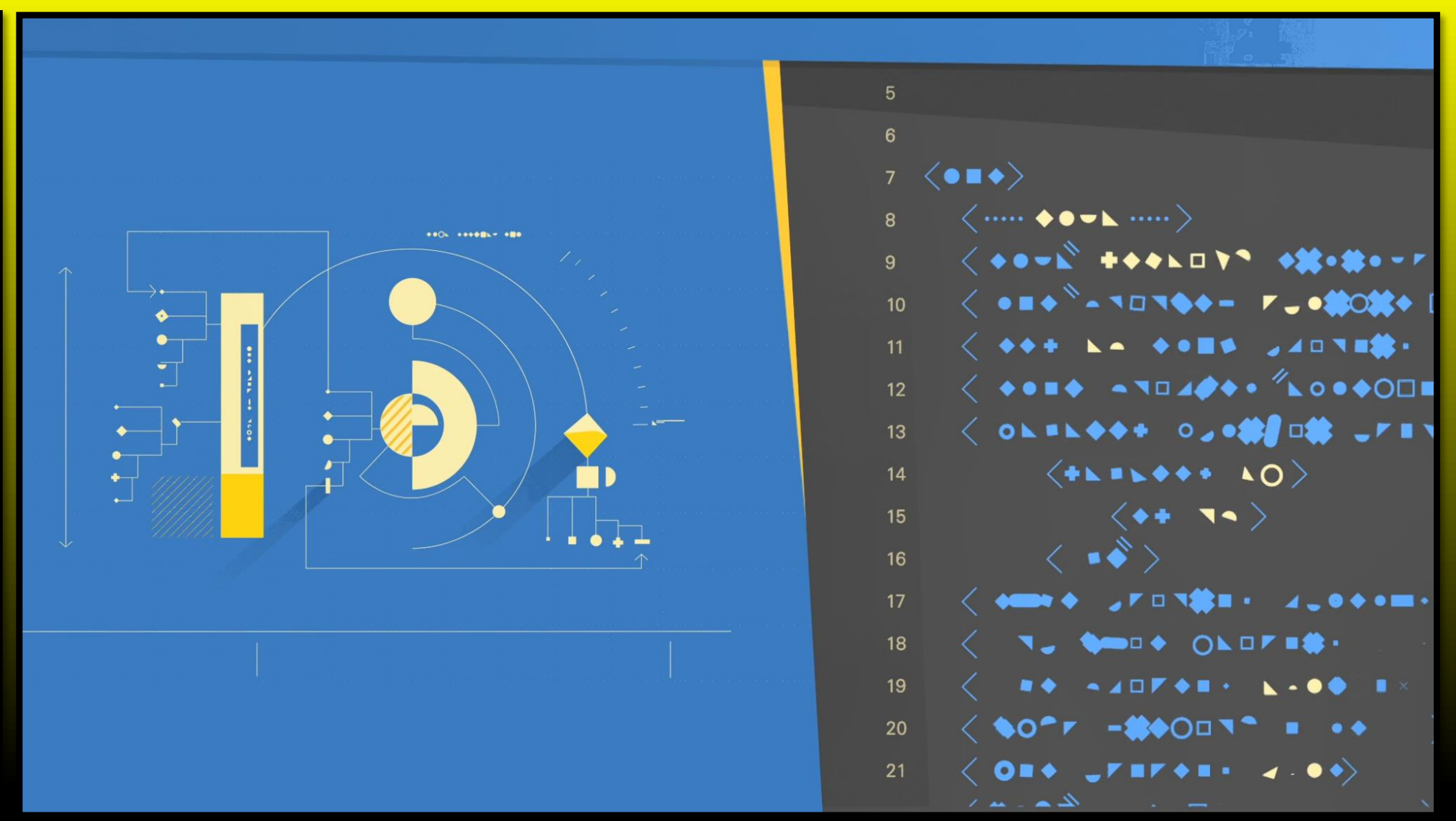
The research was supported in parts by funds from National Science and Engineering Research Council (NSERC) of Canada.

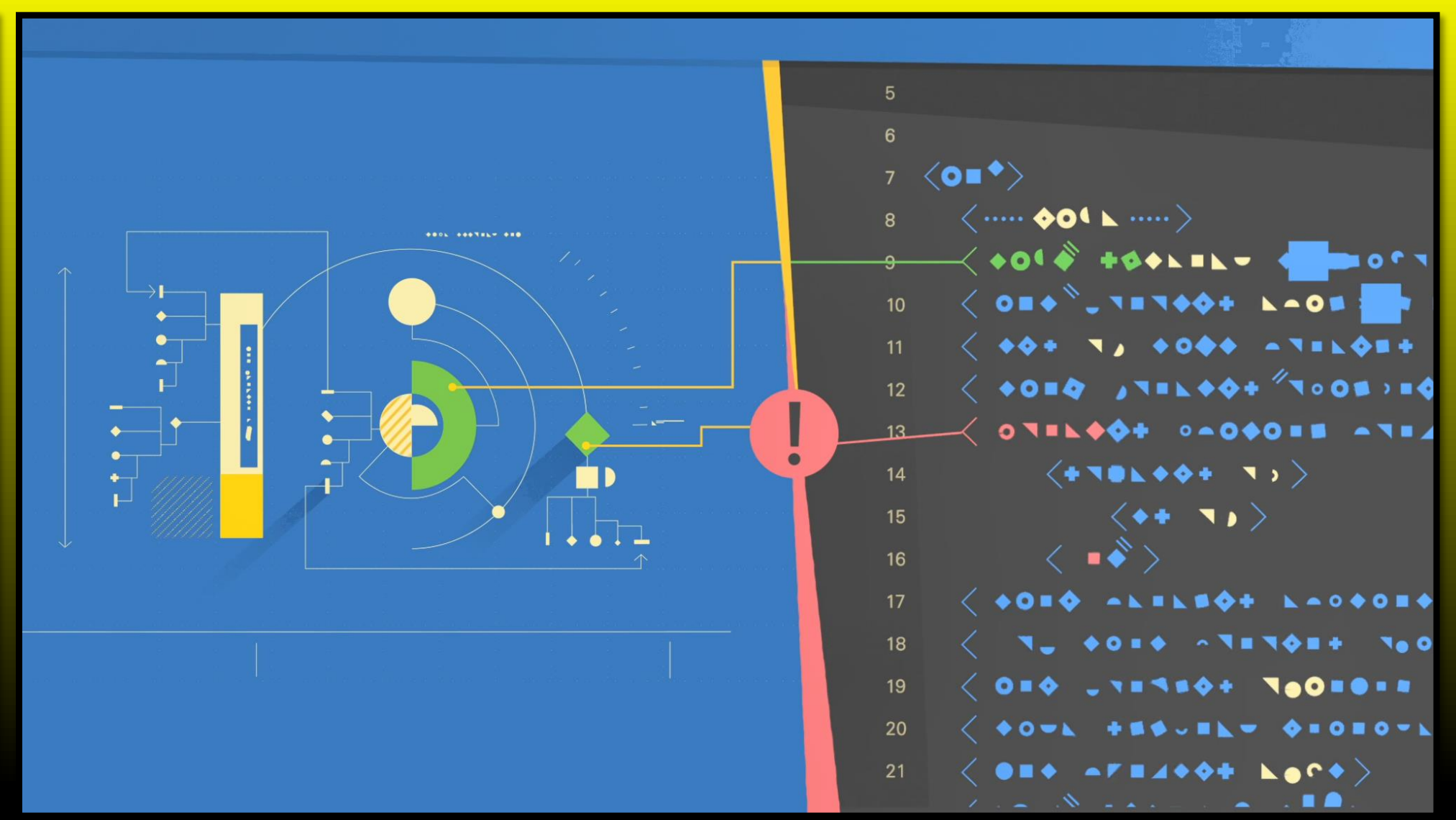
Reference

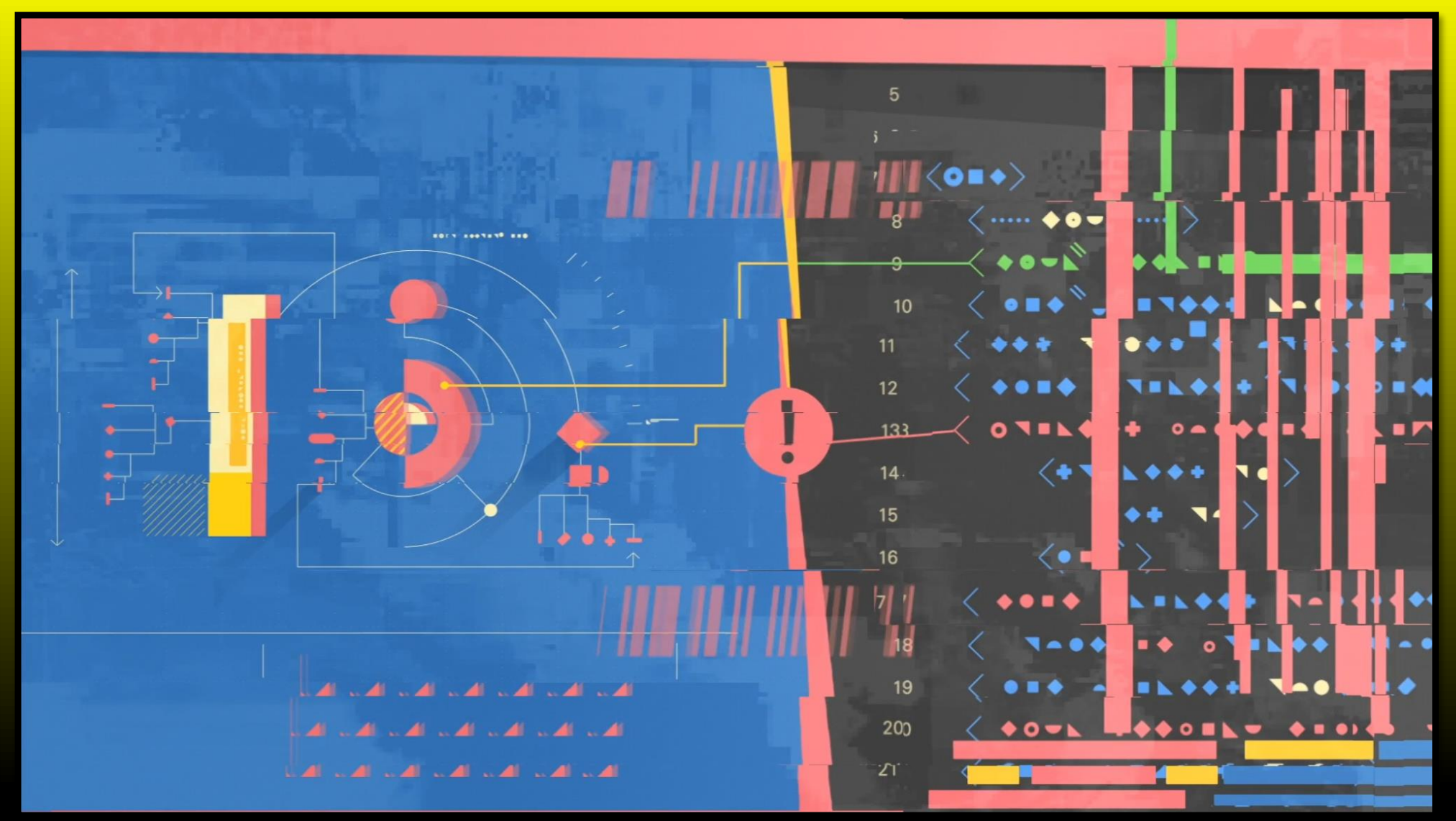
I. Ahmed, T.S. Norvell, R. Venkatesan, *"A Review of Formal Program Verification Tools based on Boogie Language"* in Newfoundland Electrical and Computer Engineering Conference (NECEC), 2019.
II. T.S. Norvell, A.T. Md.Ashraful, L.Xiangwen, & Z. Dianyong, HARPO/L:A language for hardware/software codesign. in Newfoundland Electrical and Computer Engineering Conference (NECEC), 2008.
III. I. Ahmed, T.S. Norvell, R. Venkatesan, "Verifying the correctness of HARPO Programs in Newfoundland Electrical and Computer Engineering Conference (NECEC), 2018.











Photos are courtesy of Galois, Inc.