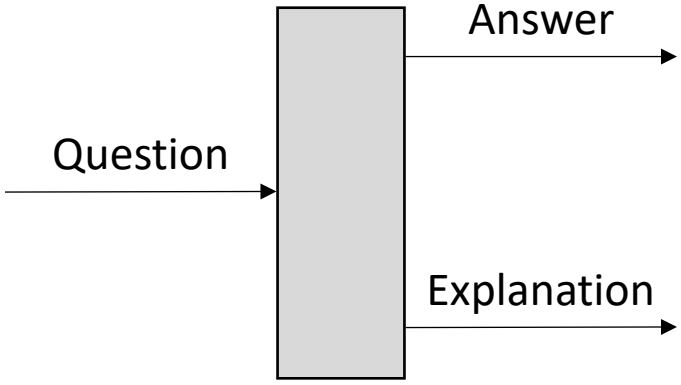
3rd International Workshop on Explainability of Real-time Systems and their Analysis

at the IEEE Real-Time Systems Symposium

York, UK, December 10, 2024



It is our pleasure to welcome you to the 3rd International Workshop on Explainability of Real-time Systems and their Analysis (ERSA). This workshop is held on December 10, 2024 in York, UK in conjunction with IEEE Real-Time Systems Symposium (RTSS). We started this workshop because we wanted to explore whether the notion of explainability is helpful for the real-time system research community in order to deliver more value to software practitioners—in particular those involved in certification. This document is the workshop proceeding for ERSA'24.

We thank several individuals and institutions without whom this workshop would not have been possible. This includes:

1. the authors of peer reviewed papers, keynotes, and panelists for providing technical content;
2. the members of the technical program committee for evaluating the peer-reviewed papers and providing constructive feedback to the authors;
3. the organizers of RTSS that gave "go-ahead" for ERSA to take place; this includes Zhishan Guo (Hot-Topics Day Chair of RTSS);
4. the University of York for supporting ERSA;
5. Iain Bate for being willing to help on short notice with the organization;
6. people working behind the scene to provide (digital and physical) infrastructure, advice, and proofreading.

The papers in this workshop proceeding provide new ideas on explainability. This year, we received papers from authors who have not published at ERSA before. This is a testament to the change of the area and our community. We believe and hope you will find the papers interesting; and that they will help you and help us all in defining this new area of research.

Sincerely,

Philippa Ryan
Bjorn Andersson
Co-Chairs of ERSA'24

Program Co-Chairs

Bjorn Andersson, SEI/CMU, USA
Philippa Ryan, UYork, UK

Program Committee

Ademola (Peter) Adejokun, Lockheed Martin, USA
Ahlem Mifdaoui, UToulouse, FR
Al Mok, Texas, USA
Andrew Banks, LDRA, UK
C. Michael Holloway, NASA, USA
Chi-Sheng (Daniel) Shih, NTU, TW
Dionisio de Niz, SEI/CMU, USA
Elena Troubitsyna, KTH, SE
Ganesh Pai, KBR/NASA ARC, USA
George Romanski, FAA, USA
Guillem Bernat, Rapita, UK
Hyoseung Kim, CR, USA
Isaac Amundson, Collins Aerospace, USA
Jie Zou, UYork, UK
John Lehoczky, CMU, USA
Mallory Graydon, NASA, USA
Mark Klein, SEI/CMU, USA
Rafael Zalman, Infineon, DE
Shige Wang, Motional, USA

Sound WCET Analysis, Explanation of the Method and of the Results

Reinhard Wilhelm
Informatik
Saarland University
Saarbrücken, Germany
ORCID ID 0000-0002-5599-7560

Jan Reineke
Informatik
Saarland University
Saarbrücken, Germany
ORCID ID 0000-0002-3459-2214

***Abstract*—Sound WCET analysis computes reliable upper bounds to all execution times of a program as required by a schedulability analysis. It is a complex method, composed of many component methods. Software developers and certification authorities are interested in understanding the derivation of the results in order to develop trust in their correctness. The results cannot be understood without a basic understanding of the methods. We argue that essentially one of the component methods, Microarchitectural Analysis, is critical for understanding and accepting the results. In this article we therefore explain Microarchitectural Analysis and show how its provides local explanations for the overall result.**

In addition, we show that progress monotonicity is a sufficient condition for timing compositionality and that it increases explainability.

***Index Terms*—WCET analysis, real-time, instruction execution times, timing compositionality**

I. Introduction

Sound WCET analysis computes reliable upper bounds to all execution times of a program. The problems of WCET analysis are caused by performance-enhancing features of microarchitectures. They introduce a large variability of execution times of instructions. This article explains the principle behind sound solutions of the WCET problem. This principle is to prove the absence of *timing accidents*, i.e., events during the execution of an instruction execution that increase the execution time compared to the fastest execution. These proofs are based on the determination of invariants about the set of potential execution states at each program point. Such invariants can be computed by a fixed-point iteration by Abstract Interpretation. The particular ingredients of the employed instances of abstract interpretations are explained and connected to fundamental insights into the timing-predictability of execution platforms. We show how the computed invariants are essential for explaining the results of WCET analysis.

II. WCET Analysis—the Problem

WCET analysis can be seen as the search for the longest path through the control-flow graph of a program. The nodes are the instructions of the program, annotated with their execution times. This task was relatively easy in the (good old) times of instructions with constant execution times [1], [2]. Structural induction over the structure of a program was used to compute global upper bounds on all execution times of instructions.

Unfortunately, in modern high-performance processors the execution times of instructions vary widely. This is caused by their dependence on the execution state of performance-enhancing features such as caches, pipelines, and all kinds of speculation. The core of the WCET–analysis problem for modern high-performance processors thus is, how to safely bound the execution times of individual instructions in a program. This is done by the analysis component called *Microarchitectural Analysis* in the picture of the tool structure in Fig. 1. The determination of a safe upper bound on the execution time of the whole program and of the path on which this bound is determined is called *Global Bounds Analysis* in Fig. 1. This analysis explores all paths in a state space, spanned by the program and the architecture. It would be desirable to cut down the size of this space. However, as we will see later, a ghost that haunts WCET researchers is the existence of *Timing Anomalies* [3], [4], namely that cheaper continuations of execution paths may lead to globally more expensive paths and vice versa. For microarchitectures exhibiting timing anomalies, it would be incorrect to explore only worst-case transitions. The full graph needs to be explored. We will come back to timing anomalies in the context of timing predictability.

III. WCET-Analysis—A Solution

The different execution times result from different *execution states* in which an instruction may be executed. A memory access is fast if the accessed memory block is in the cache, it is slow if it has to be fetched from memory, and it is even slower if the memory load is blocked on the bus. We call cache misses, pipeline stalls, bus collisions, and mis-speculations *Timing Accidents* and the associated extra cycles *Timing Penalties*. The search through the annotated graph could safely assume a cache hit if it were known, for example as results of a static cache analysis, that the accessed memory block were in the cache each time execution reaches that program point. So, the solution consists in computing invariants at each program point that safely describe the execution states, i.e., the occupancy of the machine resources.

We now argue that all but one component analysis from Fig. 1 are nor critical for understanding sound WCET analysis.

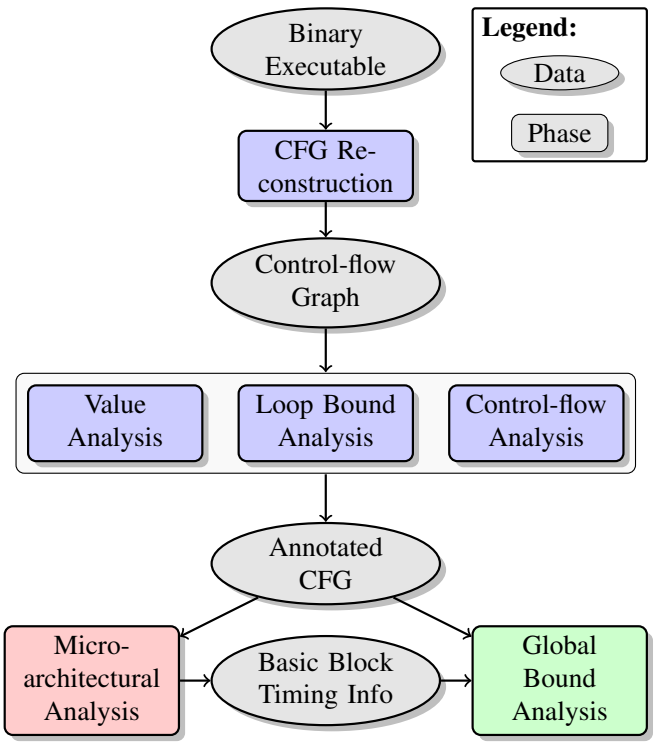


Fig. 1. Architecture of WCET tools using static analysis based on an abstraction of the execution platform such as AbsInt's aiT tool [5].

The *Value Analysis*, shown in Fig. 1, computes enclosing intervals for the values in program variables and machine registers. The results restrict potential memory accesses to enable a precise data-cache analysis and support *Loop Bounds Analysis*. The *Control-Flow Analysis* determines facts about the control flow, like infeasible control flow paths, that cannot contribute to overall execution times. In case of doubt these computed approximations can be inspected and compared with the user's expectations. The *Global Bounds Analysis* exhibits the control-flow paths through the program, augmented by the paths through the architecture, together with annotations of the costs. The overall graph shows all cycle-wise evolutions of the computation. The direct or indirect algorithm, used to compute this graph, can be taken from textbooks and proved correct. In case of doubt the annotated graph can be manually checked.

The *Microarchitectural Analysis* in this solution to the WCET-Analysis problem [6] consists in determining at each program point an invariant that describes a safe approximation of the set of execution states that are possible when execution reaches this program point. Based on these invariants our WCET analysis proves safety properties of the following kind: *A certain timing accident cannot happen when the instruction at this program point is executed.* These approximations are *safe* in the sense that any timing accident excluded based on them will indeed never happen. Each safety property proved in this way allows WCET analysis to reduce the execution-time bound of the instruction by the associated timing penalty.

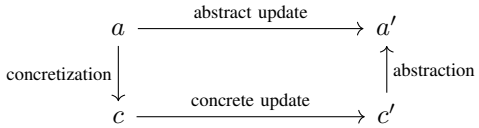
The invariants are computed by an instance of *Abstract Interpretation* [7], algorithmically by a fixed-point iteration over the control-flow graph of the program. Abstract interpretation is a semantics-based static analysis. Unlike most of the instances of abstract interpretations used in practice or published, abstraction interpretation used in WCET analysis needs to include the semantics of the underlying execution platform. Its core, therefore, is an abstraction of the underlying microarchitecture. This abstraction is structured into different components according to the structure of the architecture. Abstractions of several architectural components are described in Subsection III-A and Section IV.

A. Execution-state Invariants as Means to Explain the Result of WCET-Analysis

The execution-state invariants and the transitions between invariants at consecutive program points are the means to show local correctness to a user or certification authorities. Execution-state invariants describe possible occupancies of machine resources, i.e. what is in the caches, how far have instructions progressed through the pipeline, and which conflicts may they encounter in attempting to access pipeline units, which part of the bus bandwidth is occupied by running transfers. Microarchitectural execution states are structured as a collection of components according to the overall structure of the machine resources. The occupancy of different components is represented in different abstract domains. Understanding the invariants means being able to read the abstract states.

In Abstract–Interpretation terminology, *concretization* is the function that returns the set of concrete states represented by an abstract state. The abstract cache states, for instance, describe sets of concrete cache contents, together with replacement information. In the case of an LRU cache the latter are upper bounds on the ages of the memory blocks guaranteed to be in the cache. In the case of non–LRU caches this information may be complex. Abstract pipeline states of most current architectures are easier to understand since they are collections of concrete pipeline states.

In addition to the concretization of the abstract cache states, the transitions between invariants need to be explained. In principle, one could imagine that all component states described in an invariant would be concretized, then the concrete transformation would be applied to all of the concrete ones, and then the resulting set of concrete states would be abstracted back into abstract states forming the new invariant as illustrated by the following diagram:



More realistically, the developer of the microarchitectural analysis determines a conservative approximation of the composition of these three functions and explains the result to the interested public. The abstract update needs to satisfy the following *local correctness condition*: Given a set of

concrete caches states c described by abstract cache state a. The transition from all the cache states in c yields concrete cache states collected in a set c'. The application of the abstract transition relation to a must yield an abstract cache state a' that contains at least all the concrete cache states in c'. This is captured by the following diagram:

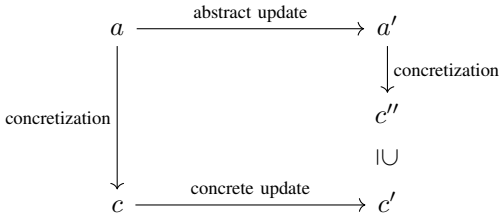


Fig. 2 shows an example of an abstract transition in the case of caches. This figure should be read as follows: Transitions from one *concrete cache state* to a successor state happen under a memory access. In the lower part of Fig. 2 transitions from four concrete cache states under an access to memory block C lead to two potential successor states. The initial four concrete cache states are described by (can be abstracted to) the *abstract cache state* pictured above them. This is an abstract cache state that contains all memory blocks contained in all of these concrete cache states with an age at least as large as that of all their ages in the concrete cache states. The transition from the abstract cache state leads to an abstract cache state that describes the two resulting concrete cache states in the same way. We also say that the concretization of the resulting abstract cache state contains the two concrete cache states.

In this particular case, the abstraction is *exact*, i.e., the resulting abstract cache states represent the two potential successor states, and no more. In general, the cache abstraction from the example, and other abstractions, may introduce additional spurious concrete states, which may result in a loss of precision, but correctness remains guaranteed by overapproximating any set of reachable concrete states.

The *local correctness condition* we have just explained is the basis to explain the *global correctness* of the analysis. If each abstract transition is locally correct, then it can be shown that the concretization of the reachable set of abstract states computed by fixed-point iteration is guaranteed to contain all reachable concrete states.

Again, abstract transitions on non–LRU abstract cache states may be complex. However, there is no way to give a simple explanation of a complex mechanism.

IV. Microarchitectural Abstractions

Attempting to determine the real worst-case execution time by exhaustive exploration of the space of all paths is a hopeless endeavor for all but trivial programs. Not only the space of all paths through the control-flow graphs needs to be explored, but also the much larger space of paths through the sets of execution states of the underlying microarchitecture.

This latter space can be reduced by abstraction, considering *abstract execution states* instead of concrete execution states.

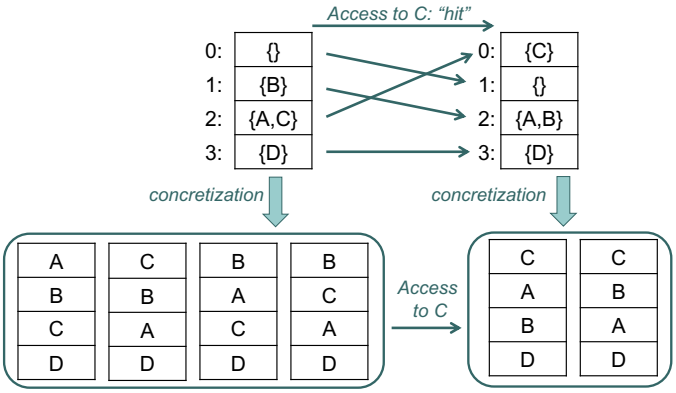


Fig. 2. Transition from one abstract LRU cache state to a successor abstract cache state and their concretizations

The different components of the execution platform are abstracted as to make the space exploration feasible, but on the other hand keep enough information for the above mentioned exclusion of timing accidents. It turns out that different types of components need different types of abstractions.

A. Abstractions of State–Dependent Resources

The essential property of *state-dependent resources* as far as timing of instruction execution is concerned is that their state influences instruction execution times, that is, the execution of instructions takes different number of cycles depending on the state of the resource. For example, an instruction or an operand fetch takes different number of cycles depending on the state of the instruction or data cache. The access to a memory bank takes a different number of cycles depending on whether the bank is open or closed. The next instruction to be fetched depends on the state of the branch predictor. Static cache analysis has been the mother of all WCET analyses. Compact abstractions of cache states with efficient updates, such as the one shown in Fig. IV, have been identified for caches with LRU-replacement policy [8].

B. Abstractions of Bandwidth Resources

We call microarchitectural components such as buses and other interconnects *bandwidth resources* since they offer limited resources in time to different competing actors. Contention is resolved based on some protocol, sometimes based on the state of the resource. In contrast to storage resources, where the actual state of a resource can still be influenced by long-passed state changes, this state usually results from recent actions, i.e., whether two cores are currently competing for access or not. Abstract states in the static analysis of bandwidth resources record everything needed to predict guaranteed access to the resource, such as all potential states of the access protocol, all maximal delay times caused by running accesses, and the minimal available bandwidth of the resources.

Bandwidth resources are particularly challenging when they are shared, as e.g. buses are in multi-core processors, due to the large number of possible interactions between processes running on different cores. A promising approach to efficiently

analyze shared bandwidth resources is compositional analysis, which we discuss in the next section.

C. Abstractions of Progress Resources

Pipelining overlaps the execution of multiple instructions to improve performance. From an analysis point of view pipelining is challenging as it prohibits decomposing the problem into the analysis of one instruction at a time. Flowing through the pipeline, each instruction gradually progresses through the resource. We thus regard pipelines as *progress resources*. Abstract pipeline states in static pipeline analysis record the minimum progress instructions under execution can be guaranteed to have made at a program point.

Efficient abstractions for pipeline states have been particularly hard to find. Out-of-order pipelines required using expensive power-set domains [9], [10]. This means that the pipeline analysis computed (often very large) sets of pipeline states at each program point instead of small descriptions of such sets like in the case of caches. Out-of-order pipelines also unavoidably came with timing anomalies, which we will discuss further in the following section. In contrast to common belief, even in-order pipelines can exhibit timing anomalies as shown in [11].

V. Timing Predictability

The notion of *Timing Predictability* has been around even before the WCET problem became exciting [12]. Varying instruction execution times were not a problem at this time. The method described in this paper was first instantiated for two different microarchitectures used by Airbus [6]. It became immediately clear that these two microarchitectures, a Motorola Coldfire and a PowerPC 655 had different *Timing Predictabilities* [13].

Thiele and Wilhelm [14] published a first recommendation for the design of timing-predictable microarchitectures based on empirical evidence. The dissertation of Jan Reineke was the first to present a formal notion of timing predictability, namely that of cache replacement strategies [15], [16].

A. Timing Anomalies and Timing Compositionality

Microarchitectures exhibiting timing anomalies force sound WCET analyses to explore inherently larger search spaces. On the other hand, microarchitectures that are provably free from anomalies lead to more scalable WCET analyses and more explainable results, as the analysis and its explanation can focus on a smaller set of states. A natural question thus is how to systematically construct microarchitectures that are free from timing anomalies.

Similarly, multi-core platforms with shared resources offer a severe challenge for WCET analysis since different interferences on the shared resources, in general, lead to different timing behaviors. Explicitly exploring all interleavings of accesses to shared resources leads to an enormous increase in the size of the search space [17], [18]. A promising approach to scalably analyze WCET in multi-core systems is compositional analysis [19], [20], where the timing contributions of shared resources are analyzed separately and then composed. In such an approach, the "base" WCET of each task is analyzed assuming the task is run in isolation. Then, the amount of interference on each shared resource is bounded separately and added to the base WCET to obtain a bound on the execution time under contention. We argue that compositional analyses are naturally more explainable than integrated analyses as each individual analysis is less complex. Unfortunately, such an approach requires timing compositionality [21], the ability to soundly compose contributions from multiple resources. Complex processors have been shown to violate timing compositionality.

It turns out that timing compositionality and freedom from timing anomalies are strongly related. In fact, Hahn and Reineke [22] showed that *progress monotonicity* is a sufficient condition for both properties. More precisely, consider two microarchitectural states a and b in which every instruction exhibits more progress towards retirement in b than it does in a, we say that $a \leq b$. A microarchitecture exhibits *progress monotonicity* if for any pair of states $a \leq b$, the immediate successor states $s(a)$ and $s(b)$ maintain the ordering, i.e., $s(a) \leq s(b)$. In [22] they also demonstrate how to construct a processor that provably satisfies progress monotonicity and thus facilitates fast and explainable WCET analysis.

Computer architects are mainly concerned with increasing the average-case performance and ignore the requirements of the embedded real-time domain. The most notable exception is the Kalray many-core processor which is designed to avoid interferences and make sound and precise WCET analysis possible [23].

VI. Explainability Challenges

The correctness of the architectural abstraction is the critical point for customers and certification authorities when it comes to the soundness of a WCET–analysis tool. Formal verification of the soundness is infeasible in the absence of formal models of the underlying execution platforms. Sophisticated testing strategies have been used to convince one of the correctness [24].

The ideal would be a formal or at least semi–formal derivation of the abstract model from an available architecture description in a HW description language like VHDL or Verilog. Marc Schlickling and Markus Pister [25] have attempted this approach, but were bogged down with too many detail problems as tool support for analysis and verification of HW description languages was under–developed at the time of their attempt. The growing ecosystem of RISC-V [26] open-source cores and systems-on-chip along with open-source synthesis toolchains such as Yosys [27] make such an approach appear more viable today.

In this context, it would even be conceivable to generate two tools from the Verilog code describing the underlying hardware:

1) An abstract hardware model used within WCET analysis. Based on this model the WCET analysis would

generate "WCET certificates" explaining the WCET bound.

2) A WCET certificate checker relying purely on the concrete hardware model to confirm the correctness of the WCET analysis result. This would reduce the trusted compute base by eliminating the need to trust the correctness of the abstract model.

ACKNOWLEDGEMENTS

We appreciate the comments of the reviewers who helped us to improve the paper. This work has received funding from the European Research Council under the European Union's Horizon 2020 research and innovation programme (grant agreement No. 101020415).

REFERENCES

[1] A. C. Shaw, "Reasoning about time in higher-level language software," *IEEE Trans. Software Eng.*, vol. 15, no. 7, pp. 875–889, 1989.

[2] P. P. Puschner and C. Koza, "Calculating the maximum execution time of real-time programs," *Real Time Syst.*, vol. 1, no. 2, pp. 159–176, 1989.

[3] T. Lundqvist and P. Stenström, "Timing anomalies in dynamically scheduled microprocessors," in *RTSS*, pp. 12–21, 1999.

[4] J. Reineke, B. Wachter, S. Thesing, R. Wilhelm, I. Polian, J. Eisinger, and B. Becker, "A definition and classification of timing anomalies," in *Proceedings of 6th International Workshop on Worst-Case Execution Time (WCET) Analysis*, July 2006.

[5] AbsInt Angewandte Informatik GmbH, "aiT WCET Analyzers." https://www.absint.com/ait/.

[6] C. Ferdinand, R. Heckmann, M. Langenbach, F. Martin, M. Schmidt, H. Theiling, S. Thesing, and R. Wilhelm, "Reliable and precise WCET determination for a real-life processor," in *EMSOFT*, vol. 2211 of *LNCS*, pp. 469 – 485, 2001.

[7] P. Cousot and R. Cousot, "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *POPL '77: Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, (New York, NY, USA), pp. 238–252, ACM Press, 1977.

[8] C. Ferdinand and R. Wilhelm, "Efficient and precise cache behavior prediction for real-time systems.," *Real-Time Systems*, vol. 17, no. 2-3, pp. 131–181, 1999.

[9] S. Thesing, *Safe and Precise WCET Determinations by Abstract Interpretation of Pipeline Models*. PhD thesis, Saarland University, 2004.

[10] M. Langenbach, S. Thesing, and R. Heckmann, "Pipeline modeling for timing analysis," in *Static Analysis, 9th International Symposium, SAS 2002, Madrid, Spain, September 17-20, 2002, Proceedings* (M. V. Hermenegildo and G. Puebla, eds.), vol. 2477 of *Lecture Notes in Computer Science*, pp. 294–309, Springer, 2002.

[11] S. Hahn, J. Reineke, and R. Wilhelm, "Toward compact abstractions for processor pipelines," in *Correct System Design - Symposium in Honor of Ernst-Rüdiger Olderog on the Occasion of His 60th Birthday, Oldenburg, Germany, September 8-9, 2015. Proceedings*, pp. 205–220, 2015.

[12] J. A. Stankovic and K. Ramamritham, "Editorial: What is predictability for real-time systems?," *Real Time Syst.*, vol. 2, no. 4, pp. 247–254, 1990.

[13] R. Heckmann, M. Langenbach, S. Thesing, and R. Wilhelm, "The influence of processor architecture on the design and the results of WCET tools," *IEEE Proceedings on Real-Time Systems*, vol. 91, no. 7, pp. 1038–1054, 2003.

[14] L. Thiele and R. Wilhelm, "Design for timing predictability," *Real-Time Systems*, vol. 28, no. 2-3, pp. 157–177, 2004.

[15] J. Reineke, *Caches in WCET Analysis: Predictability - Competitiveness - Sensitivity*. PhD thesis, Saarland University, 2009.

[16] J. Reineke, D. Grund, C. Berg, and R. Wilhelm, "Timing predictability of cache replacement policies," *Real-Time Systems*, vol. 37, no. 2, pp. 99–122, 2007.

[17] A. Abel, F. Benz, J. Doerfert, B. Dörr, S. Hahn, F. Haupenthal, M. Jacobs, A. H. Moin, J. Reineke, B. Schommer, and R. Wilhelm, "Impact of resource sharing on performance and performance prediction: A survey," in *CONCUR 2013 - Concurrency Theory - 24th International Conference, CONCUR 2013, Buenos Aires, Argentina, August 27-30, 2013. Proceedings* (P. R. D'Argenio and H. C. Melgratti, eds.), vol. 8052 of *Lecture Notes in Computer Science*, pp. 25–43, Springer, 2013.

[18] T. Kelter and P. Marwedel, "Parallelism analysis: Precise WCET values for complex multi-core systems," *Sci. Comput. Program.*, vol. 133, pp. 175–193, 2017.

[19] S. Altmeyer, R. I. Davis, L. S. Indrusiak, C. Maiza, V. Nélis, and J. Reineke, "A generic and compositional framework for multicore response time analysis," in *RTNS*, pp. 129–138, 2015.

[20] R. I. Davis, S. Altmeyer, L. S. Indrusiak, C. Maiza, V. Nélis, and J. Reineke, "An extensible framework for multicore response time analysis," *Real Time Syst.*, vol. 54, no. 3, pp. 607–661, 2018.

[21] S. Hahn, J. Reineke, and R. Wilhelm, "Towards compositionality in execution time analysis: definition and challenges," *SIGBED Rev.*, vol. 12, no. 1, pp. 28–36, 2015.

[22] S. Hahn and J. Reineke, "Design and analysis of SIC: a provably timing-predictable pipelined processor core," *Real Time Syst.*, vol. 56, no. 2, pp. 207–245, 2020.

[23] B. D. de Dinechin, D. van Amstel, M. Poulhiès, and G. Lager, "Time-critical computing on a single-chip massively parallel processor," in *Design, Automation & Test in Europe Conference & Exhibition, DATE 2014, Dresden, Germany, March 24-28, 2014* (G. Fettweis and W. Nebel, eds.), pp. 1–6, European Design and Automation Association, 2014.

[24] R. Wilhelm, M. Pister, G. Gebhard, and D. Kästner, "Testing implementation soundness of a WCET analysis tool," in *A Journey of Embedded and Cyber-Physical Systems - Essays Dedicated to Peter Marwedel on the Occasion of His 70th Birthday* (J. Chen, ed.), pp. 5–17, Springer, 2021.

[25] M. Schlickling and M. Pister, "Semi-automatic derivation of timing models for WCET analysis," in *Proceedings of the ACM SIGPLAN/SIGBED 2010 conference on Languages, compilers, and tools for embedded systems, LCTES 2010, Stockholm, Sweden, April 13-15, 2010* (J. Lee and B. R. Childers, eds.), pp. 67–76, ACM, 2010.

[26] K. Asanović and D. A. Patterson, "Instruction sets should be free: The case for RISC-V," Tech. Rep. UCB/EECS-2014-146, Aug 2014.

[27] C. Wolf, "Yosys open synthesis suite." https://yosyshq.net/yosys/.

Towards Explainable Compositional Reasoning

Isaac Amundson, Amer Tahat, David Hardin, and Darren Cofer
Applied Research and Technology, Collins Aerospace, USA
{first.last}@collins.com

***Abstract*—Formal verification tools such as model checkers have been around for decades. Unfortunately, despite their ability to prove that mission-critical properties are satisfied in both design and implementation, the aerospace and defense industry is still not seeing widespread adoption of these powerful technologies. Among the various reasons for slow uptake, difficulty in understanding analysis results (i.e., counterexamples) tops the list of multiple surveys. In previous work, our team developed AGREE, an assume-guarantee compositional reasoning tool for architecture models. Like many other model checkers, AGREE generates potentially large counterexamples in a tabular format containing variable values at each time step of program execution up to the property violation, which can be difficult to interpret, especially for novice formal methods users. In this paper, we present our approach for achieving *explainable* compositional reasoning using AGREE in combination with generative AI. Our preliminary results indicate this technique works surprisingly well, and have encouraged us to expand this approach to other areas in explainable proof engineering.**

I. Introduction

Formal methods provides a mathematically rigorous means of verification that one would expect for the development of high-assurance systems such as those in the aerospace and defense industries. Certification guidance has even been published on how formal methods can be used to satisfy airworthiness objectives for airborne software in commercial aircraft [1]. However, despite the effectiveness of these powerful proof techniques, their adoption into traditional development processes has been slow and uneven. Reasons for slow uptake include scalability limitations of the underlying algorithms, poorly designed user interfaces and other tool usability factors, and the need for formal training to properly use them [2].

The DARPA Pipelined Reasoning of Verifiers Enabling Robust Systems (PROVERS) program was recently launched with the goal of producing scalable and usable formal methods tools that can be integrated into traditional aerospace and defense development processes. Specifically, a key outcome of the program is that product engineers with minimal formal methods background will be able to benefit from these powerful technologies, further driving their adoption while simultaneously improving product dependability.

To address these challenges, our team is developing the Industrial-Scale Proof Engineering for Critical Trustworthy Applications (INSPECTA) framework[1]. INSPECTA consists of *ProofOps* and *BuildOps* tools and methods that integrate with current aerospace DevOps pipelines and achieve provably correct design and implementation at each level of the system hierarchy. In order to address the key objectives of PROVERS, we pay particular attention to addressing scalability and explainability concerns with respect to the proof tools in our framework.

Within the *ProofOps* workflow, INSPECTA uses the Assume-Guarantee Reasoning Environment (AGREE) [3], a formal compositional reasoning tool for Architecture Analysis and Design Language (AADL) [4] models. Compositional reasoning partitions the formal analysis of a complex system architecture into verification tasks corresponding to the architecture's decomposition. By partitioning the verification effort into proofs about each subsystem within the architecture, the analysis will scale to handle large system designs.

Although AGREE does not suffer from some of the scalability issues inherent in other formal methods frameworks due to the compositional nature of the analysis, generated counterexamples can still be difficult to understand, especially for formal methods novices when the counterexamples contain several steps, each consisting of multiple variables. This problem is not unique to AGREE, but is common to most model checkers in use today [5]. Recently, however, a novel approach to producing explainable counterexamples has emerged in the form of generative AI.

Research on applying generative AI to formal reasoning has already gained significant attention. For instance, OpenAI researchers conducted pioneering work in 2020, leveraging large language models (LLMs) for mechanical theorem proving [6]. This resulted in the development of GPT-f, a proof assistant for Metamath, which achieved a 56% success rate and proved 200 theorems [7]. Other studies have explored LLMs for proof generation and repair. First et al. achieved a 50% success rate in proof repair for Isabelle/HOL [8], using Minerva [9], a model based on Google's PaLM [10]. Research has also examined GPT-3.5 and GPT-4 for Coq theorem proving [11], primarily focusing on diagnosing failed proofs. LLMs have further been applied to discover program invariants [12], [13] and support automated reasoning, as seen in the Clover project by Stanford and VMware, which emphasizes verifiable code generation [14].

In previous work [15], [16], Tahat et al. developed a copilot for large-scale proof repair using multi-shot conversational learning. The approach achieved a 97% success rate across 58 theorems from a repository containing 20,000 lines of Coq code from the Copland proofbase. Additionally, they

[1]https://loonwerks.com/projects/inspecta.html

introduced an evaluation framework to assess the convergence of dialogues toward predefined proof sets.

In this paper, we present our current work on using generative AI to provide clear and concise explanations of counterexamples generated by AGREE. Although using generative AI for explainable formal verification has been explored in other works (e.g., [17]), to the best of our knowledge, this is the first application of applying generative AI for producing explainable counterexamples from compositional reasoning over architecture models. Our initial results indicate this approach is well-suited for providing clear explanations of root cause, as well as suggestions for addressing the contract violations.

II. Explainable AGREE

A. Overview

AGREE provides a formal contract language for specifying *assumptions* (i.e., expectations on a component's input and the environment) and *guarantees* (i.e., bounds on a component's behavior). Because AGREE is implemented as an AADL *annex* in the Open Source AADL Tool Environment (OSATE), the contracts are specified directly on components in the AADL model. AGREE then uses a k-induction model checker to prove properties about one layer of the architecture using properties allocated to subcomponents. The analysis proves correctness of (1) component interfaces, such that the output guarantees of each component must be strong enough to satisfy the input assumptions of downstream components, and (2) component implementations, such that the input assumptions of a system along with the output guarantees of its subcomponents must be strong enough to satisfy its output guarantees.

When a contract violation is found (i.e., when an assumption is determined to be invalid or a guarantee is unsupported), AGREE produces a counterexample consisting of values for each system variable at each execution step. A sample counterexample is depicted in Figure 1. Currently, OSATE includes the AADL Simulator tool that can accept an AGREE counterexample as input and walk through the trace in the graphical editor, but it is of limited help when it comes to identifying the root cause of the contract violation.

B. Making Counterexamples Actionable

We therefore desire AGREE counterexamples that are *actionable*; that is, an explanation of the violation in terms that will quickly lead to a passing analysis (e.g., by making changes to the model or formal contract). To achieve this, we implemented an interactive conversational copilot powered by GPT-4o (omni) multi-modal generative AI, specifically developed to assist AGREE users in identifying the root causes of counterexamples and to support the subsequent model repair process. It was designed to be user-friendly and integrates with the OSATE IDE (see Figure 2).

In the remainder of this section, we detail our methodology and present our key findings using the `Integer_Toy` and `Car` models included with the AGREE distribution.

Counterexample

Variables for the selected component implementation

Variable Name	0	1	2
Inputs:			
{Target Speed.val}	121	0	0
{Target Tire Pitch.val}	0	1/5	0
State:			
{[G car 1] actual speed is less than constant target speed}	true	true	false
{_TOP.AXL..ASSUME.HIST}	true	true	true
{_TOP.CNTRL..ASSUME.HIST}	true	true	true
{_TOP.SM..ASSUME.HIST}	true	true	true
{_TOP.THROT..ASSUME.HIST}	true	true	true
{const_tar_speed}	true	false	true
Outputs:			
{Actual Speed.val}	11	10	100/11
{Actual Tire Pitch.val}	0	1/5	0
{State Signal.val}	0	0	0

Variables for AXL

Variable Name	0	1	2
Inputs:			
{AXL.Speed.val}	46	46	46
{AXL.Target Tire Direction.val}	0	1/5	0
State:			
{AXL..ASSUME.HIST}	true	true	true
Outputs:			
{AXL.Actual Tire Direction.val}	0	1/5	0

Variables for CNTRL

Variable Name	0	1	2
Inputs:			
{CNTRL.Actual.val}	11	10	100/11
{CNTRL.Target.val}	121	0	0
State:			
{CNTRL..ASSUME.HIST}	true	true	true
{CNTRL.e}	110	-10	-100/11
{CNTRL.e_dot}	-110	120	-10/11
{CNTRL.e_int}	110	100	-210/11
{CNTRL.u}	110	-10	-100/11
Outputs:			
{CNTRL.Actuator_Input}	110	-10	-100/11

Variables for SM

Variable Name	0	1	2
Inputs:			
State:			
{SM..ASSUME.HIST}	true	true	true
Outputs:			
{SM.State_Out.val}	0	0	0

Variables for THROT

Variable Name	0	1	2
Inputs:			

Fig. 1: AGREE counterexample generated from the `Car` model.

C. Contextual Prompt Constraints Problem

The GPT-4o generative multi-modal model exhibits significant power in translating human instructions into code and vice versa, particularly when the language in question has been part of its pre-training data and there exists a substantial open-source code base, such as C or Python. However, this capability comes with the drawback of potential hallucinations. Since AGREE is not as widely adopted as languages like C or Python, this problem is exacerbated. Consequently, the lack of relevant context is a significant challenge for generating explainable AGREE counterexamples.

To mitigate the contextual prompt constraints problem, we implemented a dynamic Retrieval-Augmented Generation (RAG) system, allowing it to adjust its context based on user inquiries.

Despite GPT-4o's 128k token capacity, which we estimate can accommodate several thousand lines of AADL in a single prompt, uploading an entire repository's contents can be prohibitively expensive and may well exceed the prompt token limitations. We therefore implemented a practical two-

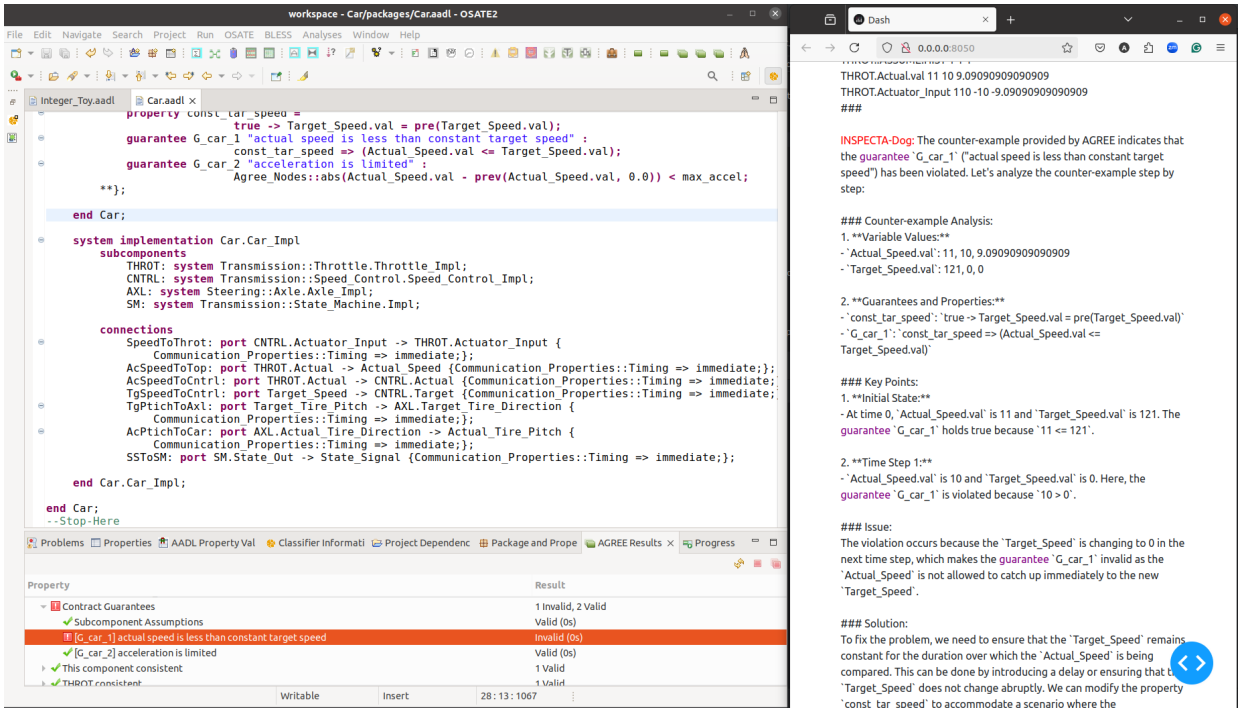


Fig. 2: AGREE copilot in OSATE provides an explanation for a counterexample generated on the `Car` model.

step optimization technique to meet our current needs.

First, the RAG system reads the top-level AADL file. It then parses the file's import chain, extracting only the files in the model workspace that are specified on this chain. This step significantly reduces the initial prompt size. The second optimization addresses another practical requirement: handling parts of the repository that may have been included in the model's pre-training data, such as core libraries. To manage this, the RAG system applies a filtering technique to the file names, guiding the system to ignore certain files, such as standard libraries, and retain user-defined files. This approach further reduces the initial prompt size to include only files that the model has not previously encountered. Finally, user inputs are automatically incorporated into the extracted context from the current file and its import chain, allowing for more accurate responses to user inquiries.

This approach significantly mitigates contextual constraint-based hallucinations that can arise from the absence of AADL model specifications. However, it does not address the absence of the counterexample itself or the lack of guidance on the critical system requirements that should be preserved during the model repair process.

D. Model Repair Problem

Given that counterexamples are generated interactively and may not be included in the initial context, we dynamically extend the RAG system. This allows users to upload an exemplar AADL/AGREE model along with a corresponding counterexample (in text or CSV format). Upon submission, the copilot provides a detailed, step-by-step explanation of the counterexample, identifies its root cause(s), and suggests potential solutions, as shown in Figure 2.

Summary:
The key issue was the mismatch between the requirement for the top-level system's output (`< 50`) and the guarantee in the model (`< 70`). By aligning the guarantee in the top-level system with the requirement, we ensure consistency and eliminate the counterexample.

Fig. 3: Refined explanation using a requirements file for the `Integer_Toy` model.

However, a significant challenge encountered was that these explanations and suggested alternatives could include two types of hallucinations, both syntactic and semantic. The former are typically minor and can be detected and resolved using a multi-shot approach. The latter are more problematic, as the copilot might suggest altering a component's guarantee, which could successfully remove the counterexample but risk violating core system requirements that should remain unchanged. We refer to this as the *Model Repair Problem*.

1) Requirements for Counterexample Explanations: To mitigate the Model Repair Problem, we configured the tool to generate solutions that conform to a predefined set of system requirements written in natural language, which are uploaded via a CSV file or directly included in the context.

As a result, the tool was able to more accurately identify the root cause and suggest appropriate solutions, as demonstrated in Figure 3. This refinement significantly enhanced the accuracy of the recommendations.

E. Preliminary Results and Conversational Quality Assessment Problem

Our initial evaluations were conducted manually, focusing on the copilot's ability to accurately identify the root cause of the counterexamples, repair the model, and ensure compliance with the requirements. The system was evaluated on two case studies. The first case study involved the `Integer_Toy` model, while the second dealt with a larger model that imports several files, totaling 7 files and approximately `380` lines of AADL. The copilot successfully identified the root cause of all counterexamples for the specified guarantees (13 out of 13) on the first attempt, demonstrating a high degree of accuracy. However, these manual evaluations highlight the need for greater automation; consequently, we plan to develop a more automated evaluation system to enable testing on more realistic and complex use cases.

We are in the process of selecting a golden set of examples from a formally verified library we developed previously, and constructing a testing set by introducing deliberate violations. These examples will be used by the copilot to evaluate its ability to correctly identify the root causes. While we have demonstrated initial success in this area, model repair remains a more complex challenge. This is because repairing models can result in multiple solutions, particularly for more intricate use cases. One of the key limitations is the tool's ability to consistently remove counterexamples while ensuring compliance with the specified requirements. To address these issues, we are developing a toolset aimed at measuring convergence towards the correct semantics of the golden examples, within a few-shot learning context. This remains an ongoing challenge that we will address in our future work.

III. Conclusion

In an effort to make AGREE results more explainable, we have developed a generative AI-based tool that produces natural language explanations from (potentially complex) AGREE counterexamples. Although initial results are encouraging, we will continue to evaluate our approach on increasingly complex models and formal specifications. In addition, we believe usability of other AGREE features can also benefit from generative AI. The most obvious candidates are the formalization of AGREE contracts from natural language requirements and the modification of models to conform to their contracts.

Our prototype implementation is currently loosely coupled with AGREE and OSATE. In order to truly address AGREE usability, tighter tool integration is required, and will be the focus of upcoming work as we continue to refine the tool. We envision an integrated copilot that smartly parses the model abstract syntax tree, interacts with the user, and makes automated updates to the AGREE contracts and AADL model by using features provided by the IDE.

INSPECTA includes a DevOps Assurance Dashboard for displaying development status, analysis results, and progress towards achieving assurance goals. The explainable counterexamples will be accessible from the dashboard, and therefore a mechanism will need to be implemented to retrieve them from the modeling workspace and display them properly. The dashboard will also capture and display tool usage metrics to help us better understand the degree to which a more explainable counterexample aids the user in addressing a requirement or design issue (e.g., by tracking the number of times the model is modified or AGREE is run before producing a passing result). Metrics analysis will in turn drive new usability enhancements in AGREE. We look forward to sharing the outcome of these efforts in the near future.

IV. Acknowledgment

This work was funded by DARPA contract FA8750-24-9-1000. The views, opinions and/or findings expressed are those of the authors and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government.

References

[1] RTCA, *DO-333: Formal Methods Supplement to DO-178C and DO-278A*, December 2011.

[2] J. A. Davis, M. Clark, D. Cofer, A. Fifarek, J. Hinchman, J. Hoffman, B. Hulbert, S. P. Miller, and L. Wagner, "Study on the barriers to the industrial adoption of formal methods," in *Formal Methods for Industrial Critical Systems*, C. Pecheur and M. Dierkes, Eds. Springer Berlin Heidelberg, 2013, pp. 63–77.

[3] D. Cofer, A. Gacek, S. Miller, M. W. Whalen, B. LaValley, and L. Sha, "Compositional verification of architectural models," in *NASA Formal Methods*, A. E. Goodloe and S. Person, Eds. Springer, 2012, pp. 126–140.

[4] P. H. Feiler and D. P. Gluch, *Model-Based Engineering with AADL: An Introduction to the SAE Architecture Analysis & Design Language*, 1st ed. Addison-Wesley Professional, 2012.

[5] A. P. Kaleeswaran, A. Nordmann, T. Vogel, and L. Grunske, "A systematic literature review on counterexample explanation," *Information and Software Technology*, vol. 145, 2022.

[6] S. Polu and I. Sutskever, "Generative language modeling for automated theorem proving," *arXiv preprint arXiv:2009.03393*, 2020.

[7] N. Megill and D. A. Wheeler, "Metamath: A computer language for mathematical proofs," 2019.

[8] E. First, M. N. Rabe, T. Ringer, and Y. Brun, "Baldur: Whole-proof generation and repair with large language models," *arXiv preprint arXiv:2303.04910*, 2023.

[9] A. Lewkowycz, A. Andreassen, D. Dohan *et al.*, "Solving quantitative reasoning problems with language models," *arXiv preprint arXiv:2206.14858*, 2022.

[10] A. Chowdhery, S. Narang, J. Devlin *et al.*, "Palm: Scaling language modeling with pathways," *arXiv preprint arXiv:2204.02311*, 2022.

[11] S. Zhang, E. First, and T. Ringer, "Getting more out of large language models for proofs," *arXiv preprint arXiv:2305.04369*, 2023.

[12] K. Pei, D. Bieber, K. Shi *et al.*, "Can large language models reason about program invariants?" *Proceedings of the 40th International Conference on Machine Learning*, July 2023.

[13] H. Wu, C. Barrett, and N. Narodytska, "Lemur: Integrating large language models in automated program verification," *arXiv preprint arXiv:2310.04870*, 2023.

[14] C. Sun, Y. Sheng, O. Padon, and C. Barrett, "Clover: Closed-loop verifiable code generation," *arXiv preprint arXiv:2310.17807*, 2024.

[15] A. Tahat, D. Hardin, A. Petz, and P. Alexander, "Proof repair utilizing large language models: A case study on the copland remote attestation proofbase," in *Proceedings of International Symposium On Leveraging Applications of Formal Methods Verification and Validation (AISolA)*, 2024.

[16] ——, "Metrics for large language model generated proofs in a high-assurance application domain," in *High Confidence Software and Systems Conference (HCSS'24)*, 2024.

[17] R. Martins, "Transforming logic into language: Bridging the gap with large language models," in *2nd International Workshop on Explainability of Real-time Systems and their Analysis (ERSA'23)*, December 2023.

Strengthening Real-Time Analysis by Evolving Counterexamples

Leo Bishop and Leandro Soares Indrusiak
School of Computer Science
University of Leeds
Leeds, United Kingdom

***Abstract*—Analysis models are commonly developed for real-time systems to provide temporal upper bounds for system behaviour. However, state-of-the-art analysis models have repeatedly been shown to be flawed by the identification of counterexamples demonstrating system temporal behaviour that exceed the model's upper bound. We propose a novel approach to evaluate analysis models: automated identification of counterexamples using heuristic search algorithms. We demonstrate that a simple genetic algorithm, a popular type of heuristic search algorithm, is rapidly and reliably able to produce counterexamples for a Network-on-Chip analysis model across a range of Network-on-Chip configurations. We provide a proof of concept for the wider application of genetic algorithms to real-time system analysis models, aiding in the faster identification of flaws in existing models and improving confidence in models for which counterexamples cannot be identified.**

***Index Terms*—real-time systems, analytical models, genetic algorithms, schedulable analysis, counterexamples**

I. Introduction and Background

Effective safety analysis requires considering worst-case scenarios [1], but identifying worst-case behaviour in complex systems is far from trivial. Decades of research and development have produced numerous real-time analysis models that are able to upper-bound the temporal behaviour of a variety of computing and communication systems. For simple systems, a few proof sketches can provide enough intuition to convince a well-informed expert that the model is safe, i.e. that its outputs will always upper-bound the system's temporal behaviour even in worst-case scenarios. As the systems become more complex, it is increasingly difficult to convince oneself and others that a particular model is safe for a given system.

Maida et al. [2] discuss the consequences of trusting unsafe analysis models supported by misleading proof sketches, and argue for the use of automated proof checking to obtain trustworthy upper-bounds. While we fully agree with their argument, we are also aware of the difficulties associated with automated proof checking in systems where worst-case scenarios arise out of complex patterns of interdependence between components. Over the past decades there were several analytical models that were initially considered safe and correct, only to be proven unsafe after years of use in a variety of domains such as automotive buses [3], on-chip networks [4] and self-suspending task schedulers [5]. Due to the complexity of the respective systems, the flaws of those analytical models were not exposed by failing to pass a proof checker. They were exposed by counterexamples: scenarios that trigger a temporal behaviour from the system that exceeds the upper-bound outputs of the model, proving it to be unsafe.

In this paper, we present techniques that aim to elicit counterexamples for systems too complex to be described by formal statements that can be automatically proof-checked. Our techniques only require an executable model of the system (e.g. a simulator or prototype) and the analytical model for which we are seeking counterexamples. Our aim is to find counterexamples that can help us explain if and why a model is flawed, and use that information to try to correct it.

Our search for counterexamples is certainly not exhaustive, as that is prohibitive even for relatively simple systems, so we can never prove the correctness of a given model. However, if we can show that our techniques are effective at finding counterexamples for models that are flawed, we can use them to increase our confidence in the correctness of models for which no counterexamples can be found.

II. Case Study

The field of Networks-on-Chip (NoCs) refer to the application of wide-area networking techniques such as routers, switch-based routing and packet based communication to Systems-on-Chip (SoCs) [6] replacing traditional bus or point-to-point communication.

For SoCs with hard real-time communication requirements, traffic flows (packet streams which traverse a consistent route across the NoC) must be schedulable, i.e. all packets belonging to a given traffic flow meet a defined deadline even in worst case scenarios [7]. A seminal 2008 paper by Shi and Burns [7] proposed a novel analysis model to estimate the upper bound of packet latency, finally enabling guarantees of traffic flow schedulability. The analysis, which we refer as S&B, breaks down the causes of packet latency as:

- Basic latency: network latency experienced by a packet which doesn't experience resource contention.
- Direct interference: latency imposed on a packet by higher priority packets directly competing for the same network components, e.g. buffer space or interconnect bandwidth, as shown in Fig. 1(a).
- Indirect interference: latency imposed by packets that do not directly interfere but impose resource contention on intermediary packets which impose direct interference on the packet under consideration, as shown in Fig. 1(b).

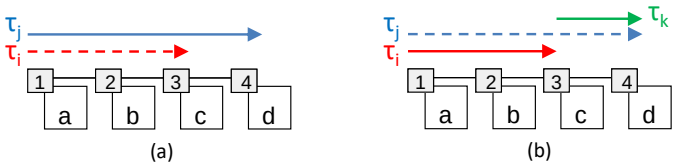


Fig. 1. Direct (a) and Downstream Indirect (b) Interference

S&B analysis was widely cited and extended in the years following its publication, with several works improving the tightness of its bounds [8] [9]. However, eight years after publication, Xiong et al. [10] identified counterexamples showing that under a specific combination of traffic flow routes S&B produces optimistic upper bounds for packet latency. This combination of routes, termed downstream indirect interference, is illustrated in Fig. 1(b). A key assumption in S&B is that each individual flit [11], e.g. from τ_j, can only apply interference to a each lower priority packet it contends with, e.g. from τ_i, once [4]. However, under the Fig. 1(b) example, as a flit from τ_j experiences interference from τ_k's packets, the flit may repeatedly interfere with a single τ_i packet across both $1 \rightarrow 2$ and $2 \rightarrow 3$, violating this assumption and allowing for the production of optimistic upper bounds by S&B.

Along with the identification of counterexamples for S&B, Xiong et al. proposed a corrected analysis in [10], which was also shown to be unsafe by a counterexample by Indrusiak et al. in [12], along with a correction which paved the way for safer analyses accounting for the MPB problem [13] [14].

The use of counterexamples has invalidated several NoC analysis models, one of them after nearly a decade of use, demonstrating the need for more rigorous validation of the correctness of real-time system analysis models.

III. Evolving Counterexamples

Identifying counterexamples for an analysis model can be approached as a search problem covering the configuration space of a given model, enabling the application of search algorithms to find counterexample configurations. As systems and their models increase in complexity, the resulting increase in the size and complexity of the search space will reduce the efficiency of basic search algorithms. Therefore, the application of more complex heuristic search algorithms is proposed.

Genetic algorithms are a popular heuristic search algorithm inspired by Darwinian evolutionary processes which iteratively generate and evaluate the fitness of possible solutions to a given problem, generating improved solutions over time. Having previously been shown to be suitable for generating counterexamples for complex models [15] we use genetic algorithms for the generation of counterexamples in this paper.

This application of genetic algorithms to real-time system analysis models and the resulting identification of counterexamples may enable the faster identification of flawed models and increase our confidence in models for which counterexamples cannot be identified.

Considering the NoC case study outlined in section II, the set of parameters for S&B include, for each traffic flow: the period; release jitter; priority; maximum packet size and route. (Route is required so that direct and indirect interference relations can be extracted and basic latency calculated.) The analysis is, of course, affected by: the NoC topology and router design (which determines which routes are feasible); header flit routing delay (which affects basic latency); and number of virtual channels (which limit the number of supported priority levels). But in this paper we target generating counterexamples only by manipulating the analysis parameters and therefore use a fixed topology, header flit routing delay and virtual channel size when running our genetic algorithm. The simultaneous exploration of analysis parameters and NoC architectural features is an interesting direction for future work.

For n traffic flows, we define a search space with $4n$ dimensions considering the following four analysis parameters per traffic flow: period; release jitter; packet size and packet route. We represent these parameters as integers. Period, release jitter and packet size are represented as their face value whilst for routes, a list of all valid routes across the given NoC topology can be generated and each possible route mapped to an integer identifier. This allows a set of traffic flows to be represented as a list of integers, which can be used as a set of genes enabling the application of a genetic algorithm.

As a result of its iterative nature, S&B analysis is only valid for traffic flows it considers schedulable, i.e. once a traffic flow's upper bound exceeds its deadline S&B no longer provides a valid temporal upper bound. This introduces an additional problem that must be handled by the genetic algorithm: that of generating traffic flows that are schedulable according to S&B. Whilst an automated process for generating schedulable traffic flows is an interesting and valuable problem to address, it is outside of the scope of this paper. As such, we only consider schedulable sets of traffic flows, discarding sets of genes generated by the genetic algorithm that result in unschedulable traffic flows. We therefore define the process for generating a new individual as:

1) Generate a new individual with a random set of genes.
2) Construct the corresponding set of traffic flows from the individual's genes.
3) Calculate each traffic flow's packet latency upper bound using S&B.
4) If all traffic flows are schedulable according to S&B, add the individual to the new population.

We developed a naive fitness function based on the comparison of S&B upper bounds against simulated packet latency.

For each individual, the simulated packet latency values are produced by simulating the corresponding set of traffic flows using a cycle accurate, transaction level NoC simulator adhering to the restrictions imposed by S&B [16].

Given real-time NoCs under the constraints imposed by S&B are deterministic, traffic flows' basic latency is consistent and easily predictable. For traffic flows which don't experience interference the simulated latency therefore always matches

the S&B upper bound (as these values are both purely a function of basic latency).

TABLE I
EXAMPLE INDIVIDUAL'S TRAFFIC FLOWS' LATENCY AND FITNESS.

Traffic Flow	Basic	S&B Upper Bound	Max Simulated	Fitness
t1	99	99	99	0
t2	143	342	332	-10
t3	120	318	312	-6
t4	16	136	184	48

As a result, traffic flows that experience only basic latency are ignored for the purposes of calculating an individual's fitness. These traffic flows are therefore identified by comparing the traffic flow's basic latency prediction with its S&B value and are then discarded from consideration. All remaining traffic flows (those experiencing interference) have their excess simulated latency over S&B upper bound calculated, i.e. *Max Simulated − S&B Upper Bound*. Finally, the individual's overall fitness is taken as the largest excess latency value produced by its valid traffic flows.

For the example individual shown in table I, *t1* is discarded as its basic latency matches its S&B value, meaning it experienced no interference. The excess values for *t2*, *t3* & *t4* are then calculated as outlined above, giving -10, -6 & 48 respectively. The individual's fitness is therefore 48, the largest excess latency value from the remaining traffic flows which experienced interference.

All our genetic algorithm runs used a 0.1 elitism rate, and a simple elitist selection algorithm with a 0.4 selection rate, single point crossover and a 0.05 mutation rate.

IV. EVALUATION

We show in Table II that across a range of NoC topologies our simple genetic algorithm is reliably and rapidly able to generate counterexamples containing traffic flows whose maximum simulated latency exceeds the S&B upper bound.

Having established the ability of our genetic algorithm to produce counterexamples, we attempt to use our genetic algorithm to produce improved and optimised counterexample configurations, maximising the excess simulated latency over the S&B upper bound. Our aim is to identify the margin by which a given analysis model underestimates the actual behaviour of the system it represents, as that provides more insight on the potential real-world consequences of using a system that complies to a flawed analysis model.

As shown in Table III, our genetic algorithm is able to reliably optimise counterexample configurations, resulting in significantly larger excess latency than the initial counterexample produced by that genetic algorithm run.

We note several runs experienced premature convergence, where the genetic algorithm is unable to produce improved individuals from its existing population via crossover. This is a common issue with genetic algorithms which is the result of the repeated reproduction of the best observed individuals and the use of crossover to produce new individuals [17].

TABLE II
AVERAGE NUMBER OF GENERATIONS TO PRODUCE A COUNTEREXAMPLE.[a]

Topology	No. Traffic Flows	Success Rate[b]	Mean Generations[c]
3x3	10	1	1.05
5x2	10	1	1
5x5	10	0.95	2.33
10x10	20	1	1.35

[a] Population size of 50 individuals.
[b] Generated a counterexample within 50 generations.
[c] Excluding runs which failed to produce a counterexample.

TABLE III
GENETIC ALGORITHM OPTIMISATION OF EXCESS LATENCY, EXPRESSED AS A FRACTION OF S&B UPPER BOUND*

Gens to Exceed	Initial Excess	Convergence Gen+	Worst Excess
1	0.18	38	0.28
1	0.12	75	0.4
2	0.12	65	0.23
1	0.04	96	0.43
1	0.14	73	0.44
1	0.05	51	0.34

* 5x5 NoC topology with 10 traffic flows & 100 individuals per population.
+ Genetic algorithm capped at 100 generations.

Prior to termination at 100 generations, 1/2 converged with at least 35 generations remaining whilst 2/3 converged with at least 27 generations remaining.

V. DISCUSSION

The reliable generation of counterexamples by our genetic algorithm across a range of NoC topologies provides a proof of concept for the application of genetic algorithms more widely to both more complex NoC analysis models and more broadly to analysis models in other real-time system domains.

The genetic algorithm configuration was selected to provide the simplest feasible implementation. The use of simple selection and crossover methods and the naive fitness function, exploiting only rudimentary knowledge of the characteristics of S&B, demonstrates the simplicity of counterexample generation for genetic algorithms. This simplicity also provides space for the application of more complex techniques when applied to more complex analysis models. Furthermore, the repeated success of our genetic algorithm at optimising counterexamples to produce increased excess latency provides further evidence of the suitability of genetic algorithms for this problem.

We must note the problem of generating valid configurations. Many analysis models, including S&B, are only valid for a subset of possible configurations. For these models the search space is constrained and the generation of counterexamples becomes dependent upon first being able to generate valid configurations. For NoCs it is viable to resort to the repeated generation of configurations to brute force the generation of a sufficient quantity of viable configurations. This approach

may not, however, be viable when applied to more complex problem domains where the valid search space is heavily constricted. This may limit the application of genetic algorithms to problem domains with either relatively large valid search spaces or those where a viable method of automatically generating valid configurations has been developed.

We must also address the simplification of the search space used in this paper. Based on our prior knowledge of the flaws in S&B's model, that MPB can arise in any NoC topology with a sufficiently large virtual channel size, these parameters were fixed and discarded as irrelevant to the generation of counterexamples. Whilst the removal of these parameters has no bearing on the feasibility of genetic algorithms for producing counterexamples the inclusion of irrelevant parameters in the search space may reduce the efficiency of the genetic algorithm [18] when applied to novel models.

There is also the practical consideration of performance. As our approach requires the use of a simulator or prototype, for all but the simplest real-time systems, the execution of this software is the key component in the runtime and resource usage of the genetic algorithm. We achieved a significant reduction in our genetic algorithm's runtime by utilising parallelisation, executing the simulator instances, for each individual in a given population, in parallel. As our NoC simulator's execution is primarily sequential the application of parallelisation reduced our genetic algorithm runtime to a fraction of a its original sequential runtime.

Moving to our counterexample optimisation, we suggest that the repeated occurrence of premature convergence in our results is primarily a result of the limited requirements of downstream indirect interference and MPB in NoCs. Valid cases of downstream indirect interference that result in MPB only require three traffic flows, rendering the individual's remaining *n-3* traffic flows redundant. By rearranging these redundant traffic flows, whilst leaving the three critical traffic flows unaltered, the genetic algorithm can produce additional novel configurations with the same fitness value. This may result in a rapid reduction in genetic diversity within the population, the main factor in premature convergence [19], as the genetic algorithm produces multiple configuration with equal fitness and no meaningful difference.

Whilst premature convergence did not pose a significant issue in the production of counterexamples for S&B, its rapid onset suggests that when applied to analysis models with a more complex search space or those where counterexamples compose a smaller proportion of the search space, premature convergence may to become a key issue. Premature convergence in genetic algorithms has been widely explored with a range of mitigation strategies proposed, primarily focused on preventing a reduction in the population's genetic diversity as the genetic algorithm converges [19] [20] [21].

VI. Conclusion

We successfully demonstrate the ability of a naive genetic algorithm to rapidly and reliably generate counterexamples for Shi & Burn's NoC analysis model. This provides a proof of concept for the application of genetic algorithms to the problem of discovering counterexamples for real-time system analysis models across a broader range of domains. This will enable automated evaluation of analysis models whose complexity renders proof sketches unusable. The ability to more easily identify counterexamples will ultimately enable faster identification of analysis models' flaws and the production of corrections, as well as improved confidence in models for which counterexamples cannot be identified.

References

[1] N. G. Leveson, "The Use of Safety Cases in Certification and Regulation," J. Syst. Saf., vol. 47, 2011.

[2] M. Maida, S. Bozhko, and B. B. Brandenburg, "Foundational Response-Time Analysis as Explainable Evidence of Timeliness," in Proc. of the 34th Euromicro Conference on Real-Time Systems (ECRTS), 2022.

[3] R. I. Davis, A. Burns, R. J. Bril, and J. J. Lukkien, "Controller Area Network (CAN) schedulability analysis: Refuted, revisited and revised," Real-Time Systems, vol. 35(3), 2007.

[4] L. S. Indrusiak, A. Burns, and B. Nikolic, "Buffer-aware bounds to multi-point progressive blocking in priority-preemptive NoCs," in Proc. of the 2018 Design, Automation and Test in Europe Conference (DATE), 2018.

[5] J. Chen et al., "Many suspensions, many problems: a review of self-suspending tasks in real-time systems," Real-Time Systems, vol. 55(1), 2019.

[6] J. Henkel, W. Wolf, and S. Chakradhar, "On-chip networks: A scalable, communication-centric embedded system design paradigm," in 17th International Conference on VLSI Design, 2004.

[7] Z. Shi and A. Burns, "Real-time communication analysis for on-chip networks with wormhole switching," in Second ACM/IEEE International Symposium on Networks-on-Chip, 2008.

[8] H. Kashif, S. Gholamian, and H. Patel, "SLA: A Stage-Level Latency Analysis for Real-Time Communication in a Pipelined Resource Model," IEEE Transactions on Computers, vol. 64(4):1177–1190, 2015.

[9] B. Nikolic, L. S. Indrusiak, and S. M. Petters, "A Tighter Real-Time Communication Analysis for Wormhole-Switched Priority-Preemptive NoCs," arXiv:1605.07888 [cs], 2016.

[10] Q. Xiong, Z. Lu, F. Wu, and C. Xie, "Real-time analysis for wormhole NoC: revisited and revised," in Proceedings of the 26th edition on Great Lakes Symposium on VLSI, 2016.

[11] L. M. Ni and P. K. McKinley, "A survey of wormhole routing techniqes in direct networks", Computer, vol. 26(2):62-76, 1993.

[12] L. S. Indrusiak, B. Nikolic, and A. Burns, "Analysis of buffering effects on hard real-time priority-preemptive wormhole networks," arXiv:1606.02942 [cs], 2016.

[13] Q. Xiong, F. Wu, Z. Lu, and C. Xie, "Extending real-time analysis for wormhole NoCs," in IEEE Transactions on Computers, 66(9), 2017.

[14] B. Nikolic, S. Tobuschat, L. S. Indrusiak, R. Ernst, and A. Burns, "Real-time analysis of priority-preemptive NoCs with arbitrary buffer sizes and router delays," Real-Time Systems, vol. 55:63-105, 2019.

[15] T. Zheng and Y. Liu, "Genetic algorithm for generating counterexample in stochastic model checking," in Proc. of the 2018 VII International Conference on Network, Communication and Computing, 2018.

[16] L. Bishop, "CyNoC," (1.5.4), [Software], Available from: https://github.com/LBishop234/CyNoC.

[17] D. B. Fogel, "An introduction to simulated evolutionary optimization," in IEEE Transactions on Neural Networks, vol. 5(1), 1994.

[18] M. S. Kadu, R. Gupta, and P. R. Bhave, "Optimal design of water networks using a modified genetic algorithm with reduction in search space," in Journal of Water Resources Planning and Management, vol. 134(2), 2008.

[19] H. M. Pandeya, A. Chaudhary, and D. Mehrotrac, "A comparative review of approaches to prevent premature convergence in GA," in Applied Soft Computing, vol. 24, 2014.

[20] K. Jebari and M. Madiafi, "Selection method for genetic algorithm," in International Journal of Emerging Sciences, Vol. 3(4), 2013.

[21] A. A. Gozali and S. Fujimura, "Localization strategy for island model genetic algorithm to preserve population diversity," in Compute and Information Science, 2018.