

ソフトウェア・エンジニアリングの 新しい潮流

Deep Specificationの世界



When exhaustive testing is impossible —i.e., almost always— our trust can only be based on proof (be it mechanized or not).

-- Dijkstra

*Given the proof, deriving a program justified by it,
is much easier than, given the program,
constructing a proof justifying it.*

-- Dijkstra

はじめに

- 小論の目的は、ソフトウェア・エンジニアリングの領域で起きている“Deep Specification”という新しい流れをIT技術者に紹介することである。
 - 第一部では、ソフトウェア・エンジニアリング見直しの背景として、現在の大規模システムの開発がはらむ問題をいくつかとりあげた。
 - 現代では、ITシステムは、社会のあらゆる領域に深く浸透している。大規模システムの開発が、明示的にせよ潜在的にせよかかえる問題が、「バグのない安全なITシステムを社会に提供する」という点で、大きな弱点となる可能性はあると筆者は考えている。
-

はじめに

- セキュリティに関心を持つIT技術者は多いと思う。“Deep Specification”のようなソフトウェア・エンジニアリングの分野で起きている変化にも、関心を持つ人が増えることを期待したい。
- 第二部では、“Deep Specification”の概略を紹介する。開発にコンピュータの力を借りるのは、当たり前のことなのだが、“Deep Specification”が借りようとしているのは、コンピュータが「論理的＝数学的推論」を正しく行う能力を持っているという、その「力」である。
- ここでは、“Deep Specification”の取り組みの中で、どのような具体的なプロジェクトが走っているのかの紹介を中心に行った。

はじめに

- これらの取り組みは、一見、抽象的で理論的なレベルでの試行に見える。また、その「成果」もリアルにみれば限定的なものだ。ただ、第一部で紹介したいくつかの問題への対応を強く意識しているという点では、現実的で実践的な問題意識にドライブされていることには注意が必要だと思う。
- “Deep Specification” のムーブメントは、1980年代にDijkstraらが推進しようとしたコンピュータ・サイエンスへの「形式的手法」の導入を、現代に復活させようとしているものと見ることができる。
- 第三部は、「形式的手法」の方法論的中核である「プログラミングと論理」の関係の認識について触れたものである。ただ紙幅の制限で、40年前の議論の歴史的回顧中心にとどまっている。これについては別の機会に補いたい。

はじめに

- 小論では、現代の「形式的手法」の技術的な詳細については、立ち入らなかった。それを理解するには、別の準備が必要だからである。特に、現代の「形式的手法」の Lingua Franca であるCoqの知識は不可欠である。次回のマルレクは、「Coq入門」をテーマにしようと思っている。
 - 小論のテーマは、前回のマルレクの「Yet Another AI（別の人工知能技術）」と内容的につながったものだ。特に前回の第三部「機械による数学的推論は可能か？」とは密接に関係している。今回の問題意識は、その問いのコンピュータ開発自体への適用、「機械によるプログラミングは可能か？」に関わるものである。
-

はじめに

- 数学をコンピュータ・プログラムの形で再構成しようとした VoevdskyらのUniMathの試みと、“Deep Specification” の試みが、ほぼ同じ時期に登場したのは偶然ではないのだ。
 - 小論では、あまり理論的な話ができなかったのだが、前回のマルレクのサブゼミ「型の理論入門」
<https://www.marulabo.net/docs/type-theory-2/>
が、前回と今回、そして次回のマルレク「Coq入門」をつなぐ理論的な背景説明になっている。興味がある方は、ぜひ参照してほしい。
-

第一部 **Agenda**

大規模システム開発の問題

- 航空機の開発
- 分散システムの開発
- ハードウェア開発
- 開発コスト
- 高速化
- 開発ツールは信頼できるか？

第二部 **Agenda**

Deep Specificationの世界

- ソフトウェア・エンジニアリングの展望
- Deep Specificationとは何か？
- Deep Specificationのプロジェクト
- 形式的手法によるCPUとネットワークの検証
- 進行中のプロジェクト

第三部 **Agenda**

プログラムと論理

- 形式的手法の二つの起源
- 形式的手法をめぐる20世紀の「論争」
- ダイクストラの21世紀へのメッセージ
- 証明支援システム Coq
- Deep Specification – Software Foundation

第一部

大規模システム開発の問題



第一部 **Agenda**

大規模システム開発の問題

- 航空機の開発
- 分散システムの開発
- ハードウェア開発
- 開発コスト
- 高速化
- 開発ツールは信頼できるか？

航空機の開発



ボーイング737MAX墜落事故

- 「2018年10月29日、インドネシアのジャワ海でライオン・エア610便(ボーイング737MAX 8型機、PK-LQP)が離陸後約10分で墜落、乗客乗員189名全員が死亡した。
飛行データの解析によれば、2つあるAoAセンサーのうち1つが故障により誤った角度を示していた。これにより、MCASが機首が上がりすぎていると判断して下降操作を行い、上昇操作を行っていたパイロットとせめぎ合いになる形で墜落した。」
- 「2019年3月10日、エチオピアのアディスアベバ、ボレ国際空港より離陸したエチオピア航空302便(ボーイング737MAX 8型機、ET-AVJ)が、離陸後約6分で墜落し、乗客乗員157名全員が死亡した。」



事故原因:

MCASの不具合と航空会社の対応の問題

- MCAS (Maneuvering Characteristics Augmentation System)を始動させるシグナルは、機首の角度を調べる2個のセンサー(AoA=迎え角センサー)から出る。2個のうち1個は正常だったが、残る1個が故障して異常に高い角度を示していた。MCASは高い角度のほうに反応するので、機首が上がりすぎていると判断し、強制的に下げようとしたのである。
- 航空会社の対応の問題
 - ボーイング社はMAX8を購入した航空会社に対して、MCASの機能を十分に説明していなかった。
 - ライオン・エアはパイロットに対し、MCASに異常があった場合はスイッチを切って手動操縦に戻すための手順を教育していなかった。
 - 事故機は前日の飛行でAoAに異常があり、MCASも異常動作を示していた。ライオン・エアはそれを知りながら修理をせずに再び出発させた。

ボーイング社の開発体制

- この事故の関連では、ブルームバーグ紙が、ボーイング社は737MAXのソフトウェア開発をインドのHCLという会社に委託していて、その中には大学を出たばかりの時給9ドルの若者がいたことをすっぱ抜いて話題になった。(ボーイング社は、今回の不具合にこの若者が関わったことを否定している。)
 - 現代の航空機は、「空飛ぶソフトウェア」と言われるほど、大規模なシステムなのだが、その高価なシステム開発に、時給1000円の人が使われていたのは、ちょっと驚きである。
-

Boeing's 737 Max Software Outsourced to \$9-an-Hour Engineers

- ❑ In offices across from Seattle's Boeing Field, recent college graduates employed by the Indian software developer [HCL Technologies Ltd.](#) occupied several rows of desks, said Mark Rabin, a former Boeing software engineer who worked in a flight-test group that supported the Max.
- ❑ The coders from HCL were typically designing to specifications set by Boeing. Still, "it was controversial because it was far less efficient than Boeing engineers just writing the code," Rabin said. Frequently, he recalled, "it took many rounds going back and forth because [the code was not done correctly.](#)"

「仕様」のバグか、「実装」のバグか？

- 先のブルームバーグの記事から、「仕様」はボーイング社が提供し、HCL社は「コード」を納品したことわかる。しかし、先の記事を見ても、不具合の原因が「仕様」の側にあったのか「コーダー」の実装の側にあったのか判然としない。これは大事なところである。
 - 個人的には、二つのセンサーの一方(場合によったら両方)が故障したり、異なる値を返した時の処理に疑問を持っている。ただ、これは分散システムの開発では避けて通れないPartial Fairure (システム全体は生きているが、個々のノードのいくつかは死んでいる)への対応の問題である。この点で、「仕様」がどうだったかが気になるのだ。
-

「空飛ぶソフトウェア」

「空飛ぶ大規模分散システム」

- 古い、ある意味「ハード」だけで動いていた飛行機には、Fail Safe のノウハウが二重三重に生かされていたと言う。ところが、現在の旅客ジェットは、ソフトウェアの塊であるだけでなく、それ自身が「空飛ぶ大規模分散システム」になっているのだ。分散システムには、分散システムの開発手法が必要である。
 - 問題は、飛行機の「自動操縦装置」をめぐっては、同じような事故が他でも起きていることだ。
 - 大量の死者を出すことは、あるいはないと思うのだが、同じような問題は、自動車の自動運転でも起こり得る。
-

中華航空140便墜落事故

- 「中華航空140便墜落事故 - 1994年、名古屋空港で発生した事故。着陸復航モードの誤作動と、モード解除に関する訓練が誤っていたことが原因で、機体がパイロットの操縦に逆らって機首を上げ続けたために失速し、名古屋空港の滑走路脇に墜落。乗員乗客271人の内264人が死亡した。」
- 140便の機材はエアバスA300であり、610便及び302便とはほぼ逆の状況下で起きたが、パイロットたちがモード解除の方法を知らず、自動システムがパイロットの操作を受け付けなくなり墜落に繋がったことなど、類似点が多い。

分散システムの開発

分散システムの開発の難しさ

- システムの構築方法が変わるにつれ、開発の方法も変化する。現代のシステムは、規模の大小を問わず、またユーザ(場合によっては開発者も！)がそれを意識する否かを問わず、ネットワークをまたいでサービスを交換するのが、基本的なスタイルになっている。
 - こうした分散システムの開発には、そうではないモノリシックなシステムの開発とは異なる注意が必要になる。分散システム上でプログラムを正しく実行させるのは、そうでないシステム上でプログラムを正しく実行させるより、本当は、難しいのだ。
-



社会インフラとしての分散システム



<http://bit.ly/2kNjlcN>



分散システム上のアプリケーション



<http://bit.ly/2kNjlcN>

誰もが分散システムを作る時代

- 先日も、AWSで大規模な障害が起きたが、ここで取り上げたいのはそうしたことはない。クラウド事業者は、基本的には、分散システム設計のプロで、何が設計の肝で、トラブルにどう対応すればいいのかをよく理解していると、僕は信じている。
 - 問題は、一般のユーザー企業も、普通のプログラマーも、そのことを意識するか否かを問わず、事実上の分散システムを作ることが、今では当たり前になっていることである。
-

分散システムの構築の難しさによるトラブルは、枚挙にいとまがないほど多い。

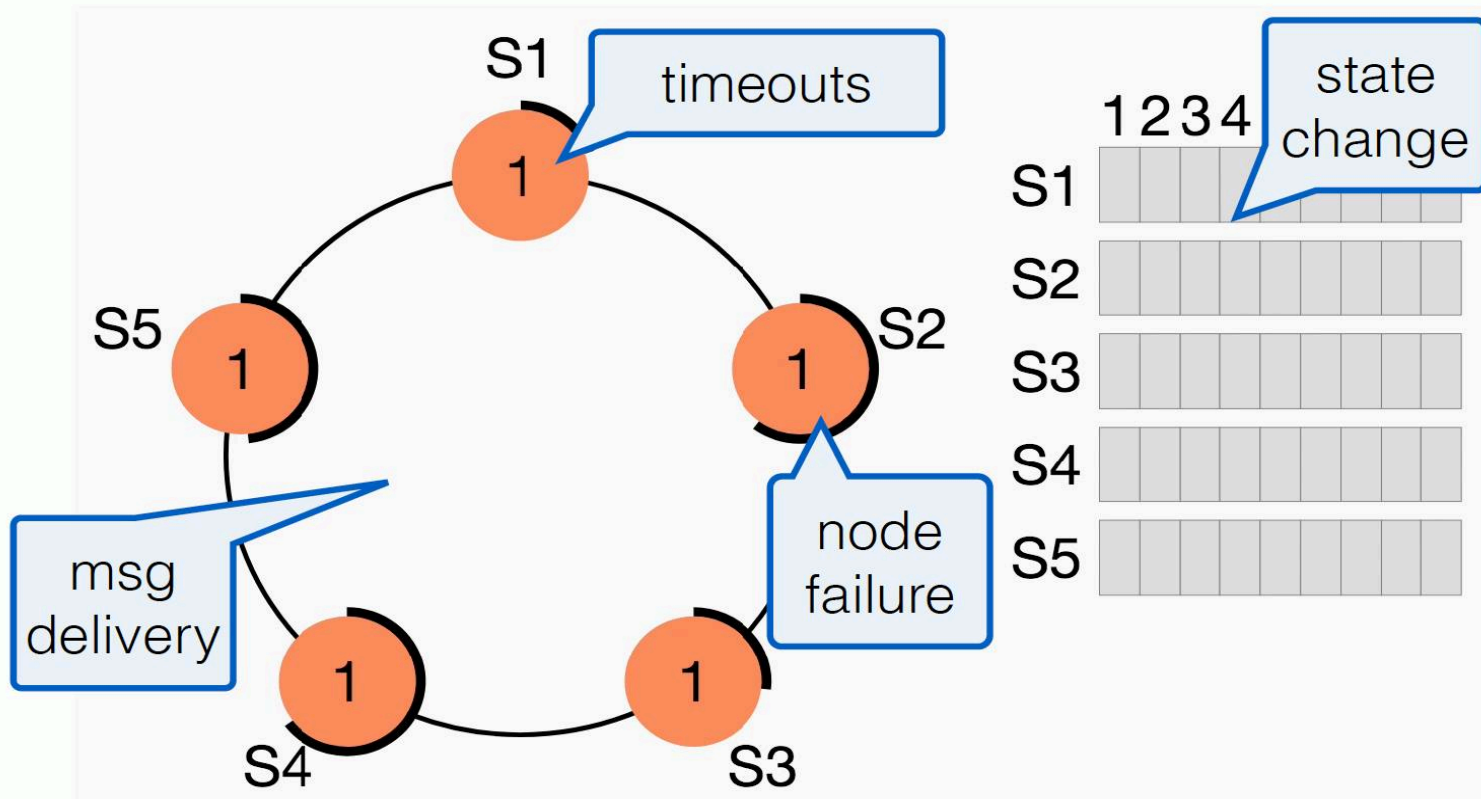
- 2002年4月に起きた、みずほ銀行の大規模な障害は、システム内のネットワーク、第一勧業銀行・富士銀行・日本興業銀行間を結ぶネットワークをコントロールする「リレーコンピュータ」のバグが原因だった。
 - 写真は、2015年7月8日のシステム障害で、創設以来223年間、途絶えたことのなかった取引の中止に追い込まれたニューヨーク証券取引所の前でボーツとする人。
-

The Stock Market Bell Rings, Computers Fail, Wall Street Cringes



分散システムでのプログラミングの難しさを考える

Formalizing distributed systems



分散システムでのプログラミングの難しさを考える

- ❑ 分散システム上のノードは、故障で死んでるかもしれないし（ノード S2）、メッセージの通信がタイムアウトを起こしてメッセージが伝わっていないかもしれない（ノード S1）。またある瞬間をとれば、メッセージは送り出されているのに、まだ受け手のノードには届いていないかもしれない。
 - ❑ 分散システム上で、正しく動くプログラムの設計・開発、なかなか難しいのだ。
 - ❑ こうした問題は、k8s でクラスターを組むときにも、必ず付いて回ると僕は思っている。“Kubernetes Failure Stories” に、色々な事例が集められている。<https://k8s.af/>
-

Kubernetes Failure Stories

A compiled list of links to public failure stories related to Kubernetes. Most recent publications on top.

- [How a simple admission webhook lead to a cluster outage - Jetstack - blog post 2019](#)
 - involved: ValidatingWebhookConfiguration, GKE node auto-repair
 - impact: prolonged downtime of non-prod environment, nodes lost, failed master upgrade
- [Post Mortem: Kubernetes Node OOM - Blue Matador - blog post 2019](#)
 - involved: AWS, SystemOOM, EBS, fluentd-sumologic, no resource requests/limits
 - impact: unknown, Pods killed
- [Kubernetes' dirty endpoint secret and Ingress - Ravelin - blog post 2019](#)
 - involved: GKE, Ingress, replication controller, SIGTERM, "graceful shutdown"
 - impact: occasional 502 errors
- [How a Production Outage Was Caused Using Kubernetes Pod Priorities - Grafana Labs 2019](#)
 - involved: Pod priorities
 - impact: cascading Pod evictions
- [Moving to Kubernetes: the Bad and the Ugly - Xing - ContainerDays EU 2019](#)
 - involved: nginx Ingress, network interrupts, conntrack, frozen CronJob, PLEG, stuck controllers
 - impact: lost requests, response time jumps, not ready nodes
- [Kubernetes Failure Stories, or: How to Crash Your Cluster - Zalando - ContainerDays EU 2019](#)

<https://k8s.af/>

ハードウェア開発

ハードウェア開発の大規模化と形式手法

- ソフトウェアの開発だけでなくハードウェアの開発も、大規模化・複雑化が進んでいる。
 - CPUの設計では、動作の徹底的なシミュレーションは可能だが、最終成果物であるCPUチップに対して、アジャイルのような開発手法はとれるわけではない。
 - 今回のテーマである「形式手法」は、ハードウェアの正しい動作を保証する手段として、ハードウェアの開発では重要な役割を果たしていて、すでに広く用いられていると語られてきた。
 - ただ、実態は、必ずしもそうでなかったようだ。RISC-Vチップの検証を、初めて完全に形式的手法で行うことに成功したチップは、次のように語っている。
-

チツパラの評価

- 「ソフトウェアに関する経験しかない研究者にとっては、この業界がハードウェアに対して行なっている形式的方法というものが、いたるところで制限されたものだということに驚くことになるだろう。
- そこでの形式方法というのは、典型的には、モデルのチェックと充足性の解決に基づいたもので、結局分析は明示的な状態空間の探索に還元される。
- 検証作業は、フルシステムの(比較的)小さなコンポーネント、例えば、プロセッサの浮動小数点ユニットに集中している。確かに、浮動小数点演算の実装アルゴリズムの完全な検証は、それ自体、英雄的な努力を必要とするのだが、これらの個々の結果は、システム全体の定理には組み込まれない。」

SpectreとMeltdown

- 彼の指摘は、昨年来話題になっている、SpectreやMeltdownといった CPUアーキテクチャ設計自体の不備をつく攻撃に対する、CPUの深刻な脆弱性を考えると重要な意味を持っている。



Spectre



Meltdown

投機的実行とSpectre

- 現在のほとんど全てのCPUは「投機の実行 Speculative execution」をサポートしている。投機の実行は、条件付き分岐命令に出会ったとき、条件の計算が終わるのを待たずに、二つの可能な分岐をキャッシュ上で先読みして実行する。条件の計算が終わって、どちらの分岐が選択されるのか確定した時点で、一方の「当たり」のキャッシュの計算結果が採用され、一方の「はずれ」のキャッシュの中身は無視される。
 - Spectreは、キャッシュ上に**一定期間**、トラップを仕掛けた命令がチェックされずに残存することを利用する。これまでのCPUの「検証」では、こうしたセキュリティ上の「大穴」が存在することが見逃されてきたのだ。
-

'WHAT THE F*CK IS GOING ON?' Linus Torvalds explodes at Intel spinning Spectre fix as a security feature

Patches slammed as 'complete and utter garbage' as Chipzilla U-turns on microcode

By [Thomas Claburn](#) and [Kat Hall](#) 22 Jan 2018 at 21:58

244 

SHARE ▼



<http://bit.ly/2IRIwKI>

「それは仕様です。バグでは……」

- 写真は、Spectreに対するインテルのパッチを、激しい言葉で非難するライナス。
彼の怒りを理解するには、日本のIT業界のジョーク、「それはバグではありません。仕様です。」を思い出せばいい。
 - 事実、インテルが提供したパッチは、この投機的実行の不都合を「バグ」ではなく、「セキュリティーの特徴 Security feature」と呼び、オプションでこの「特徴」を使うことも（おいおい！）、使わないこともできるとしたものだっただから。
-

「それは仕様です。バグでは……」

- ❑ 結局、CPUの性能を最大化するための実装が先行していて、それがどういう副作用を持つかのキチンとした検証が不十分だったということ。
 - ❑ ただ、それは、CPUを設計する「仕様」が実際には完全なものではなく、その正当性・妥当性のチェックがキチンと行われていないことを意味する。それでは、CPUアーキテクチャー自体の持つ脆弱性への攻撃(これからもありうること)には、個別にモグラ叩きのように対応するしかなくなる。
 - ❑ 僕は、RISC-Vチップの開発で成功した、CPU開発での「形式的手法」の導入に注目している。彼のプロジェクトについては、第二部で紹介する、
-

Spectre / Meltdown Paper

Google Project Zero

- ❑ Spectre Attacks: Exploiting Speculative Execution

<https://spectreattack.com/spectre.pdf>

- ❑ Meltdown: Reading Kernel Memory from User Space

<https://meltdownattack.com/meltdown.pdf>

- ❑ **Google Project Zero**

<https://googleprojectzero.blogspot.com/>

開発コスト

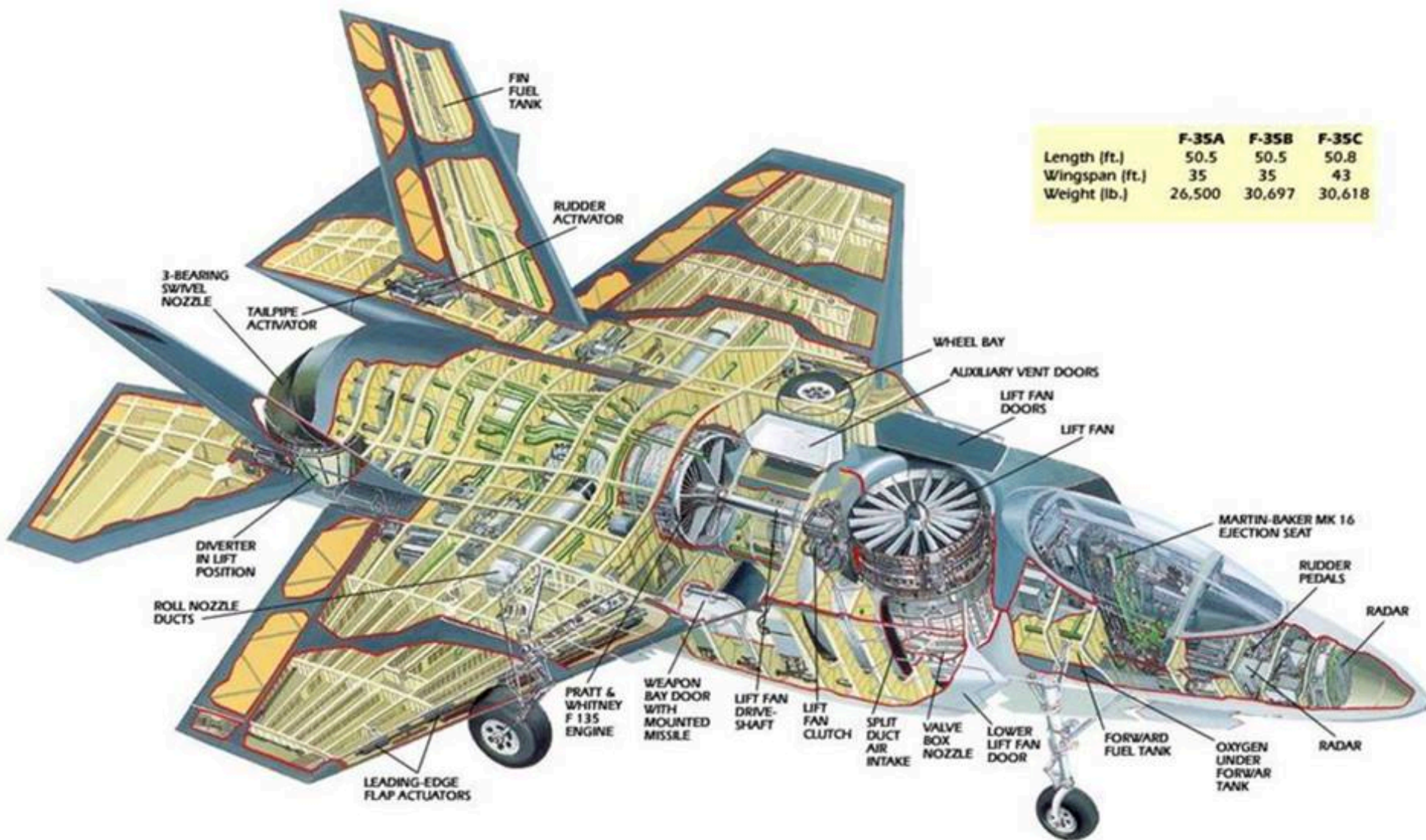
みずほ銀行 MINORI 開発費4,000億円超

- システムの大規模化・複雑化は、開発コストを増大させる。先日正式に稼働を始めたみずほ銀行のシステムMINORIは、開発に4000億円以上を投入したと言われている。もちろん、安かろう、早かろうで、バグですぐにこけるようなシステムを作っていたら、それは企業の価値を毀損することになる。
 - ただ、金融の世界もITの世界も、どんどん変化していく。開発コストの高騰は、変化への迅速な対応への足かせになる可能性がある。
 - 大規模システムでの開発コストの低減が、今後も大きな課題になるのは確実である。
-

軍用機の開発費高騰についてのDARPAの分析

- ここでは、開発コストの高騰を放置すれば、どういう事態になるかを警告した、アメリカ国防総省のDARPAのレポートを紹介したいと思う。直接には、F35のような軍用機の開発コストを分析しています。
- みずほ銀行の4000億円の開発費なんか、可愛いもんだと思うような、かつ、「ほんとかよ！」と思うような、シュールな分析がされている。
- **Formal Model-Based Design & Manufacture: -- A Template for Managing Complexity in Large-Scale Cyber-Physical Systems"**
["https://www.innovation4.cn/library/r16328](https://www.innovation4.cn/library/r16328)
(ブックマークしていた、資料のリンクが切れていて、ググリ直した。DARPAの資料を中国のサイトで読むことになってしまった)

F-35 Joint Strike Fighter (2001)



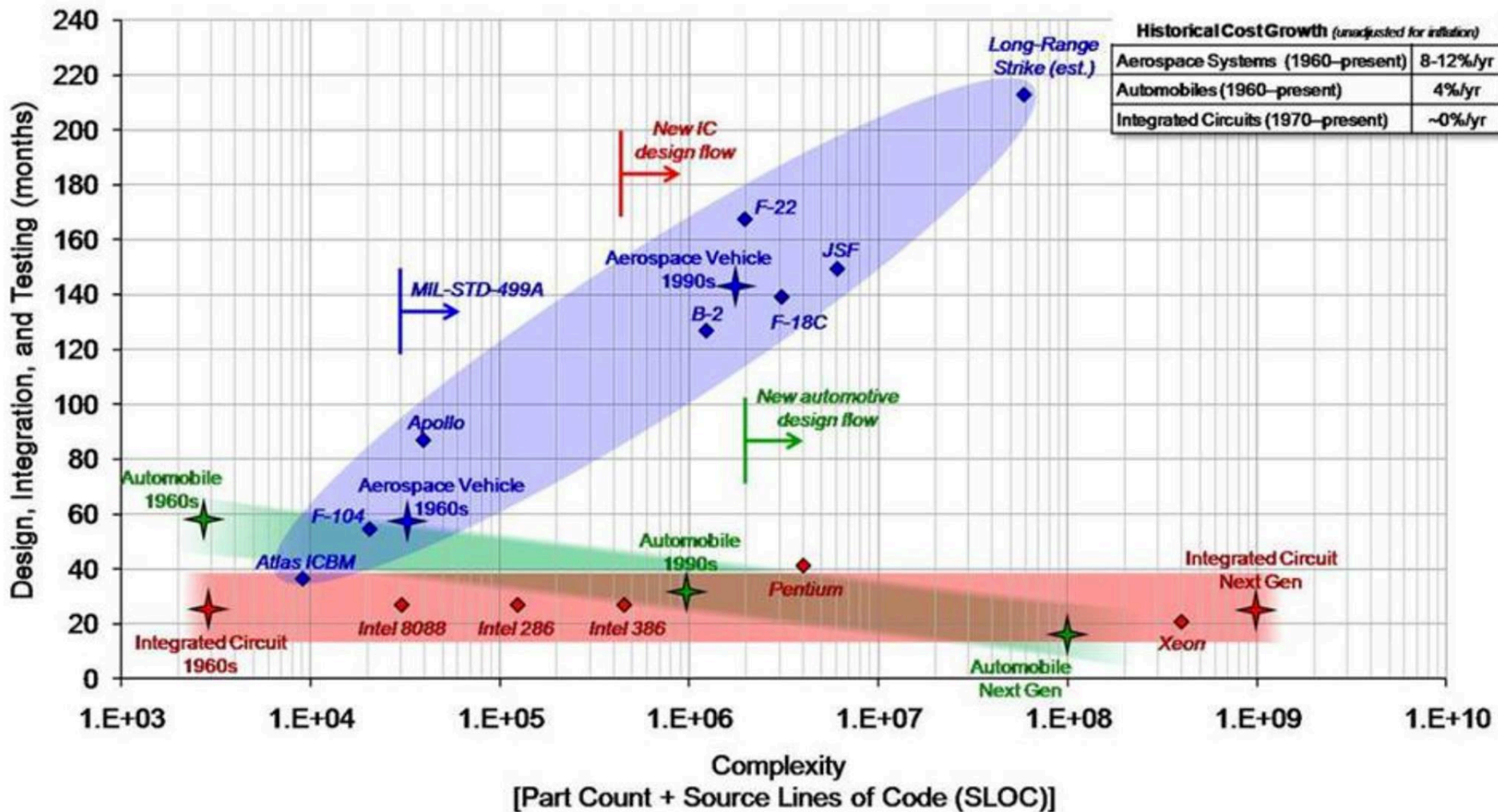
	F-35A	F-35B	F-35C
Length (ft.)	50.5	50.5	50.8
Wingspan (ft.)	35	35	43
Weight (lb.)	26,500	30,697	30,618

CPU・自動車・軍用機の開発コストと特徴

- 次の図を見て欲しい。ここでは、大規模システムとして、三つの分野を取り上げている。CPUの開発(ピンク)、自動車の開発(グリーン)、そして軍用機の開発(パープル)である。
 - 横軸は、部品数・ソースコードの量で計ったシステムの「複雑さ」、縦軸は、設計・統合・テストが終わるまでの「開発時間」。(開発時間は、コストに直結する。)
 - CPUの開発では、複雑さが年々増大しているものの、開発期間はほぼ一定で推移している。自動車では、年々、開発期間が短くなっている傾向が見れる。
 - 軍用機の開発は、それらとは全く違っている。紫の領域の右上トップにある“Long-Range Strike”というのは、F35のことである。
-



Structural & software complexity

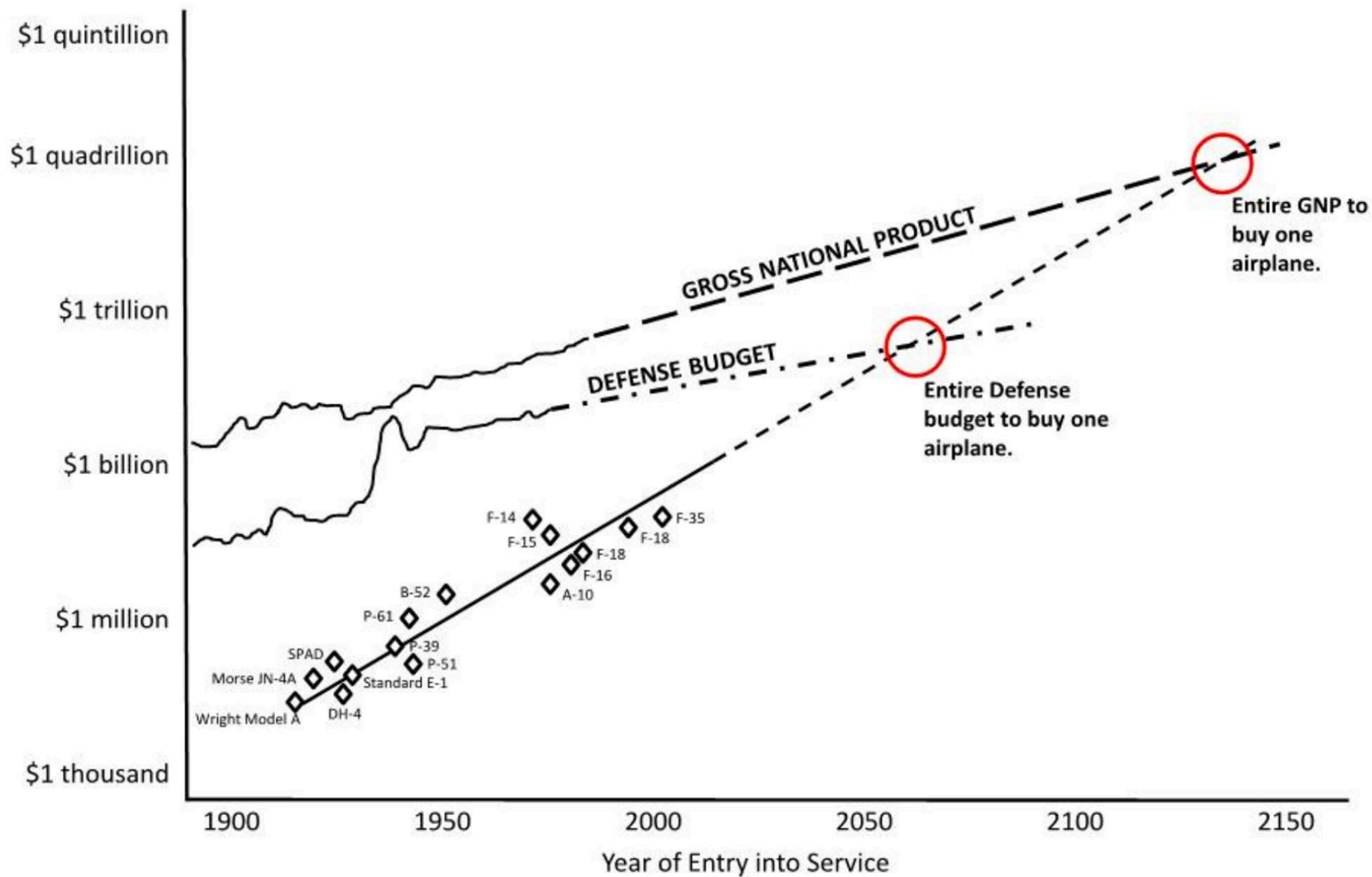


Paul Eremenkoの予想

- 次の図の予想は驚異的なものだ。
 - このまま軍用機開発のコストの増大が続くなら、21世紀の後半には、軍用機一機を買うために、アメリカの国防予算全てを注ぎ込まなければいけなくなり、22世紀の中頃には、アメリカのGNP全てを一機の購入にあてなければいけなくなる！
 - このレポートをDARPAで作成した、Paul Eremenco は、のちにGoogleに移り Project Araを主導し、現在は AirBus にいる。とても面白い人だ。
-



Cost growth



15
9月

Thick smoke could be seen
rising from the Aramco
refinery in Abqaiq.

Guardian

【事案分析】サウジアラビア 石油施設へのドローン攻撃



SUBSCRIBE

Houthi drones attacked the world's largest oil processing facility in Saudi Arabia and a major oilfield operated by Saudi Aramco early on Saturday, sparking a huge fire at a



高速化

人間と機械の「時間のギャップ」

- バグのない大規模システムの開発の難しさの背景を考える時、我々は、システムの大規模化・分散化を中心とする複雑化に、主に注意が向く。
 - それはある意味当然なのだが、もう一つの背景として、CPUの高速化の問題がある。クロック 1 nano秒でドライブされる CPUは、可能的には一秒間に 100万個の命令を実行できる。もちろん、それは悪いことではない。
 - 問題は、機械の中で流れる時間と、人間の中で流れる時間には、大きなギャップがあるということだ。それは、言い換えると機械が実行可能な行動の数と、人間が実行可能な行動の数には、大きなギャップがあるということだ。
-

プログラムとデバッグ

- それは、簡単なコードを書くだけで機械がそれ以上にたくさんの仕事をしてくれるということなのだから、我々は大いに助けられている。それは悪いことではない。
- ただ、CPUの高速化がもたらした、人間と機械の「時間のギャップ」は、時には問題を起こす。
- 例えば、何か「問題」が起きて、それを直そうとするデバッグの際、機械が実行した過程を、ワンステップずつトレースするのは、人間の力では事実上不可能になることがある。我々にとってやりやすいのは、「人間の時間」の中で書かれた「人間のコード」をトレースすることである。
- 逆に、プログラムを書く際、人間は、そのプログラムの背後で機械が実際に実行していることを完全に理解しているわけではない。そこには、いろいろな誤りが潜むことになる。

“Latency as an Effect”

“Principles of Reactive Programming”

Coursera lecture by Erik Meijer

<https://class.coursera.org/reactive-001/lecture/51>

CPU上での処理時間

execute typical instruction	1/1,000,000,000 sec = 1 nanosec
fetch from L1 cache memory	0.5 nanosec
branch misprediction	5 nanosec
fetch from L2 cache memory	7 nanosec
Mutex lock/unlock	25 nanosec
fetch from main memory	100 nanosec
send 2K bytes over 1Gbps network	20,000 nanosec
read 1MB sequentially from memory	250,000 nanosec
fetch from new disk location (seek)	8,000,000 nanosec
read 1MB sequentially from disk	20,000,000 nanosec
send packet US to Europe and back	150 milliseconds = 150,000,000 nanosec

<http://norvig.com/21-days.html#answers>

ネットワーク・プログラミングと その背後で起きること

```
val socket = Socket()  
val packet = socket.readFromMemory()  
// このsocket命令の背後で起きることは？  
// 例外が発生しない時には、次の処理へ  
val confirmation = socket.sendToEurope(packet)  
// この send 命令の背後で起きることは？  
// 例外が発生しない時には、次の処理へ
```

ネットワーク・プログラミングと その背後で起きること（実行時間ベース）

```
val socket = Socket()
val packet = socket.readFromMemory()
// 50,000 ナノ秒ブロックする
// 例外が発生しない時には、次の処理へ
val confirmation = socket.sendToEurope(packet)
// 150,000,000 ナノ秒ブロックする
// 例外が発生しない時には、次の処理へ
```

1 ナノ秒を 1 秒だと思うと

execute typical instruction	1 second
fetch from L1 cache memory	0.5 seconds
branch misprediction	5 seconds
fetch from L2 cache memory	7 seconds
Mutex lock/unlock	½ minute
fetch from main memory	1½ minutes
send 2K bytes over 1Gbps network	5½ hours
read 1MB sequentially from memory	3 days
fetch from new disk location (seek)	13 weeks
read 1MB sequentially from disk	6½ months
send packet US to Europe and back	5 years

SpectreやMeltdownが利用するタイムギャップ

1 ナノ秒を 1 秒だと思おうと



execute typical instruction	1 second
fetch from L1 cache memory	0.5 seconds
branch misprediction	5 seconds
fetch from L2 cache memory	7 seconds
Mutex lock/unlock	½ minute
fetch from main memory	1½ minutes
send 2K bytes over 1Gbps network	5½ hours
read 1MB sequentially from memory	3 days
fetch from new disk location (seek)	13 weeks
read 1MB sequentially from disk	6½ months
send packet US to Europe and back	5 years

タイムギャップは機械と人間の間にだけあるわけではない。
機械の中にもタイムギャップはある！

分散コンピューティングの難しさのタイムギャップ

1 ナノ秒を 1 秒だと思おうと



execute typical instruction	1 second
fetch from L1 cache memory	0.5 seconds
branch misprediction	5 seconds
fetch from L2 cache memory	7 seconds
Mutex lock/unlock	½ minute
fetch from main memory	1½ minutes
send 2K bytes over 1Gbps network	5½ hours
read 1MB sequentially from memory	3 days
fetch from new disk location (seek)	13 weeks
read 1MB sequentially from disk	6½ months
send packet US to Europe and back	5 years

1ナノ秒を1秒だとした場合の実行時間

```
val socket = Socket()
val packet = socket.readFromMemory()
// 3日間ブロックする
// 例外が発生しない時には、次の処理へ
val confirmation = socket.sendToEurope(packet)
// 5年間ブロックする
// 例外が発生しない時には、次の処理へ
```

12 months to walk coast-to-coast
3 months to swim across the Atlantic
3 months to swim back
12 months to walk back



Humans are twice as fast as computers!

開発ツールは信頼できるか？

Csmith が明らかにしたもの

- “We created a tool that generates random C programs, and then spent two and a half years using it to find compiler bugs. So far, we have reported more than **325 previously unknown bugs** to compiler developers. Moreover, every compiler that we tested has been found to crash and **also to silently generate wrong code when presented with valid inputs. (PLDI 2011)**”

“Finding and Understanding Bugs in C Compilers”

“Finding and Understanding Bugs in C Compilers”

Xuejun Yang, Yang Chen, Eric Eide, John Regehr
2011年

[http://www.cs.utah.edu/~regehr/papers/pldi11-
preprint.pdf](http://www.cs.utah.edu/~regehr/papers/pldi11-preprint.pdf)

Finding and Understanding Bugs in C Compilers

Xuejun Yang Yang Chen Eric Eide John Regehr

University of Utah, School of Computing
{jxyang, chenyang, eeide, regehr}@cs.utah.edu

Abstract

Compilers should be correct. To improve the quality of C compilers, we created Csmith, a randomized test-case generation tool, and spent three years using it to find compiler bugs. During this period we reported more than 325 previously unknown bugs to compiler developers. Every compiler we tested was found to crash and also to silently generate wrong code when presented with valid input. In this paper we present our compiler-testing tool and the results of our bug-hunting study. Our first contribution is to advance the state of the art in compiler testing. Unlike previous tools, Csmith generates programs that cover a large subset of C while avoiding the undefined and unspecified behaviors that would destroy its ability to automatically find wrong-code bugs. Our second contribution is a collection of qualitative and quantitative results about the bugs we have found in open-source C compilers.

Categories and Subject Descriptors D.2.5 [Software Engineering]: Testing and Debugging—testing tools; D.3.2 [Programming Languages]: Language Classifications—C; D.3.4 [Programming Languages]: Processors—compilers

```
1  int foo (void) {  
2      signed char x = 1;  
3      unsigned char y = 255;  
4      return x > y;  
5  }
```

Figure 1. We found a bug in the version of GCC that shipped with Ubuntu Linux 8.04.1 for x86. At all optimization levels it compiles this function to return 1; the correct result is 0. The Ubuntu compiler was heavily patched; the base version of GCC did not have this bug.

We created Csmith, a randomized test-case generator that supports compiler bug-hunting using differential testing. Csmith generates a C program; a test harness then compiles the program using several compilers, runs the executables, and compares the outputs. Although this compiler-testing approach has been used before [6, 16, 23], Csmith’s test-generation techniques substantially advance the state of the art by generating random programs that are expressive—containing complex code using many C language features—while also ensuring that every generated program has a

CompCert

コンパイラーの形式的検証

- CompCert プロジェクトは、ミス・コンパイルの問題に対する、根本的で数学的に根拠を持つ解決への前進である。すなわち、コンパイラー自身に形式的で、ツールに支援された検証を行ったのである。

コンパイラーのソースコード自身に、証明技術のプログラムを適用することで、我々は、数学的な確実さを持って、コンパイラーによって生成された実行可能なコードは、Cプログラムのソースによって指定されたセマンティックスと、正確に同じように振る舞うことを証明できるのである。それゆえ、ミス・コンパイルの全ての危険は一掃される。

CompCert

Semantic preservation theorem:

CompCertについて、我々は、Coq証明支援システムの助けを借りて、次の定理を証明した。

全てのソースプログラムSとコンパイラーが生成したコードCに対して、コンパイラーがソースSに適用されて、コンパイル・エラーを報告することなくコードCを生成するのなら、Cの観察可能な振る舞いは、Sの可能な観測可能な振る舞いの一つである。

C コンパイラー CompCertは、このことを保証する。

先の論文でのCompCertの評価

- The striking thing about our CompCert results is that the middleend bugs we found in all other compilers are absent. **As of early 2011, the under-development version of CompCert is the only compiler we have tested for which Csmith cannot find wrong-code errors.** This is not for lack of trying: we have devoted about six CPU-years to the task. The apparent unbreakability of CompCert supports a strong argument that developing compiler optimizations within a proof framework, where safety checks are explicit and machine-checked, has tangible benefits for compiler users. **(PLDI 2011)**

2019 CWE Top 25 Most Dangerous Software Errors

https://cwe.mitre.org/top25/archive/2019/2019_cwe_top25.html

Common Weakness Enumeration

Introduction

The Common Weakness Enumeration (CWE™) Top 25 Most Dangerous Software Errors (CWE Top 25) is a demonstrative list of the most widespread and critical weaknesses that can lead to serious vulnerabilities in software. These weaknesses are often easy to find and exploit. They are dangerous because they will frequently allow adversaries to completely take over execution of software, steal data, or prevent the software from working. The CWE Top 25 is a community resource that can be used by software developers, software testers, software customers, software project managers, security researchers, and educators to provide insight into some of the most prevalent security threats in the software industry.

Below is a brief listing of the weaknesses in the 2019 CWE Top 25, including the overall score of each.

Rank	ID	Name	Score
[1]	CWE-119	Improper Restriction of Operations within the Bounds of a Memory Buffer	75.56
[2]	CWE-79	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')	45.69
[3]	CWE-20	Improper Input Validation	43.61
[4]	CWE-200	Information Exposure	32.12
[5]	CWE-125	Out-of-bounds Read	26.53
[6]	CWE-89	Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')	24.54
[7]	CWE-416	Use After Free	17.94
[8]	CWE-190	Integer Overflow or Wraparound	17.35
[9]	CWE-352	Cross-Site Request Forgery (CSRF)	15.54
[10]	CWE-22	Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')	14.10
[11]	CWE-78	Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')	11.47
[12]	CWE-787	Out-of-bounds Write	11.08
[13]	CWE-287	Improper Authentication	10.78
[14]	CWE-476	NULL Pointer Dereference	9.74
[15]	CWE-732	Incorrect Permission Assignment for Critical Resource	6.33
[16]	CWE-434	Unrestricted Upload of File with Dangerous Type	5.50
[17]	CWE-611	Improper Restriction of XML External Entity Reference	5.48
[18]	CWE-94	Improper Control of Generation of Code ('Code Injection')	5.36
[19]	CWE-798	Use of Hard-coded Credentials	5.12
[20]	CWE-400	Uncontrolled Resource Consumption	5.04
[21]	CWE-772	Missing Release of Resource after Effective Lifetime	5.04
[22]	CWE-426	Untrusted Search Path	4.40
[23]	CWE-502	Deserialization of Untrusted Data	4.30
[24]	CWE-269	Improper Privilege Management	4.23
[25]	CWE-295	Improper Certificate Validation	4.06

CWE-119: Improper Restriction of Operations within the Bounds of a Memory Buffer

```
void host_lookup(char *user_supplied_addr){
    struct hostent *hp;
    in_addr_t *addr;
    char hostname[64];
    in_addr_t inet_addr(const char *cp);

    /*routine that ensures user_supplied_addr is in the right format for conversion */

    validate_addr_form(user_supplied_addr);
    addr = inet_addr(user_supplied_addr);
    hp = gethostbyaddr( addr, sizeof(struct in_addr), AF_INET);
    strcpy(hostname, hp->h_name);
}
```

第二部

Deep Specificationの世界



第二部 **Agenda**

Deep Specificationの世界

- ソフトウェア・エンジニアリングの展望
- Deep Specificationとは何か？
- Deep Specificationのプロジェクト
- 形式的手法によるCPUとネットワークの検証
- 進行中のプロジェクト

ソフトウェア・エンジニアリングの展望

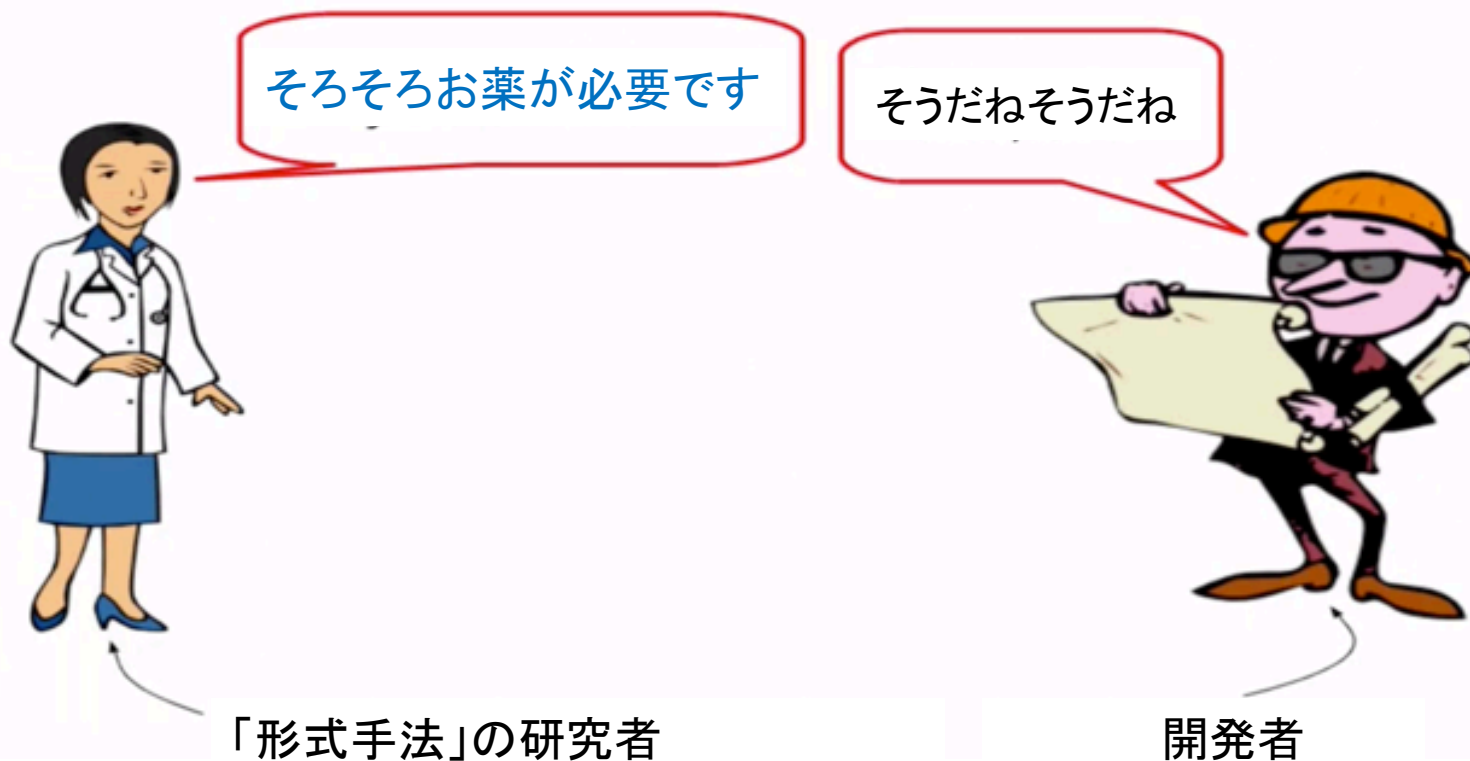
現状の認識

- 現状では、現在利用されているコードのほとんどは、信頼性のチェックを受けていない。
- システムが大規模化・複雑化する中で、そうした状況は、「予期せぬ」バグの温床になる。発生するトラブルはシステムとビジネスの信頼性を毀損し、デバッグの困難さは開発の生産性を低め、開発コストを押し上げる。
- 現在の「テスト・デバッグ・コードレビュー」という開発者中心の検証サイクルを回すだけでは、潜在的な「信頼できないコード」が引き起こす問題に、根本的には対応できない。
- システムの大規模化・複雑化と社会インフラ化は、ソフトウェア・エンジニアリングが、新しい段階に進むことを求めている。

基本的展望

- プログラムに対する信頼は、そのプログラムが正しいという証明の上に基礎付けられる。
 - プログラムは、開発者中心の「テスト・デバッグ・コードレビュー」というサイクルによって検証されるのではなく、検証者がコンピュータを使ってプログラムの正しさを証明することで検証される。
 - 仕様は、その正しさが証明可能な形で、形式的に定義され、プログラムは、そうした仕様(＋最適化)から導出される。
-

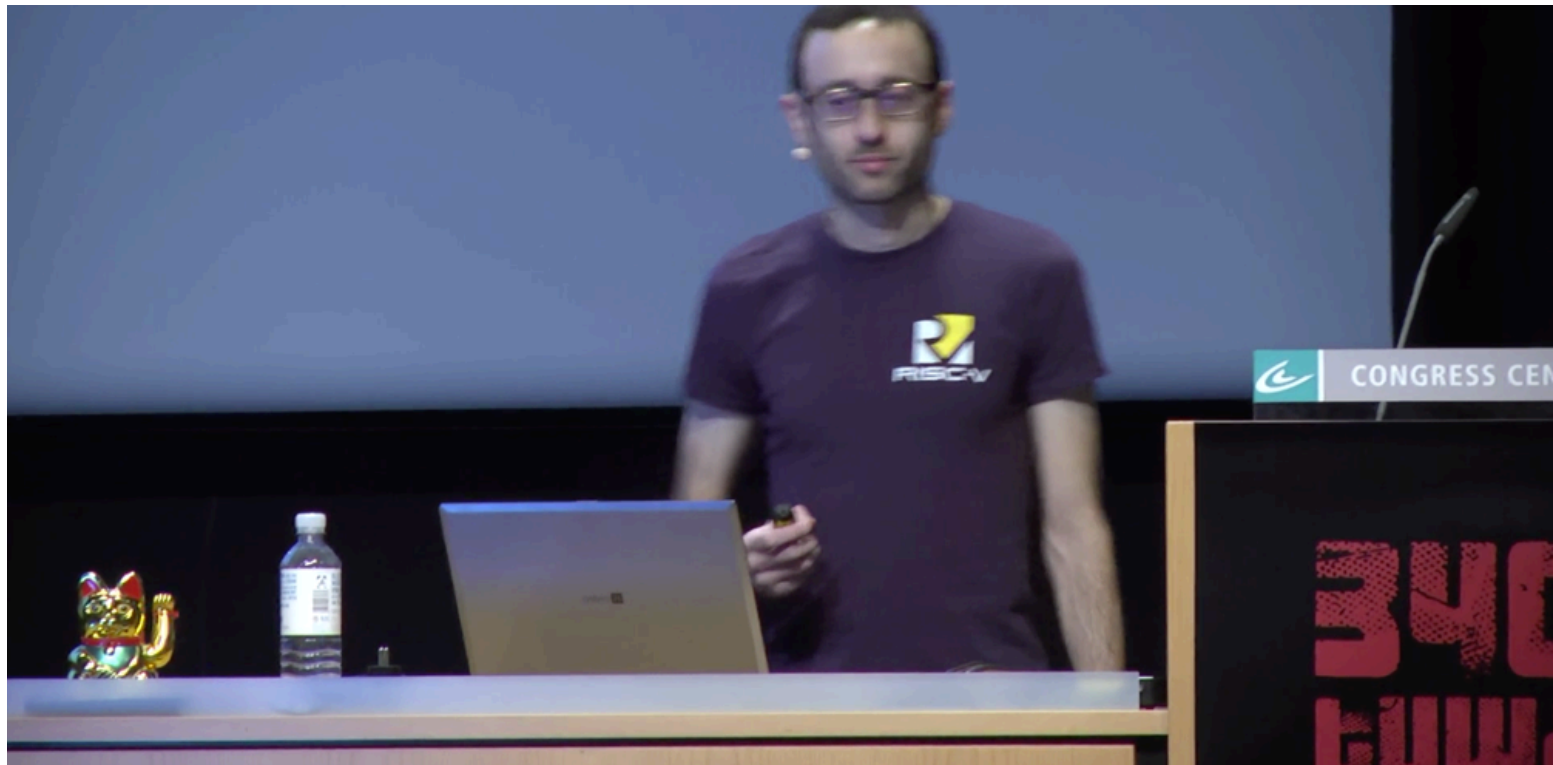
ほとんどのエンジニアは、「機械がチェックする数学的証明」などは、アカデミーの世界の好奇心の対象ぐらいにしか思っていない。



それに対して、「**テスト・デバッグ・コードレビュー**」は、本質的に重要なものとして受け止められている。それは、ほとんど全てのエンジニアの日常の一部になっている。

「もうじき来るぞ。毎日のソフトとハードの開発に、 機械がチェックする数学的証明が」

“Coming Soon: Machine-Checked Mathematical Proofs
in Everyday Software and Hardware Developme”

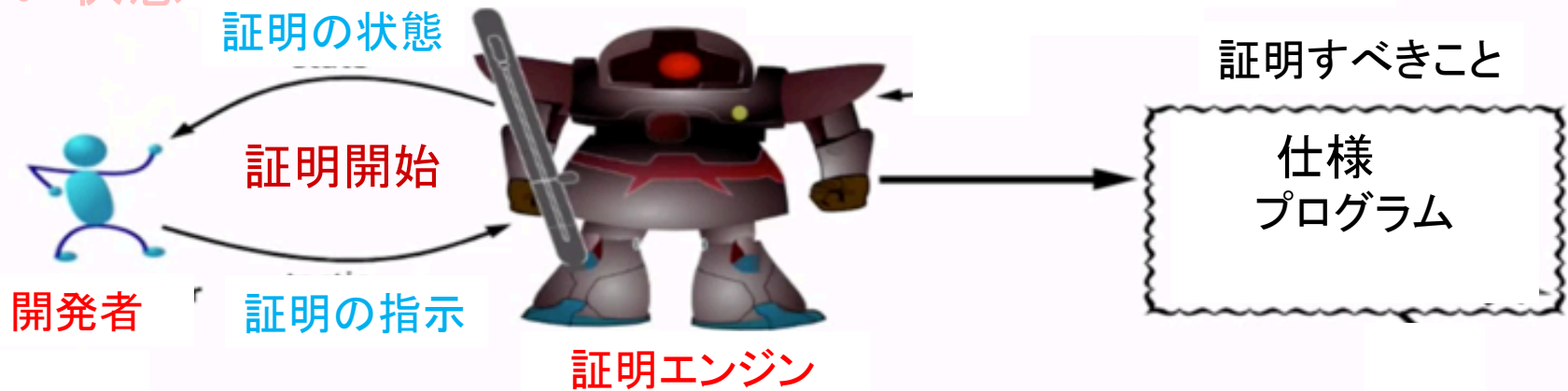


Adam Chlipala <http://bit.ly/2GppRxe>

近未来の開発者

「テスト・デバッグ・コードレビュー」に代わるもの

信頼性の
ない状態



近未来の開発者

「テスト・デバッグ・コードレビュー」に代わるもの

証明成功！



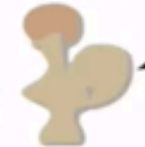
開発者

証明の新しいノウハウ



証明エンジン

証明
チェッカー

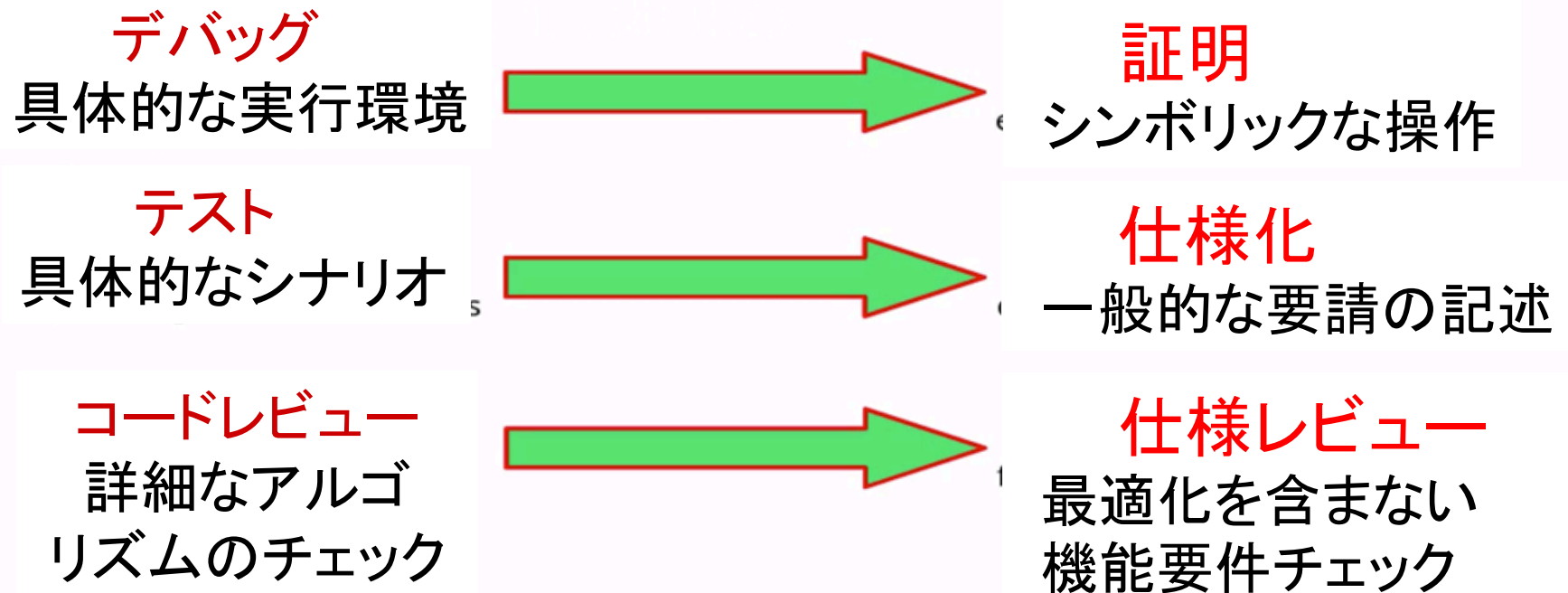


信頼性の
獲得

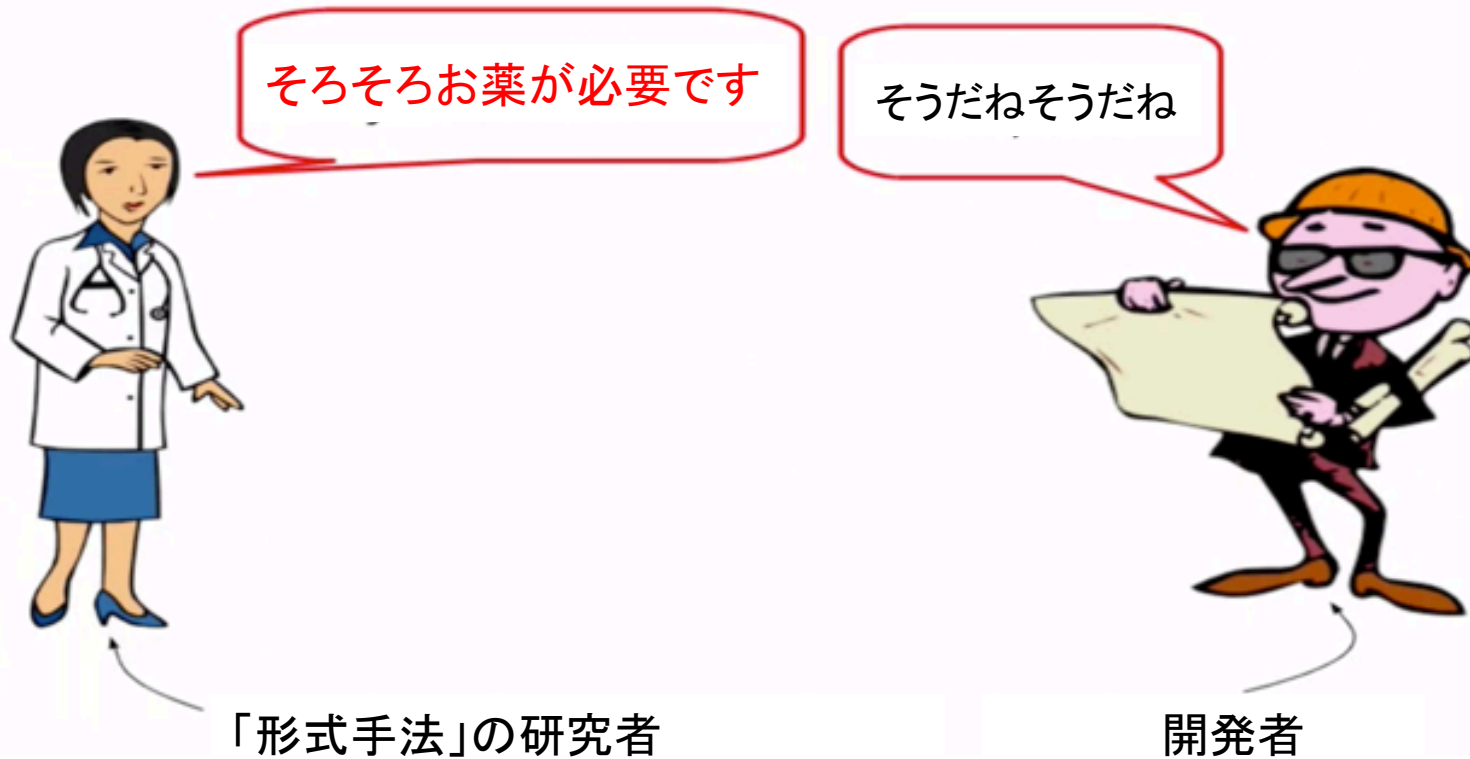
検証された
仕様
プログラム



「デバッグ・テスト・コードレビュー」の 本質的な置き換えが起きる。



「機械がチェックする数学的証明」が、コンピュータのソフトウェアとハードウェアの**開発者の新しい日常**を可能にする。



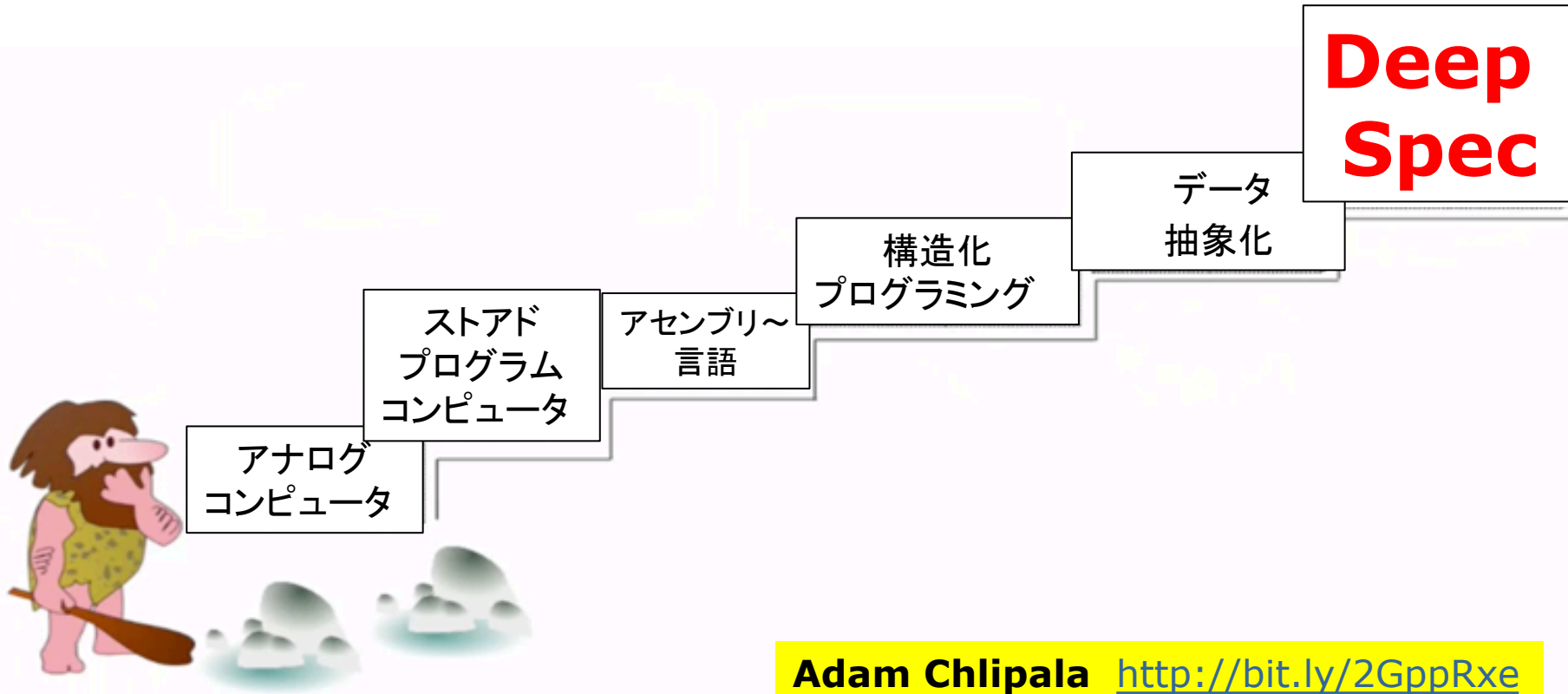
それは、開発に必要な労力を劇的に軽減する可能性を持ち、同時に、システムが正しく安全に動作する保証を増大させる。

これまでも、開発の世界は変わってきた



Adam Chlipala <http://bit.ly/2GppRxe>

次のステップは、Deep Specになる



Adam Chlipala <http://bit.ly/2GppRxe>

Deep Specificationとは何か？

Deep Specificationとは何か？

我々は、他の多くのプロジェクトでも使われている小規模な検証済みコンポーネントから、大規模な検証済みシステムが常に構築されている世界を期待している。

そのために、全産業規模でのソフトウェアとハードウェア・コンポーネントの正式仕様の重要な基礎を明らかにしようとするプロジェクト「Deep Specificationの科学」での我々の取り組みを紹介する。

我々は、異なるチームの作業をまたいで、コンポーネントの正しさを示す強い定理を結びつける、重要な仕様のクラスを同定してきた。それらは、すでにいくつかの実験で使われている。

こうしたスタイルのユニークな利点を一般に広めるために、我々はそれを「**深い仕様 Deep Specification**」と呼んでいる。

<https://royalsocietypublishing.org/doi/pdf/10.1098/rsta.2016.0331>

The Science of Deep Specification

このスタイルのユニークな利点を皆に知ってもらうのを助けるために、我々はそれに、**deep specification**という名を授けた。

我々は、それは、**豊かで(rich)**、**二面を持った(two-sided)**、**形式的で(formal)**、**コンピュータで証明可能な(live)**、**諸仕様**を包含するものだと考えている。

我々の中核チームは、(Coq証明支援システムをベースにして)このコンセプトの実証システムの実験を展開している。その仕様と検証作業は、我々の四つの組織を横断して、大きく分離されたサブ・チームに分割されている。それは、分野横断の原則と効率的な仕様開発のツールにそって、ハードウェア・マイクロアーキテクチャー、コンパイラ、オペレーティング・システム、アプリケーションを包含している。我々は、基礎分野の研究者だけでなく、産業界のユーザーにも、このアプローチへの関心を、拡大しようと考えている。

Deep Specificationが満たすべき条件

- **rich** (複雑なコンポーネントの振る舞いを詳細に記述する)
 - **two-sided** (実装とクライアントの双方に接続する)
 - **formal** (明確なsemanticsを持った数学的な表記で記述されている。型チェッカーや分析・テストツール、自動化されたあるいは機械による証明システム、高度なIDEをサポートしている)
 - **live** (コンピュータでチェック可能な証明を通じて、実装とクライアントのコードと結びついている)
-

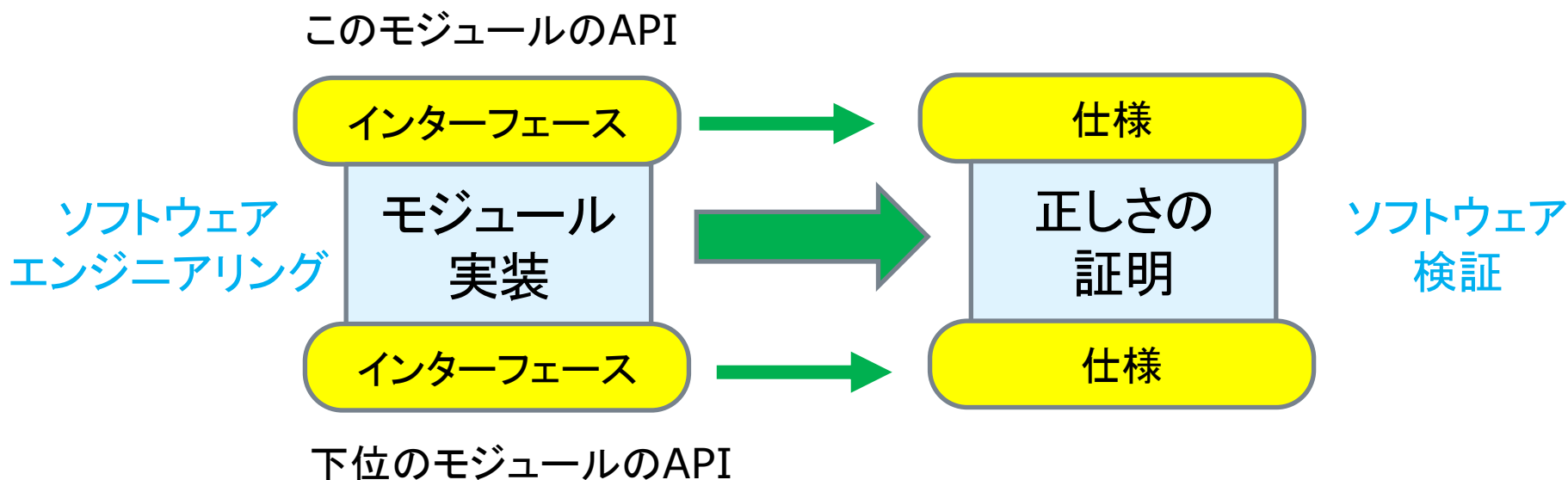
rich, two-sided, formal, live

<https://deepspec.org/page/Research/>

研究の概要

ソフトウェア・エンジニアリングでは、モジュールは、インターフェース(すなわちAPI)を外部に提供し、かつ、モジュール自身は低レベルのインターフェースのクライアントになる。

ソフトウェアの検証では、それぞれのインターフェースに対応した仕様が書かれているので、そのモジュールが仕様に関して正しいかを証明できる。



仕様の記述は十分か？

ソフトウェア・エンジニアリングでは、インターフェースは、しばしばプログラミング言語で書かれている。例えば、C言語での関数のプロトタイプを記したヘッダーファイル、Javaでのクラスの宣言がそうである。

これらは、関数の型を記述することが出来るが、その振る舞いを指定できるわけではない。UMLのようなモデリング言語は、もう少しマシなことが出来るのだが、それでも、そのモジュールが我々にしてほしいことを完全に特徴付けるほど、十分豊かではない。

形式手法のメリット

プログラム検証ロジックは、モジュールの振る舞いを特徴付けるのに、形式論理の表現力をフルに利用することが出来る。

しかし、プログラムのモデルに関して言うと、歴史的なアプローチでは、形式論理の表現力は、プログラムそれ自身には、直接には、またメカニズムとしても結合されておらず、「生きた」ものではずっとなかった。（それに対して、コンパイラーの型チェッカーは、コンパイルするときにはいつでも、APIの仕様上の型が、実装の型とあっているかを確認している。）

「形式的」論理を利用するのはいいことである。なぜなら、そうすることで、自動化された証明と証明チェッカーを使って、我々は、モジュールの実装が仕様を満たしていることを保障し、プログラムの検証ができるからである。

仕様の「二面性 (two-sided)」

しかし、もう一つ考えなければいけない重要なことがある。それは、仕様が「二面」で定義されているかということである。「二面」というのは、仕様が、上位のモジュールに対しても、下位のモジュールに対しても、「二面」で定義されているかということである。

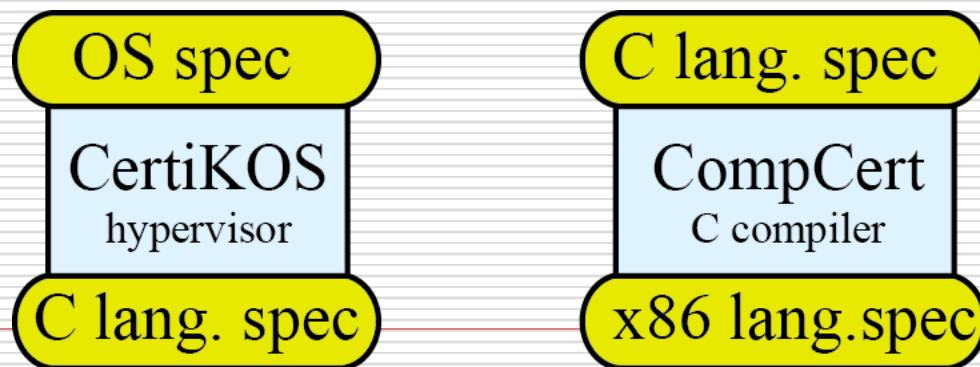
少し説明しよう。我々は、オペレーティング・システムのカーネルやコンパイラやプログラム解析ツールのような重要なコンポーネントの仕様と検証を研究している。例えば、CertiKOSは、Cで実装された管理カーネルである。そこでは、プロセスのアイソレーションや公平なスケジューリングといった諸性質が、オペレーティング・システムの仕様に関して正しいものであることが証明されねばならない。

two-sided

CertiKOS / CompCert 単独の検証？

しかし、この証明は、プログラム言語Cの仕様のクライアントについてのものである。その形式的意味論が与えられなければ、このコードの振る舞いについて推論することはできない。

一方、CompCertは、C言語の形式的意味論に関して正しいと証明された C コンパイラである。同時に、それはそれが吐き出す x86の機械語の形式的意味論に関して正しいと証明されている。



two-sided

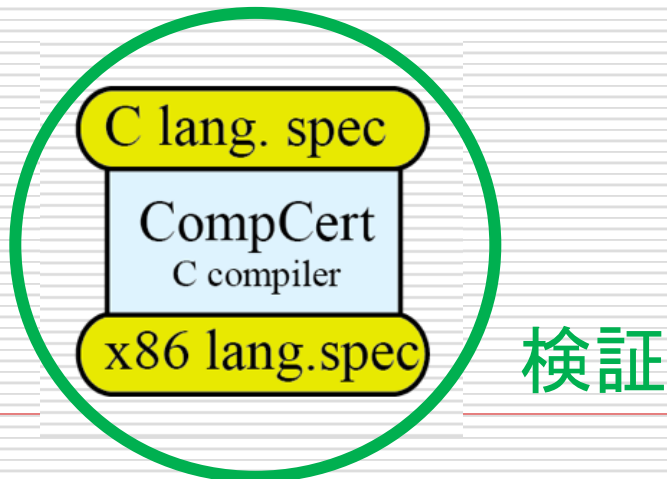
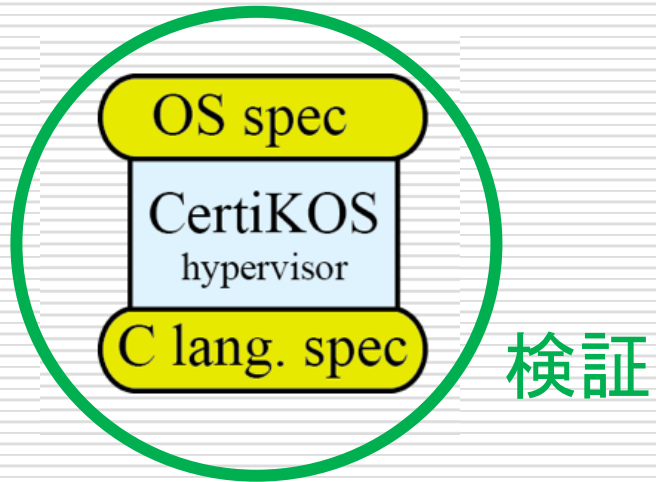
CertiKOSとCompCertとの結合の検証

我々が実践的な場面でよく出くわすことは、例えば、CertiKOSを証明するために利用されるC言語の仕様が、CompCertによって公開されているC言語の仕様とマッチしないことである。

CompCertのC言語の仕様は、CompCertが正しいことを証明するためにデザインされ、それを実行している。それを利用するクライアントの立場から見れば、仕様が不足していたり、仕様の制限が厳しすぎたりする。

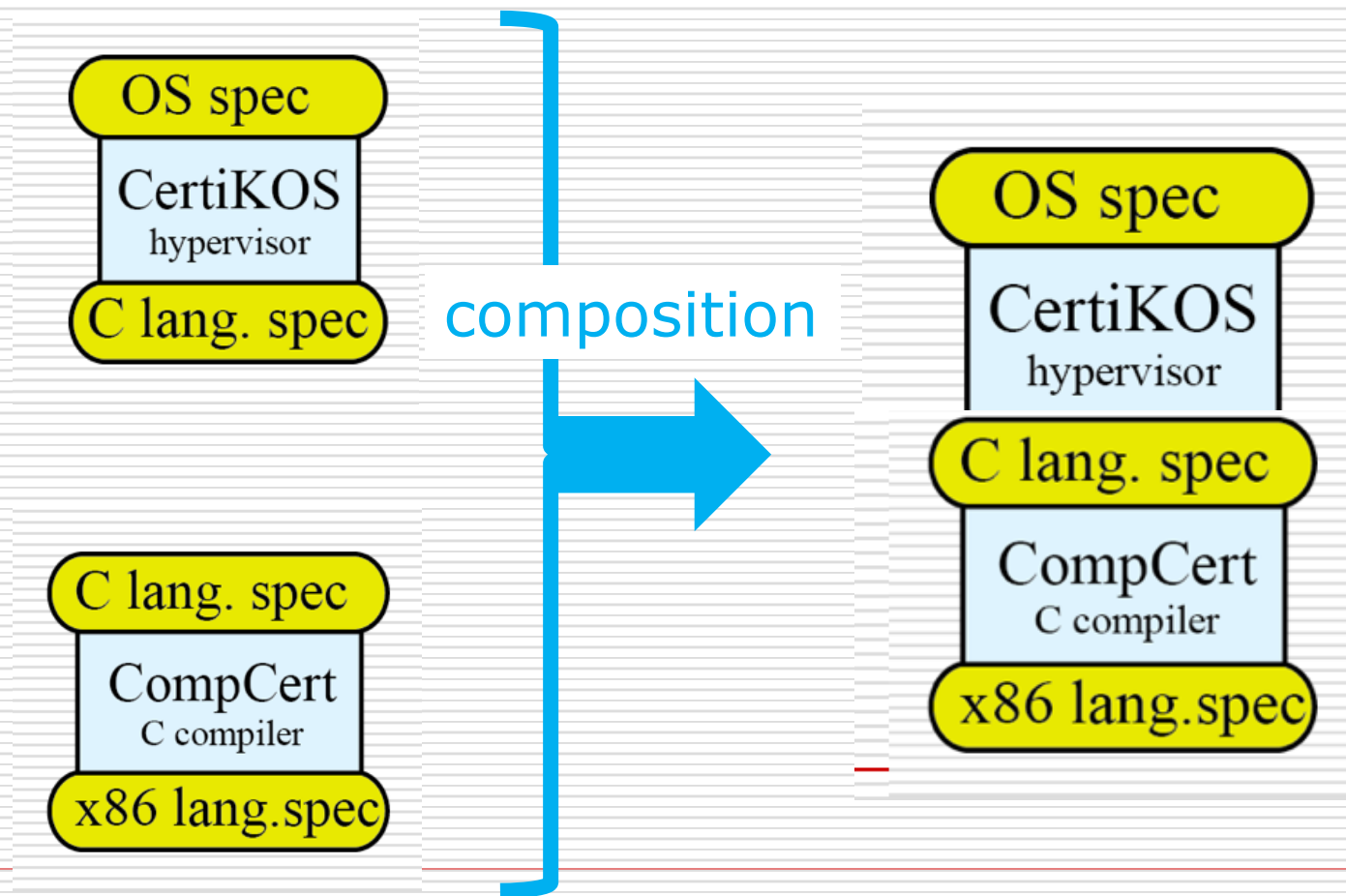
CertiKOS は、CompCertが提供していない仕様のいくつかの特徴を要求している可能性がある。これらのモジュールを一緒にしたときに初めて、C言語の仕様は、二つの面から実際に実現される。そして、このことから我々は、いかに仕様を強化し改善するかを学ぶのである。

個別のモジュールの検証ではなく

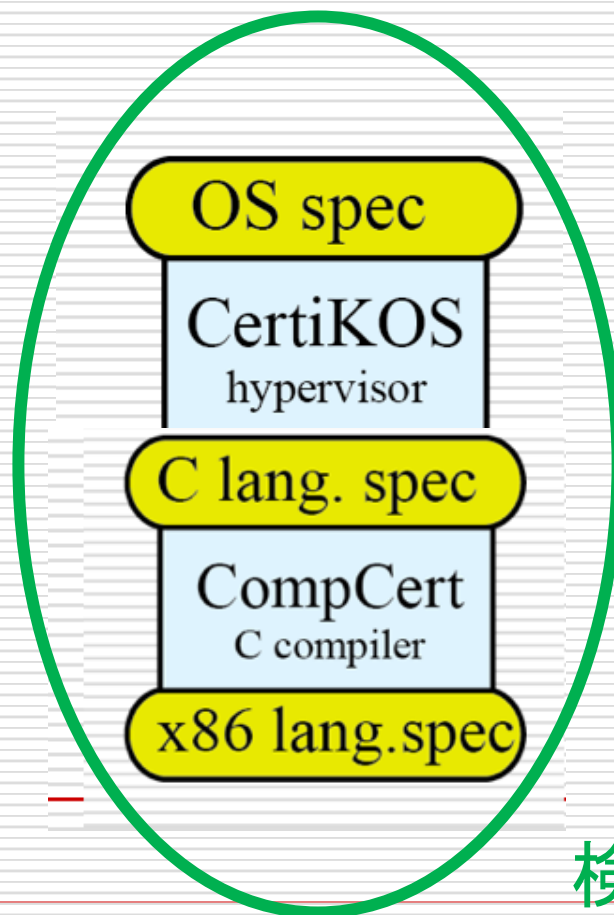
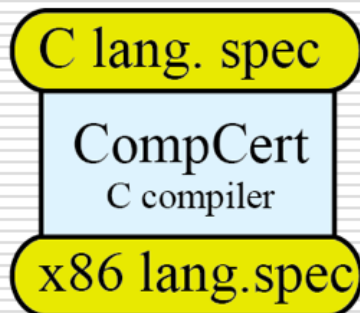
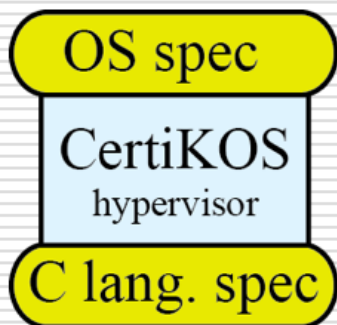


?

個別のモジュールの検証ではなく 結合されたモジュールの検証



個別のモジュールの検証ではなく 結合されたモジュールの検証



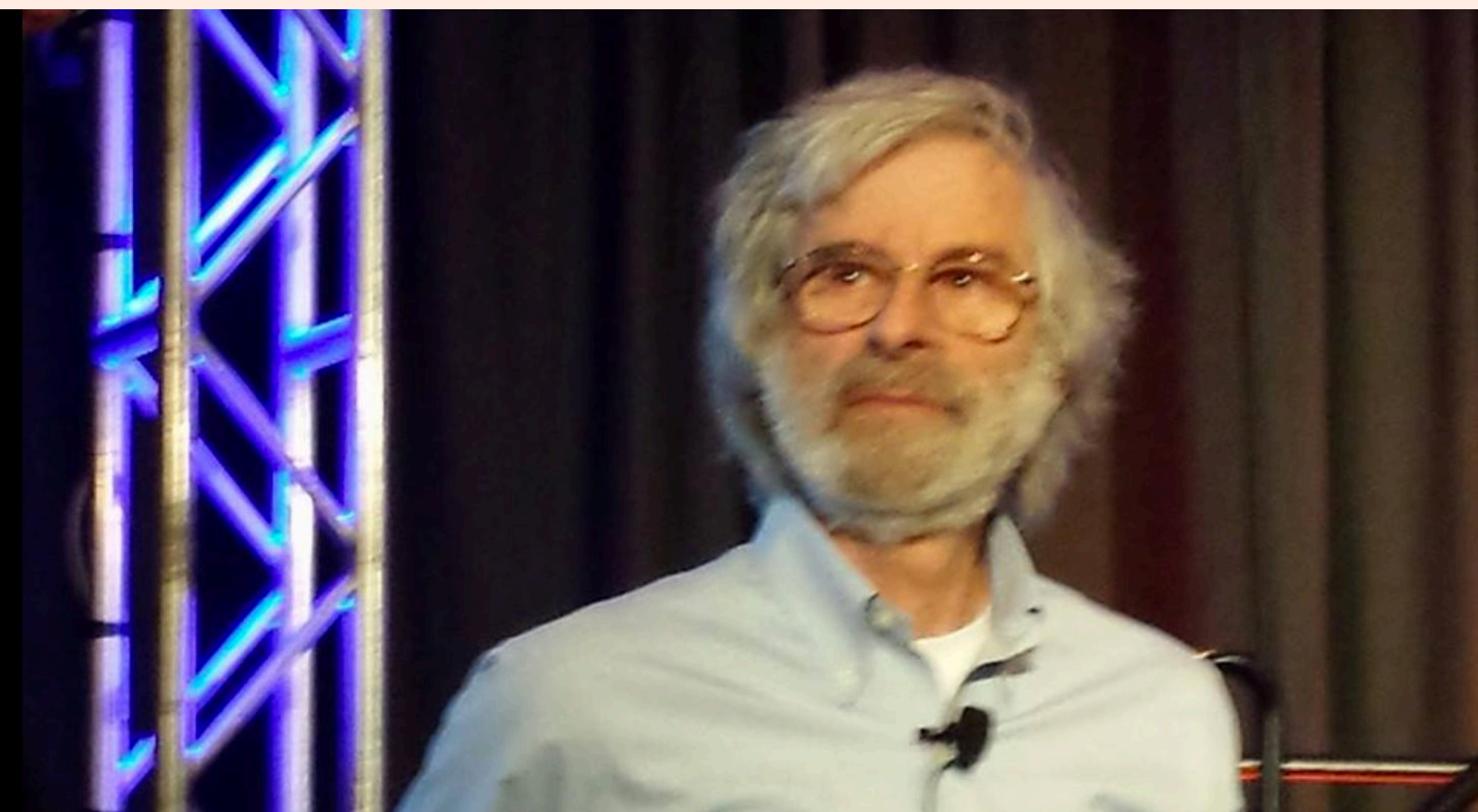
検証

"Composition: A Way to Make Proofs Harder"

Leslie Lamport

1997年

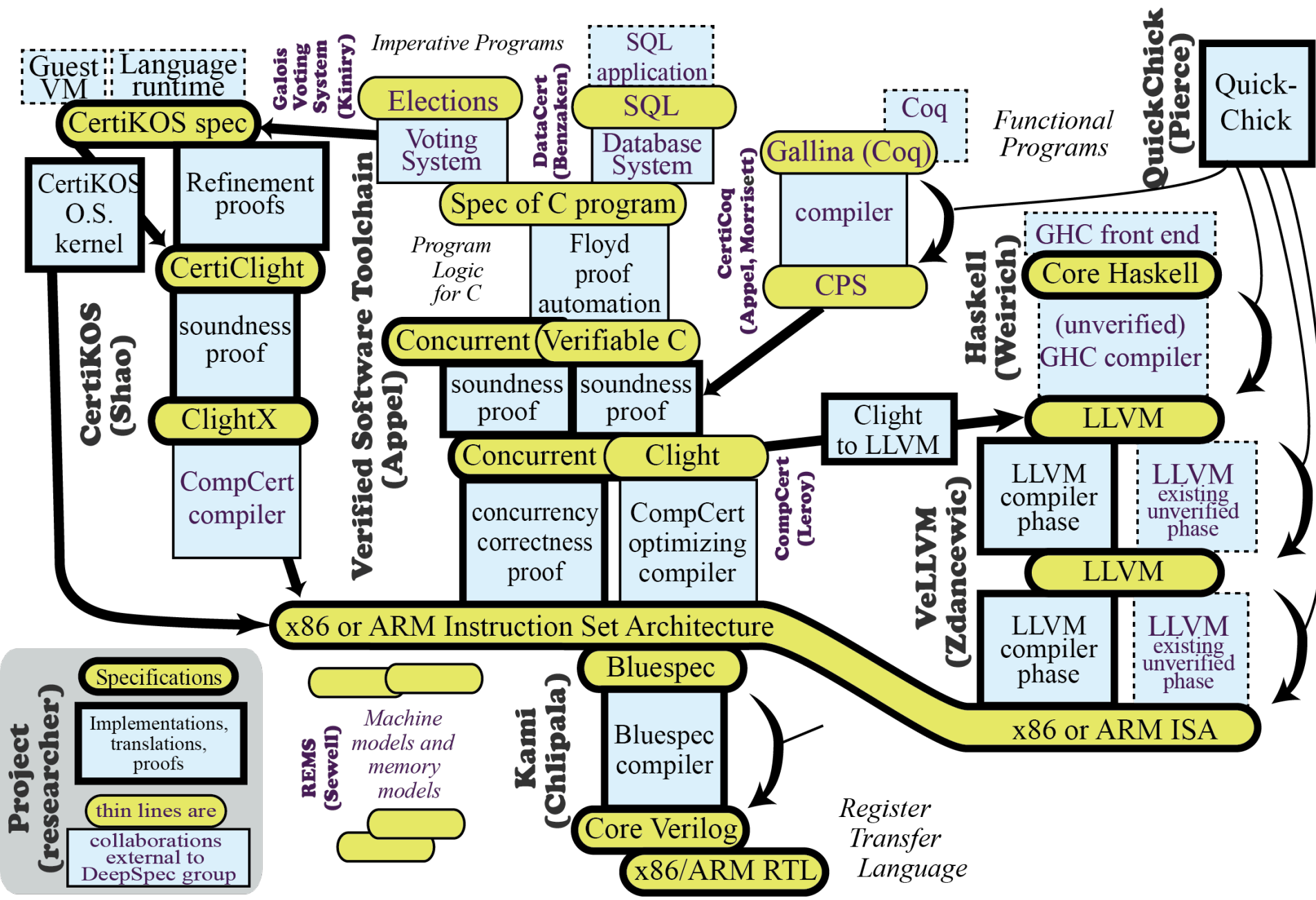
<https://www.cs.cmu.edu/~crary/819-f09/Hoare69.pdf>



"Composition: A Way to Make Proofs Harder"

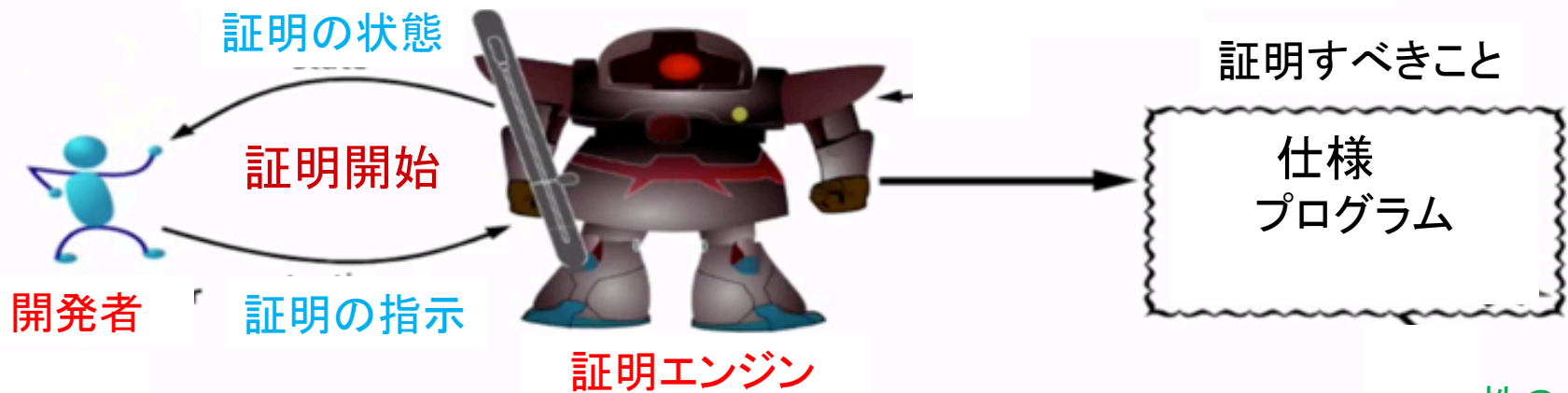
- 「1997年の時点では、不幸な現実には、技術者が自分が構築するシステムについて、形式的に仕様を決めたり推論することがほとんどないということである。オープン・システムの仕様の組み合わせによる構成について推論することが、実践的な関心にのぼることは、次の15年の間には、ありそうもない。」
 - 「現実のシステムの設計にエラーを見つけるという問題に直面した時、我々は何をなすべきか？ 完全な設計というものは、形式的的手法で扱うには、ほとんど全ての場合、あまりに複雑すぎる。」
-

Deep Specification Project



近未来の開発者

「テスト・デバッグ・コードレビュー」に代わるもの





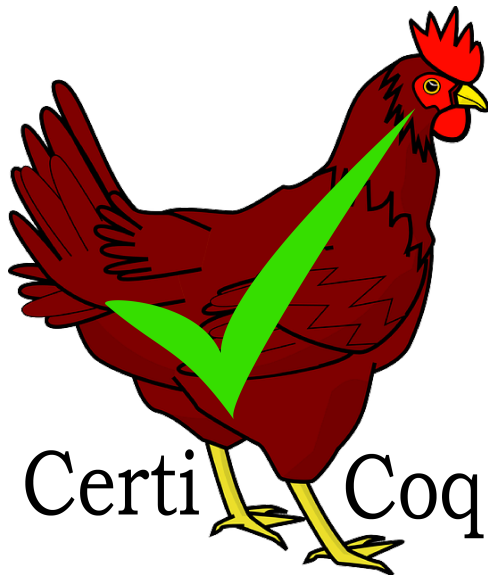
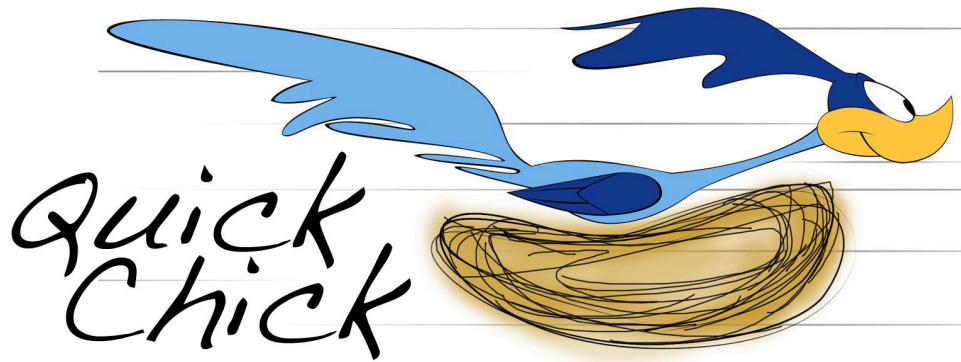
証明エンジン



Deep Specification Project

<https://deepspec.org/page/Project/>





CertiKOS

<http://certikos.org>



The CertiKOS project aims to develop a novel and practical programming infrastructure for constructing large-scale certified system software. By combining recent advances in programming languages, operating systems, and formal methods, we hope to attack the following research questions: (1) what OS kernel structure can offer the best support for extensibility, security, and resilience? (2) which semantic models and program logics can best capture these abstractions? (3) what are the right programming languages and environments for developing such certified kernels? and (4) how to build automation facilities to make certified software development really scale?

Verified Software Toolchain

<http://vst.cs.princeton.edu/>



- The Verified Software Toolchain is a toolchain for verifying C code inside the Coq proof assistant, in a higher-order concurrent, impredicative separation logic ("Verifiable C") that is provably sound w.r.t.~the Clight operational semantics of CompCert. Current work is focused on enhancing automation support ("Floyd"), particularly for well-synchronized concurrent programs executing on processors with weakly cache-coherent memory models, formally connecting the VST to other DeepSpec components (CertiKOS, CertiCoq, Vellum), and verifying high-assurance application components, including mailbox-style message delivery systems and cryptographic libraries.

Kami

<http://plv.csail.mit.edu/kami/>



- We have been developing a candidate official formal specification for RISC-V, which stands a good shot at being ratified soon as such by the RISC-V Foundation. The spec now includes virtual memory and is able to pass all the official RISC-V machine-code tests that aren't marked as specific to particular extensions. We should be able to boot Linux on the specification soon, running as a simulator.
-

Deep Specプロジェクトのその他のリンク

- ❑ **DeepSpec Web Server**

<http://bit.ly/2kHxvM6>

- ❑ **Cryptography**

<https://deepspec.org/entry/Project/Cryptography>

- ❑ **Haskell CoreSpec**

<http://bit.ly/2kHqFpW>

- ❑ **QuickChick**

<https://github.com/QuickChick>

- ❑ **Vellvm**

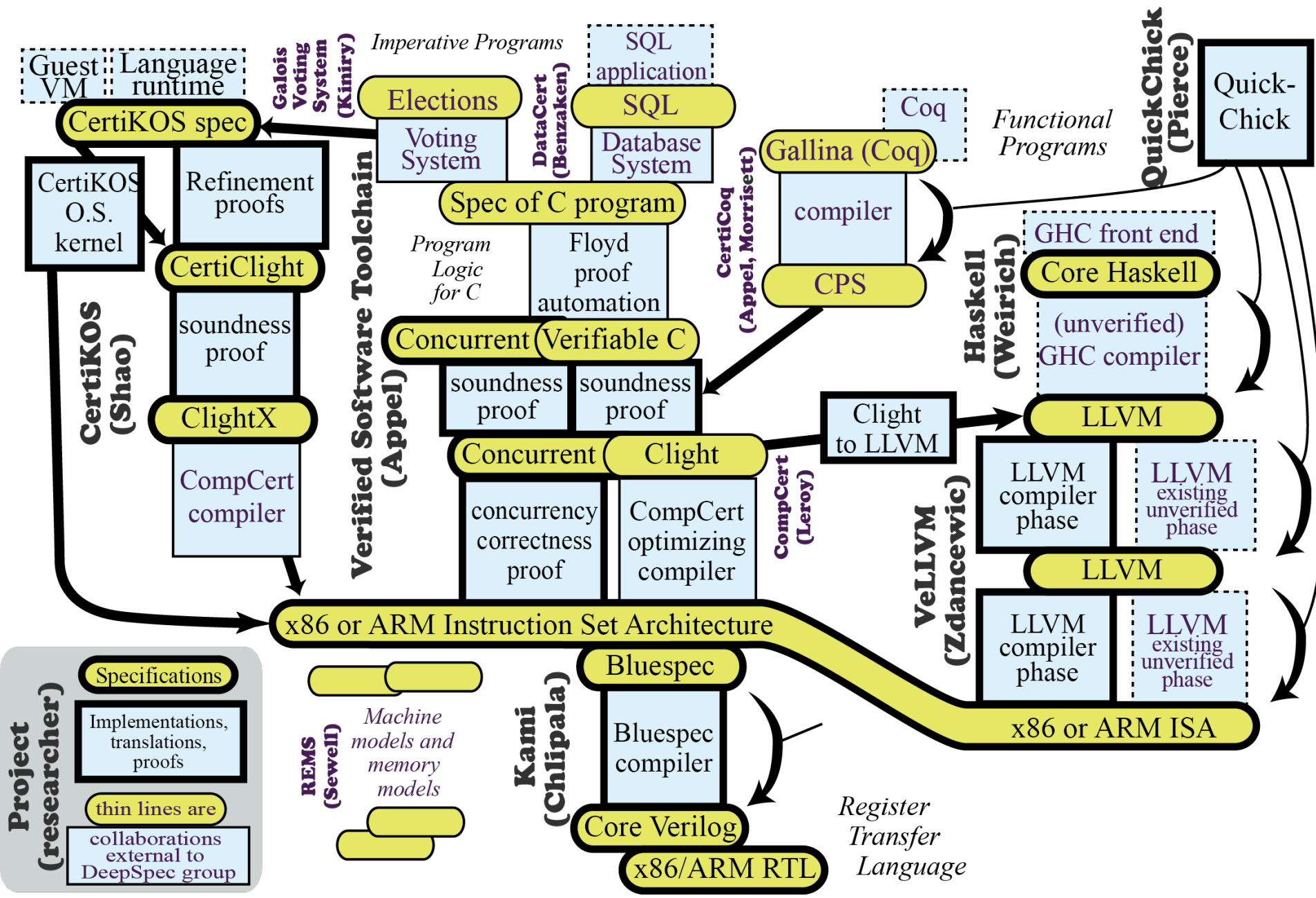
<http://www.cis.upenn.edu/~stevez/vellvm/>

- ❑ **CertiCoq**

<https://www.cs.princeton.edu/~appel/certicoq/>

Deep Specification Network

Deep Specification Network



Deep Specのネットワーク

DeepSpecチームの研究者たちは、それぞれが現在進行中の主要な研究プロジェクトに属している。それらのプロジェクトは、全てお互いに、ある仕様を利用し、他に仕様を提供している。

これらの仕様は、豊かで、現在利用されており、かつ、形式的である。しかし、これらが結合されたときに初めて、それらの仕様が二方向で有効だと保証できるのである。

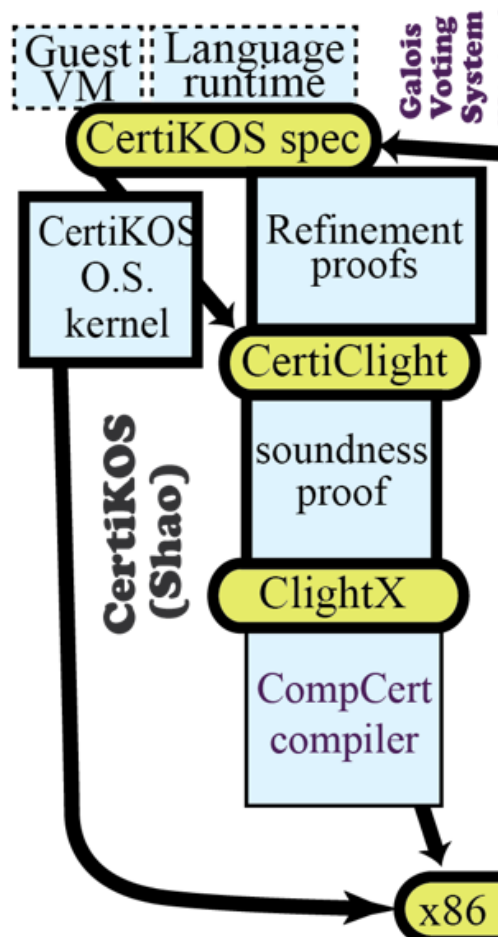
それゆえ、豊かで、生きており、形式的で、二方向を持つ deep specification で結合された、この検証されたソフトウェア コンポーネントのネットワークを見てみることにしよう。

仕様のネットワーク

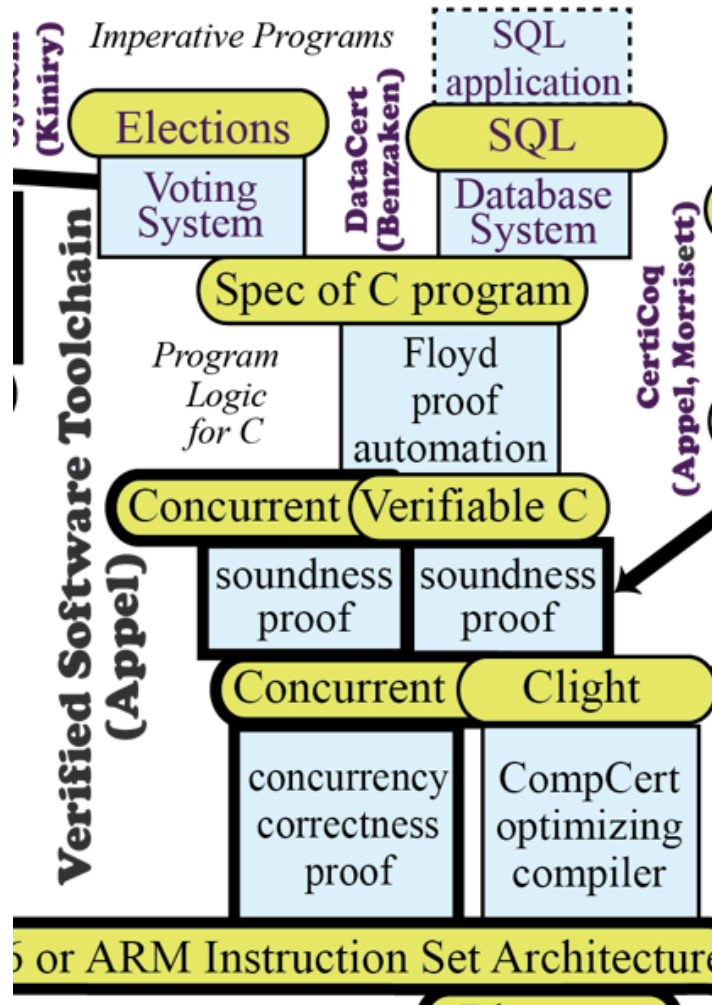
一般的な洞察は、特定のシステムを指定し実装した経験から始まらなくてはならない。この実験の中心的な目標は、セキュリティーにクリティカルなソフトウェア・システムのミドルウェア層から、お互いに関連した仕様の集まりとそれらを結合するなんらかの成果物を作り出すことである。

この図が、このdeep specificationのネットワークとそれらをつなぐもの(実装、変換、証明)を表している。実験の最後に、こうしたビジョンを得た。図の異なる領域の垂直のラベルは、どのプロジェクト研究者がそれらに主要な責任を負ったかを示している。紫のラベルのコンポーネントは、外部の協力者を示している。

CertiKOS (Shao)

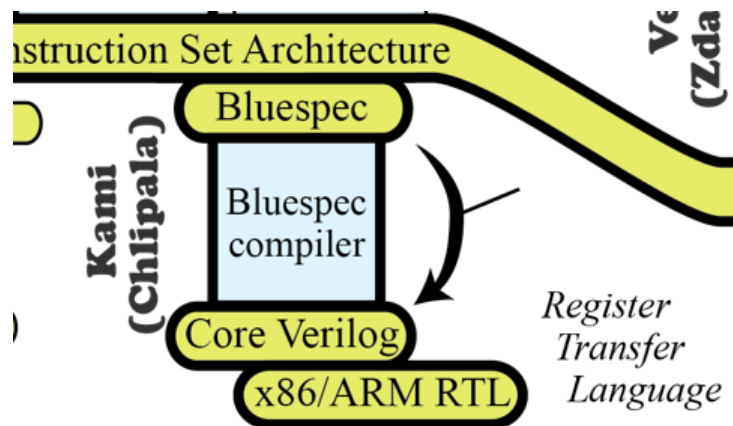


VST - Verified Software ToolChain (Appel)

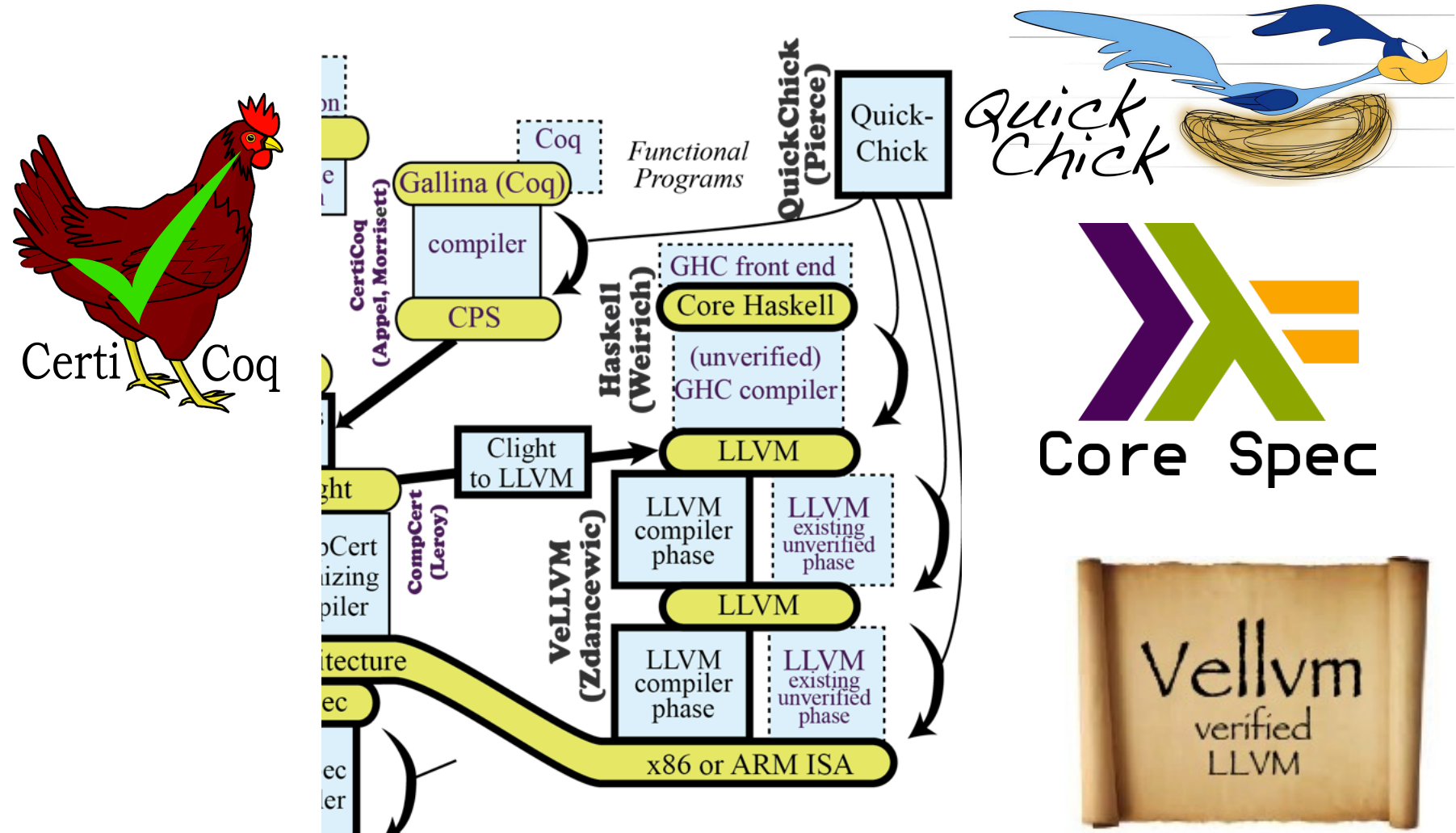


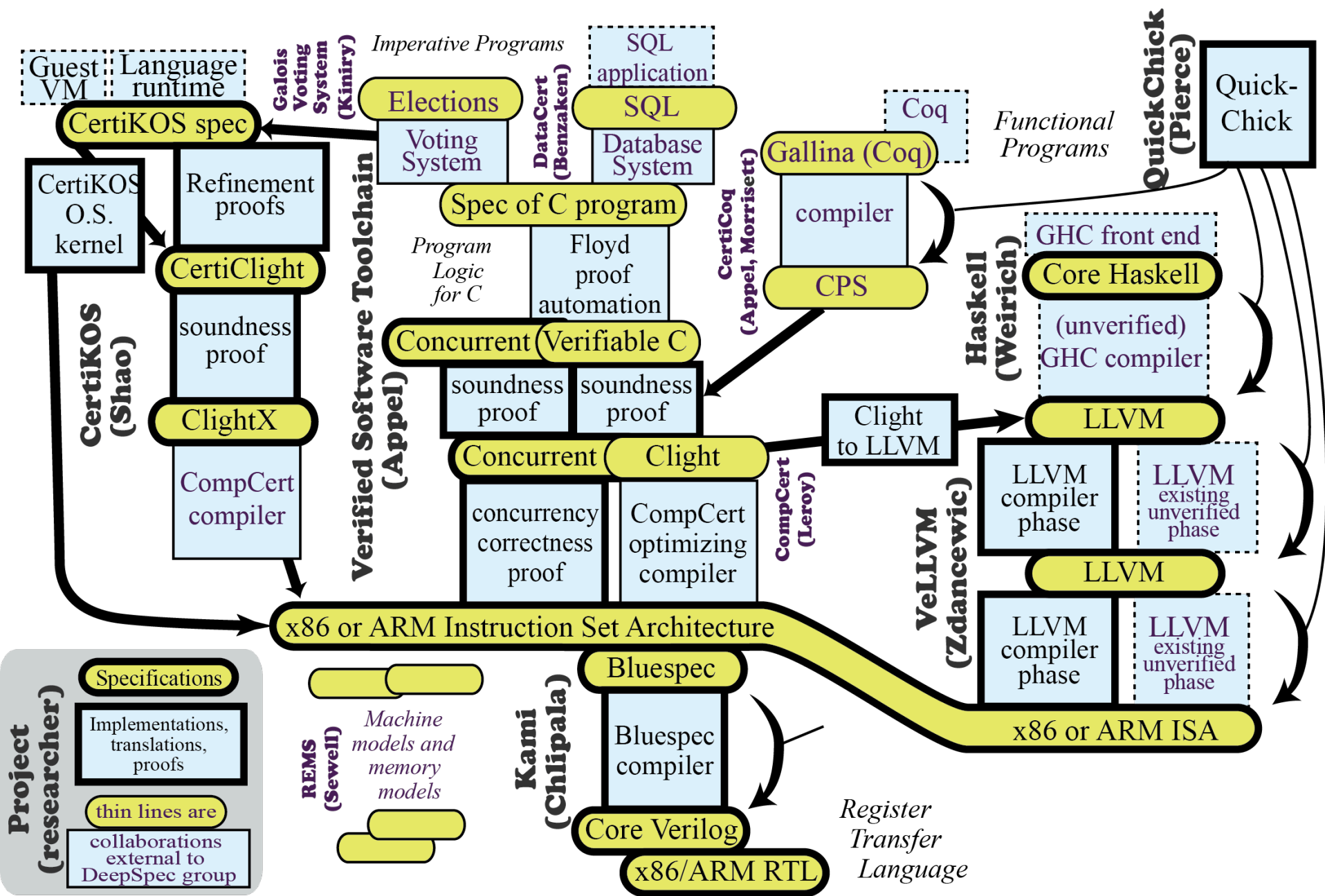
**Verified
Software
Toolchain**

Kami (Chlipala)



CertiCoq, CompCert, VeLLVM, Haskell, QuickChick





形式的手法による CPUとネットワークの検証

Kami

A Platform for High-Level Parametric Hardware
Specification and its Modular Verification
formal specification for RISC-V

<http://plv.csail.mit.edu/kami/>

Overview

- ❑ Kami is a [Coq](#) framework to support implementing, specifying, verifying, and compiling [Bluespec](#)-style hardware components with high developer productivity. *The core of the approach is modular proofs of hardware components.* We aim for specifications and proofs that mirror the modular decomposition of a large hardware system into simpler parts. *We also intend to support techniques for machine-checked correctness proofs of key synthesis algorithms that translate high-level hardware designs into low-level circuit descriptions.*
-

ハードウェア開発の「形式的方法」の現状

形式的方法の実際的な可能性についての懐疑的意見に直面すると、専門家は、形式的な方法が、コンピュータ・ハードウェア業界での品質管理において、すでに広範かつ不可欠な役割を果たしていると答える。

しかし、ソフトウェアに関する経験しかない研究者にとっては、この業界がハードウェアに対して行なっている形式的方法というものが、いたるところで制限されたものだということに驚くことになるだろう。

そこでの形式方法というのは、典型的には、モデルのチェックと充足性の解決に基づいたもので、結局分析は、明示的な状態空間の探索に還元される。

ハードウェア開発の「形式的方法」の現状

例えば、

- 検証作業は、フルシステムの(比較的)小さなコンポーネント、例えば、プロセッサの浮動小数点ユニットに集中している。確かに、浮動小数点演算の実装アルゴリズムの完全な検証は、それ自体、英雄的な努力を必要とするのだが、これらの個々の結果は、システム全体の定理には組み込まれない。
- 比較的弱い性質が証明されている。完全な系の限られた構成要素に対してさえも、自然な正確さの条件からかなりの抽象化のギャップがある。例えば、技術者が仕様について考えて、回路について主張されるべき不変式を自力で思いつくことが一般的であって、そこには、仕様への形式的な関係はない。
- ...

RISC/Vの形式的仕様

- ❑ RISC/Vの仕様は、AlloyやHerdやLemを使って、一部形式化されている。<https://riscv.org/specifications/>
これは、Alloyを使ったメモリーモデル。
<https://github.com/daniellustig/riscv-memory-model>
 - ❑ もっとも包括的なのは、Samuel Gruetterの、Haskellで記述された RISC/Vの仕様である。
<https://github.com/mit-plv/riscv-semantics>
 - ❑ 彼はまた、この仕様をCoqに変換した。
<https://github.com/mit-plv/riscv-coq>
-

Kamiが行ったこと

- ❑ Adam ChlipalaらのKamiチームは、coqで記述されたRISC/Vの仕様から、RISC/Vのハードウェア記述に利用されている BlueSpec <https://bluespec.com/> のコードを吐き出させることに成功する。それが "Kami project"である。
<https://github.com/mit-plv/kami>
- ❑ Kami project では、その出力を使って、シミュレータ上でRISC/Vチップを動かすことも、FPGAに焼き込んでRISC/Vの動作のチェックもできる。仕様から動くハードが生成できる。
- ❑ それ以上に重要なことは、仕様が完全に誤りなくチップ上に実装されたことを、きちんと証明できること。これはハードウェアの開発では画期的なこと。
- ❑ Kamiは、RISC/Vの公式仕様になるという。
- ❑ RISC/Vに限らず、この手法は、ハードウェア開発の現場に、大きなインパクトを与えるだろう。

Kami: A Coq Framework to Support Implementing and Verifying Bluespec-style Hardware Components

Adam Chlipala et al.

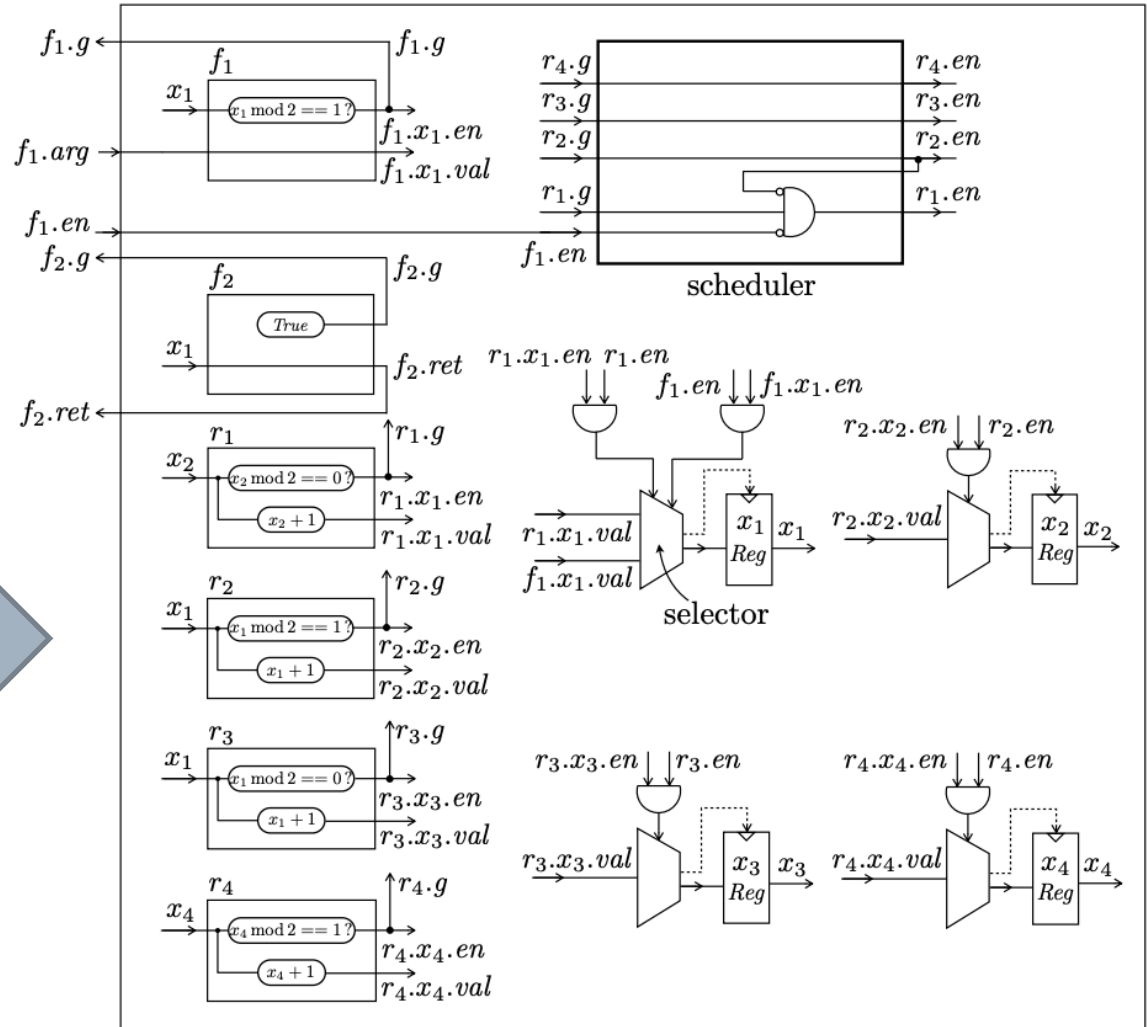
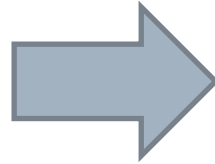
2017年

<http://adam.chlipala.net/papers/KamiICFP17/KamiICFP17.pdf>


```

1  Definition example := MODULE {
2    Register "x1" : Bit 32 <- 0
3    Register "x2" : Bit 32 <- 1
4    Register "x3" : Bit 32 <- 2
5    Register "x4" : Bit 32 <- 3
6    with Rule "r1" :=
7      Read v <- "x2";
8      Assert (#v mod 2 == 0);
9      Write "x1" <- #v + $1;
10     Retv
11   with Rule "r2" :=
12     Read v <- "x1";
13     Assert (#v mod 2 == 1);
14     Write "x2" <- #v + $1;
15     Retv
16   with Rule "r3" :=
17     Read v <- "x1";
18     Assert (#v mod 2 == 0);
19     Write "x3" <- #v + $1;
20     Retv
21   with Rule "r4" :=
22     Read v <- "x4";
23     Assert (#v mod 2 == 1);
24     Write "x4" <- #v + $1;
25     Retv
26   with Method "f1"(a: Bit 32): Unit :
27     Read v <- "x1";
28     Assert (#v mod 2 == 1);
29     Write "x1" <- #a;
30     Retv
31   with Method "f2"(_: Unit): Bit 32 :=
32     Read v <- "x1";
33     Return #v
34 }.

```



	Kami	Bluespec
IPC _{average}	0.21	1.00
Clock Frequency (MHz)	142.86	31.25
#LUTs per core	3,061	2,184
#FFs per core	1,545	3,440

Fig. 15. RISC-V cores in Kami and Bluespec

Single-threaded	IPC
Euclid's GCD	0.23
Factorial	0.22
Bubble Sort	0.25
Towers of Hanoi	0.18
Multi-threaded	IPC
Shared Counter (Dekker's)	0.19
Shared Counter (Peterson's)	0.20
Parallel Matrix Multiply	0.20
Concurrent Bank Transactions	0.18

Fig. 16. Performance on Kami-based multiprocessor

CONCLUSION AND FUTURE WORK

We have developed a fairly complete infrastructure to design, verify, and synthesize hardware systems. We have designed and verified a multiprocessor system containing RISC-V cores connected to a fairly complicated memory system with coherent caches. While we have started developing verified RISC-V components, we plan to scale up the realism level by designing more complex processor optimizations (such as superscalar, out-of-order execution) verified to adhere to the simple instantaneous-processor specification. We plan to implement a verified compiler in Coq to translate Kami modules directly into hardware netlists. Another fruitful extension of Kami would be to support proof of liveness properties like deadlock freedom.

<http://adam.chlipala.net/papers/KamiICFP17/KamiICFP17.pdf>

Verdi

Formally Verifying Distributed Systems

<https://verdi.uwplse.org/>

Verdi

Formally Verifying Distributed Systems

Distributed systems are hard to get right in large part because they must tolerate faults gracefully:

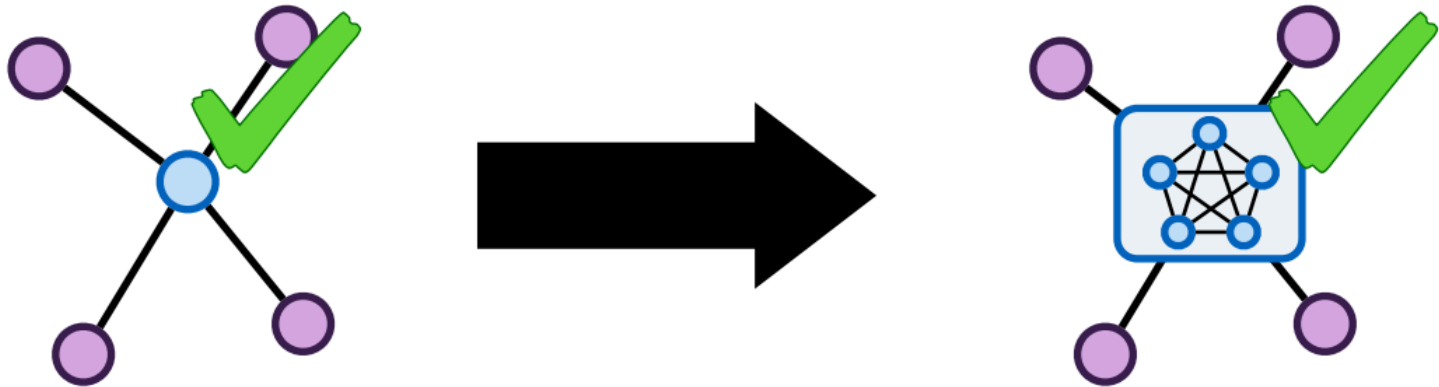
machines may crash and the network may drop, reorder, or duplicate packets.

Verdi is a framework from the University of Washington to implement and formally verify distributed systems. Verdi supports several different fault models ranging from idealistic to realistic. Verdi's *verified system*

transformers (VSTs) encapsulate common fault tolerance techniques. Developers can verify an application in an idealized fault model, and then apply a VST to obtain an application that is guaranteed to have analogous properties in a more adversarial environment.

<https://verdi.uwplse.org/>

Verifying Distributed Systems

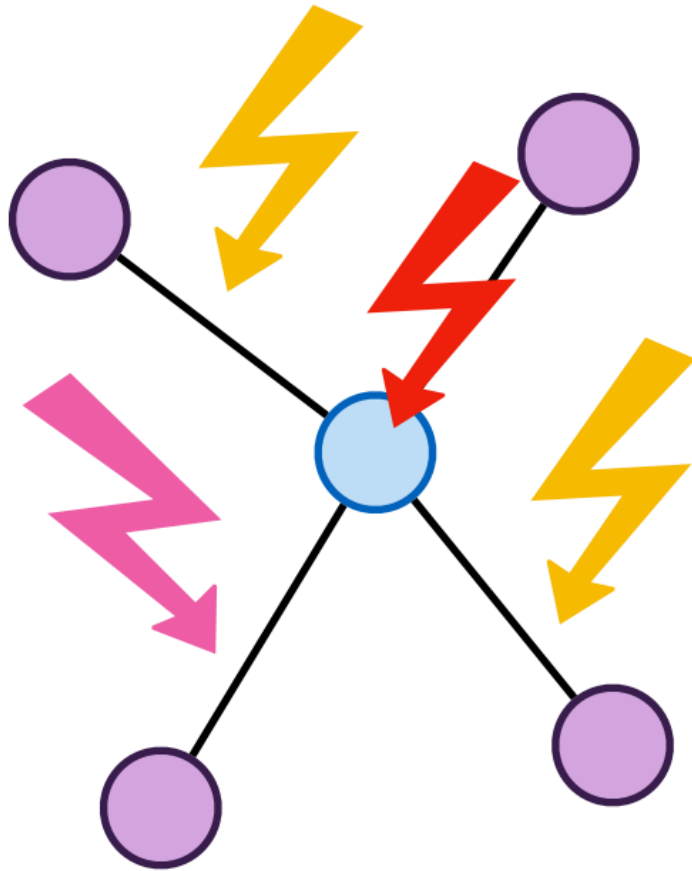


Zachary Tatlock

Coq Workshop 2018

<https://coqworkshop2018.inria.fr/files/2018/08/180708-coq-workshop.pdf>

How distributed systems fail

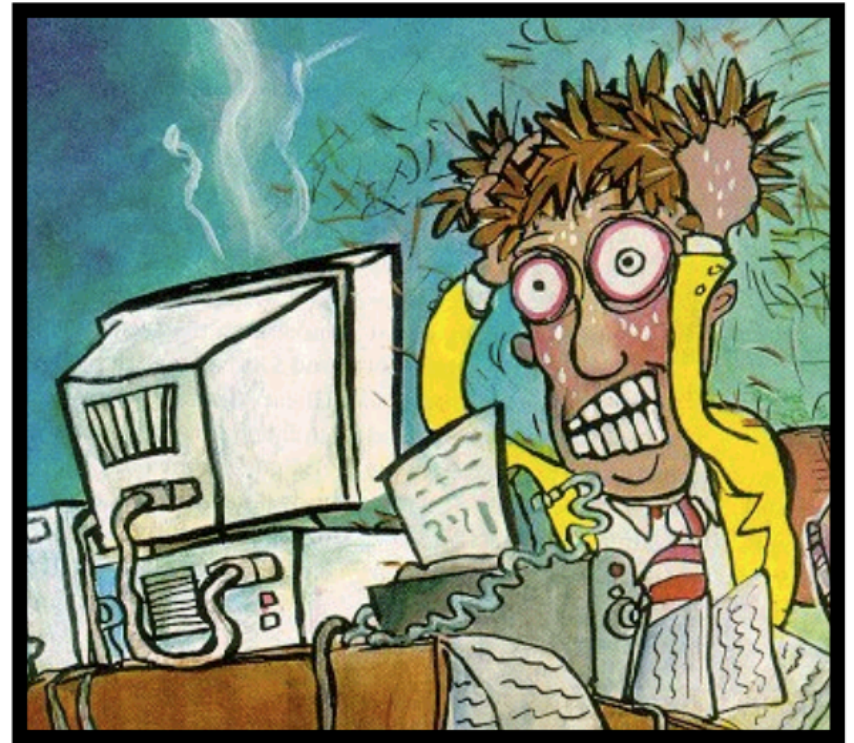
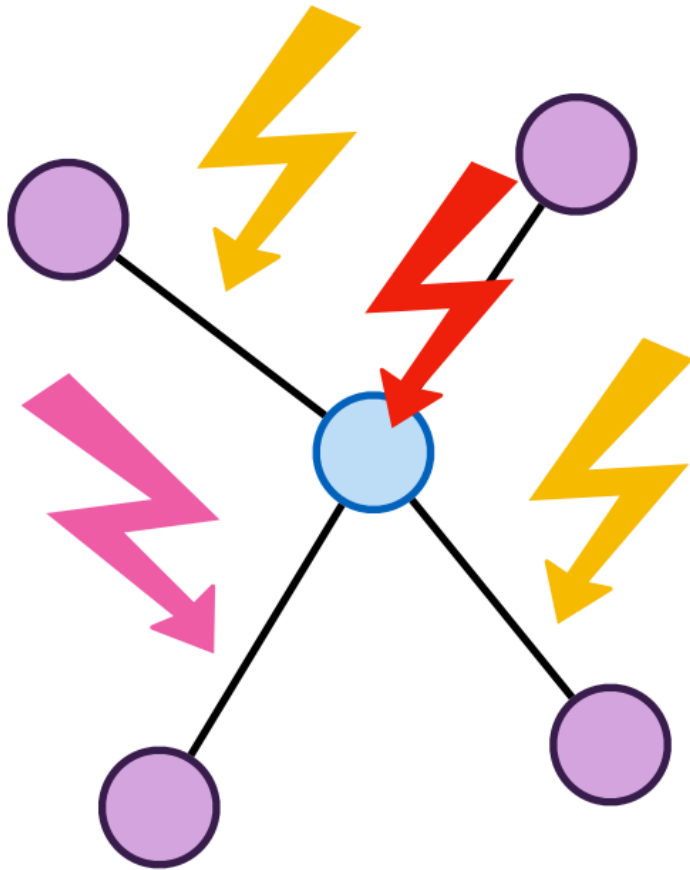


Challenges

- concurrency
- message drops
- message dups
- message reorder
- machine crash
- machine reboot

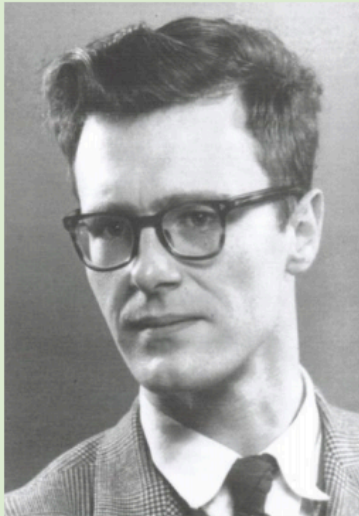
分散システムはいつ落ちるか？

How distributed systems fail



Too many possible behaviors to effectively test!

How distributed systems fail



When exhaustive testing is impossible, our trust can only be based on proof.

Edsger W. Dijkstra

Under the Spell of Leibniz's Dream

可能な場合を全て尽くしたテストが不可能なら、
信頼は証明の上にのみ基礎付けられる

Toward verified distributed systems

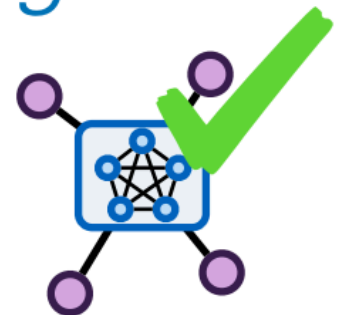
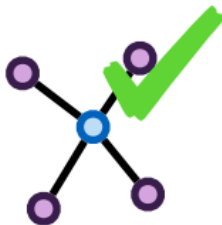
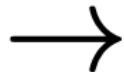
Formalize *network semantics*
capture how faults can occur



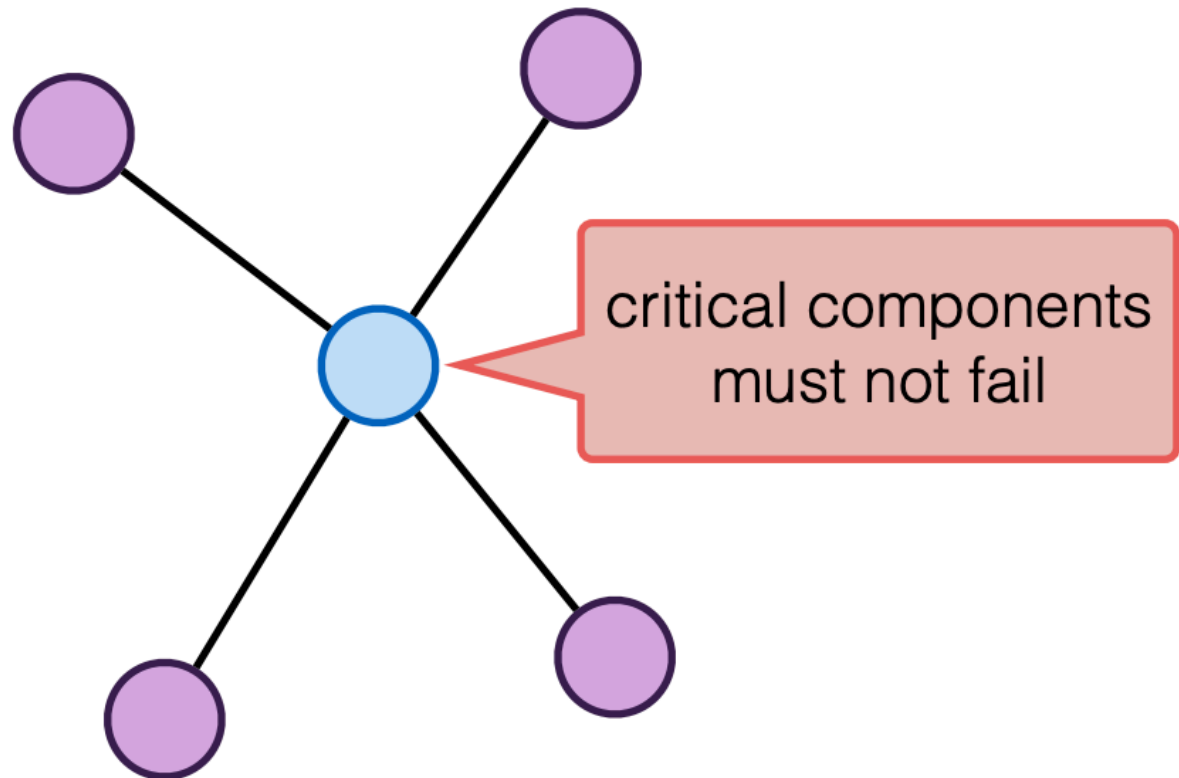
Separate app / fault reasoning

develop and prove in simple fault model

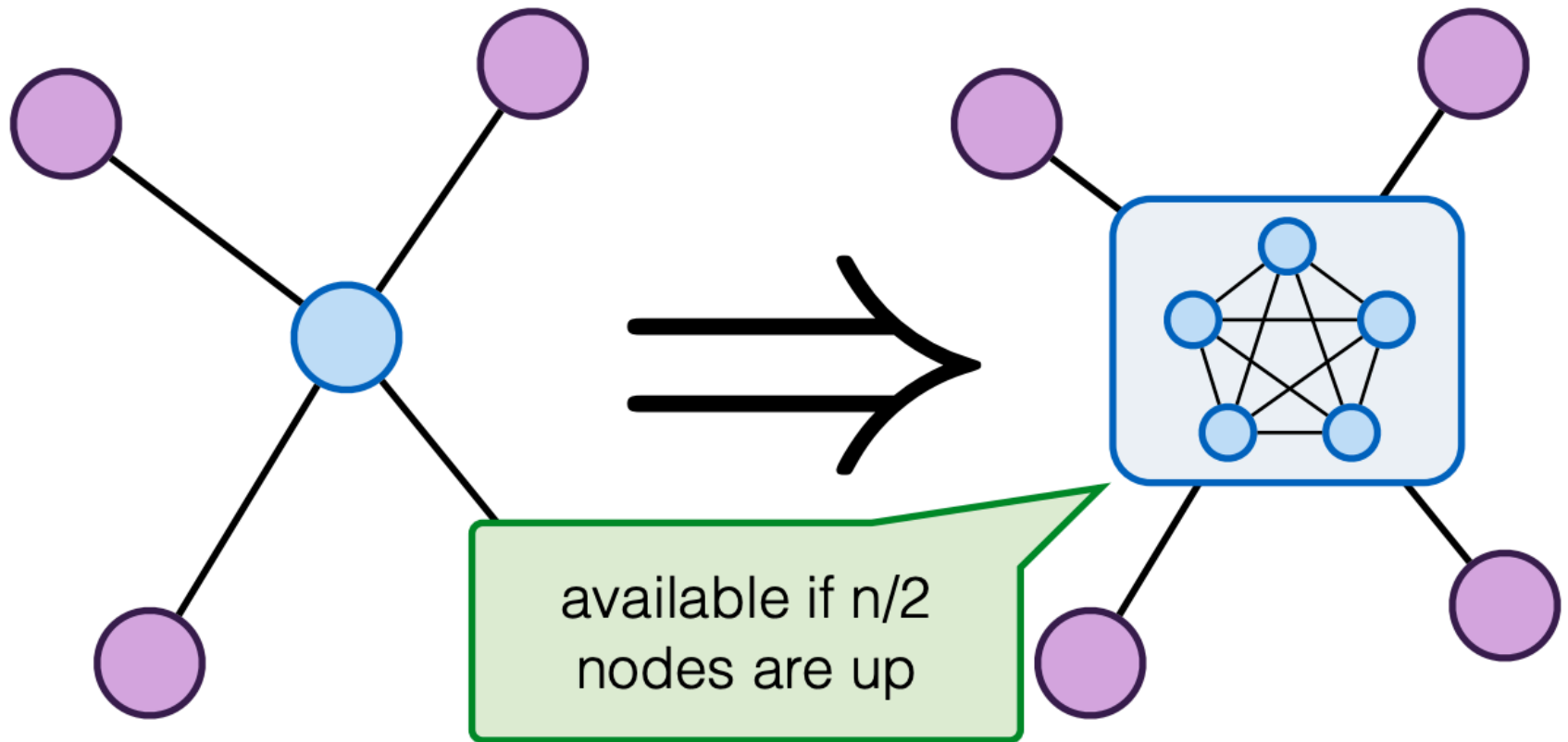
apply generic verified fault handling



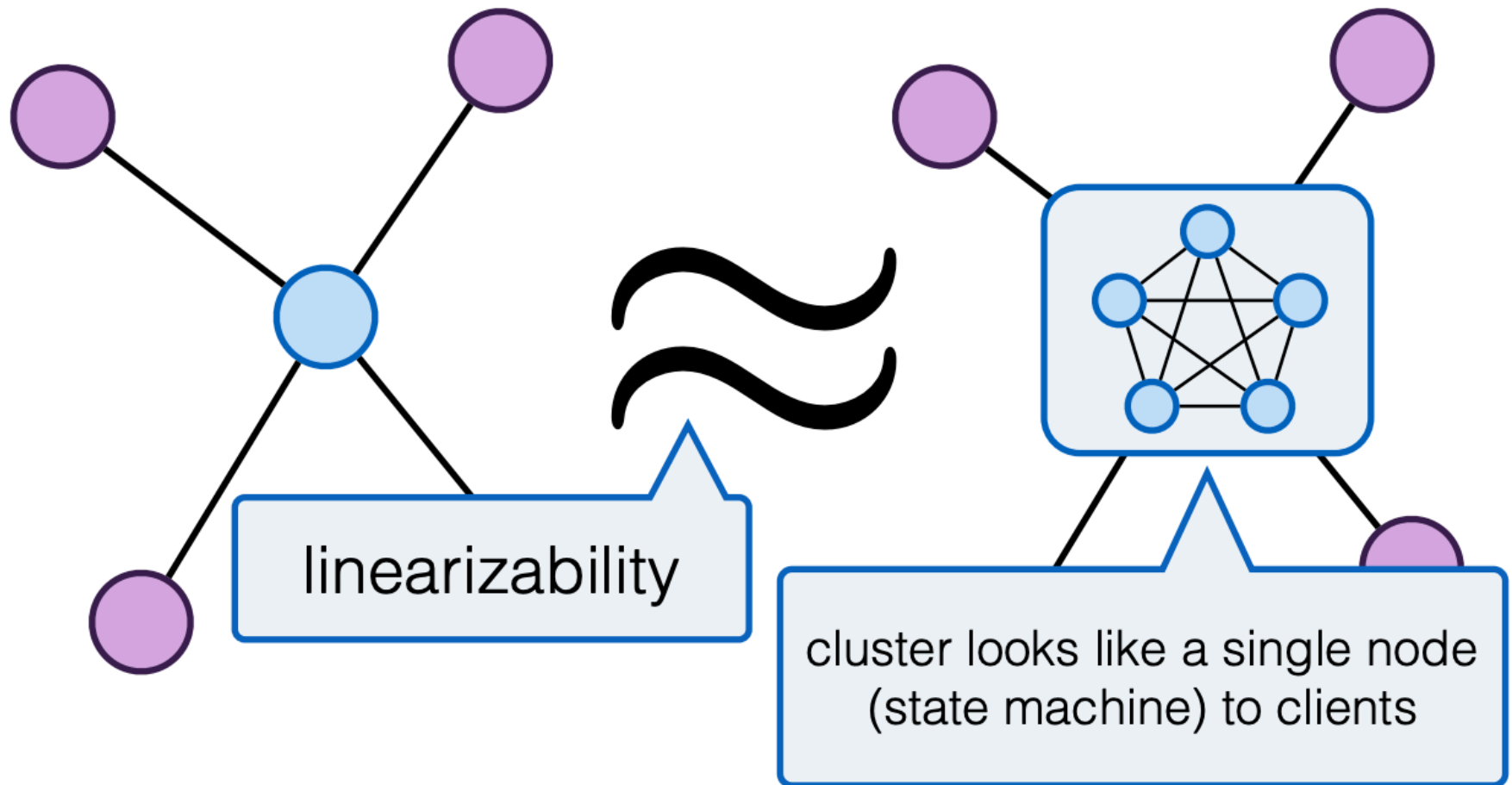
Example: verifying Raft in Verdi



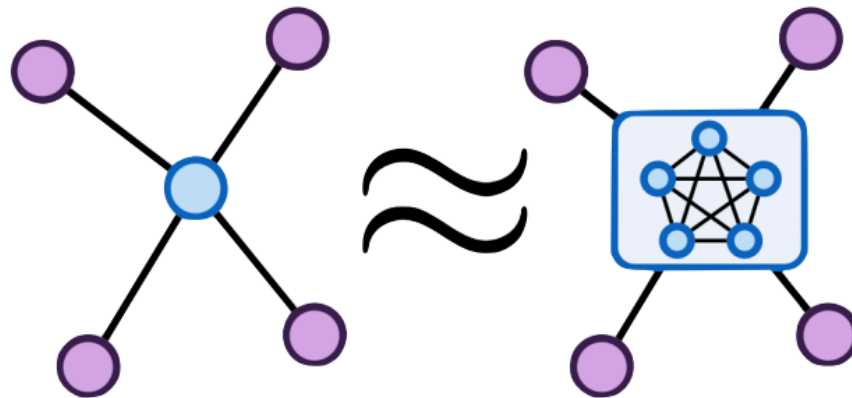
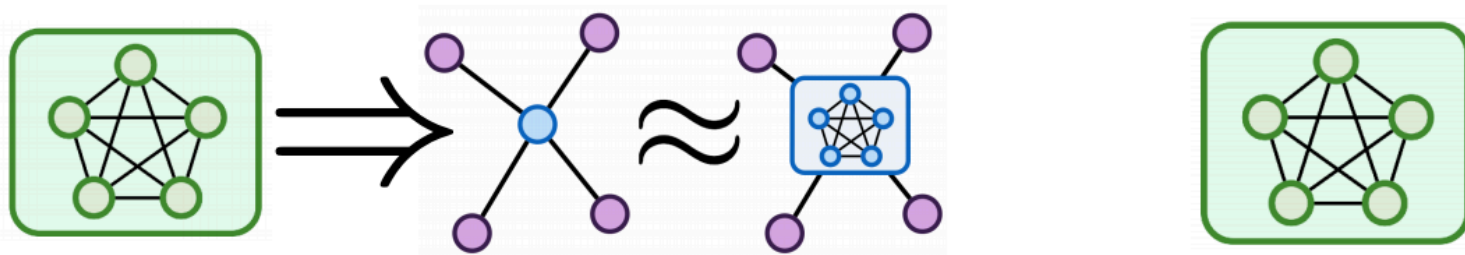
Replication for fault tolerance



Replication correctness



Proving Raft in Verdi



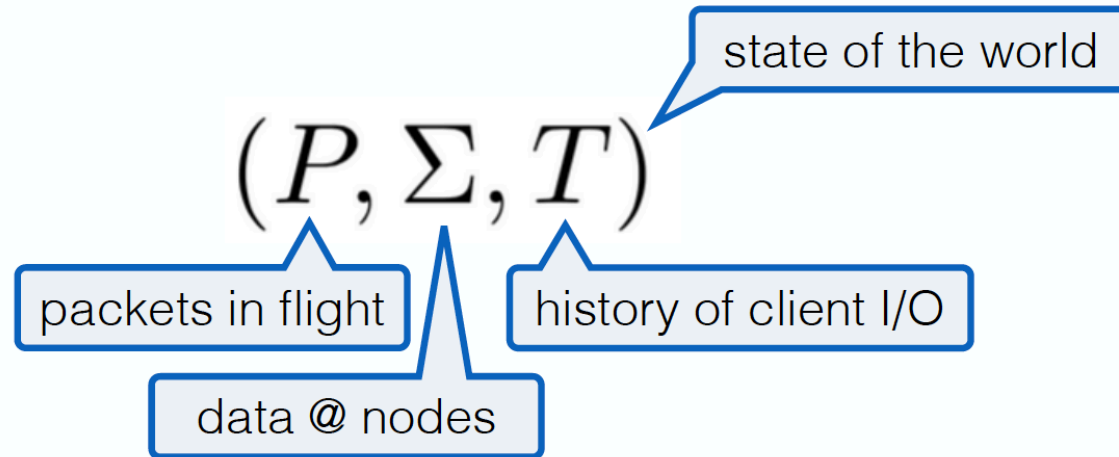
First formal verification of Raft

50k lines of Coq

18 person-months

Considerable pizza and beer

2. Network semantics



$$(P, \Sigma, T) \rightsquigarrow (P', \Sigma', T')$$

Good old small step operational semantics.

Example rule: message delivery

run handler
on message

get response

$$\frac{H_{\text{net}}(dst, \Sigma[dst], src, m) = (\sigma', o, P') \quad \Sigma' = \Sigma[dst \mapsto \sigma']}{(\{(src, dst, m)\} \uplus P, \Sigma, T) \rightsquigarrow (P \uplus P', \Sigma', T ++ \langle o \rangle)}$$

if this message is
in the network

resulting new
global state

2. Network semantics

$$\frac{H_{\text{net}}(dst, \Sigma[dst], src, m) = (\sigma', o, P') \quad \Sigma' = \Sigma[dst \mapsto \sigma']}{(\{(src, dst, m)\} \uplus P, \Sigma, T) \rightsquigarrow (P \uplus P', \Sigma', T ++ \langle o \rangle)} \text{DELIVER}$$

$$\frac{p \in P}{(P, \Sigma, T) \rightsquigarrow (P \uplus \{p\}, \Sigma, T)} \text{DUPLICATE}$$

$$\frac{}{(\{p\} \uplus P, \Sigma, T) \rightsquigarrow (P, \Sigma, T)} \text{DROP}$$

$$\frac{H_{\text{tmt}}(n, \Sigma[n]) = (\sigma', o, P') \quad \Sigma' = \Sigma[n \mapsto \sigma']}{(P, \Sigma, T) \rightsquigarrow (P \uplus P', \Sigma', T ++ \langle \text{tmt}, o \rangle)} \text{TIMEOUT}$$

An Empirical Study on the Correctness of Formally Verified Distributed Systems

Pedro Fonseca Kaiyuan Zhang Xi Wang Arvind Krishnamurthy

University of Washington

{pfonseca, kaiyuanz, xi, arvind}@cs.washington.edu

Abstract

Recent advances in formal verification techniques enabled the implementation of distributed systems with machine-checked proofs. While results are encouraging, the importance of distributed systems warrants a large scale evaluation of the results and verification practices.

This paper thoroughly analyzes three state-of-the-art, formally verified implementations of distributed systems: Iron-Fleet, Verdi, and Chapar. Through code review and testing, we found a total of 16 bugs, many of which produce serious consequences, including crashing servers, returning incorrect results to clients, and invalidating verification guarantees. These bugs were caused by violations of a wide-range of assumptions on which the verified components relied. Our

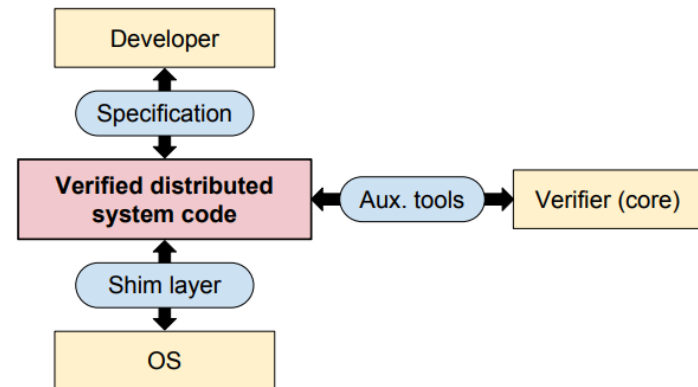


Figure 1: An overview of the workflow to verify a distributed system implementation.

InteractionTree

Basis for DeepSpec Web Server

<https://github.com/DeepSpec/InteractionTrees>

DeepSpec Web Server

We aim to build a verified web server



<https://deepspec.org/entry/Project/DeepSpec+Web+Server/>

DeepSpec Web Server

A team at Penn has begun exploring the design space and building a first-draft prototype (for now, running on Linux) of a web server library **loosely based on the popular libmicrohttpd**. The goal of this short-term effort is to understand the integration issues that will be involved in putting together a fully functional server from components under development within DeepSpec. **In particular, we want to understand what demands it will place on CertiKOS in terms of OS features (IPC, network support, shared-memory processes, interoperation between native clients and Linux VMs), what verification challenges it raises for VST, what integration challenges it poses for using VST and CertiKOS together.**

“Securing the
Internet of Things”

Specifying the DeepSpec Web Server

Lennart Beringer, Joachim Breitner, Olek Gierczak,
Wolf Honore, Nicolas Koh, Yao Li, Yishuai Li, William
Mansky, Benjamin C. Pierce, Stephanie Weirich,
Li-Yao Xia, Steve Zdancewic

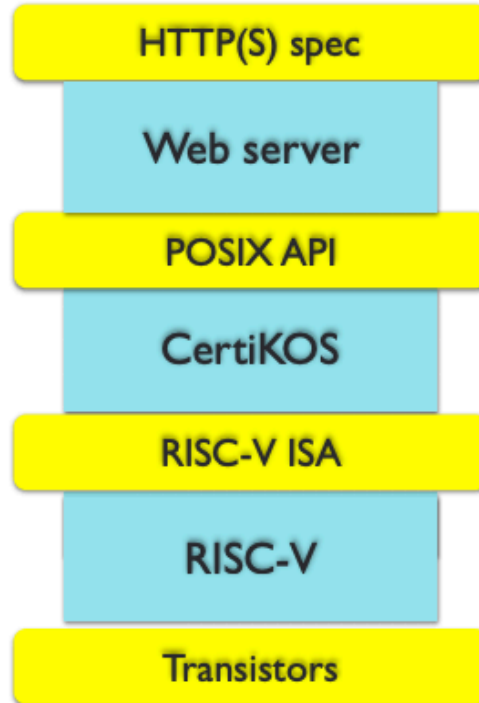
DeepSpec Workshop @ PLDI
June, 2018



<http://www.cis.upenn.edu/~bcpierce/papers/DeepWeb-dsw2018.pdf>



=



Executable high-level specification of HTTP(S) protocols and web services

System call interface specification

Instruction-set specification

RTL description of circuit behaviors

Goal: A "single QED"
encompassing the entire software
and hardware stack

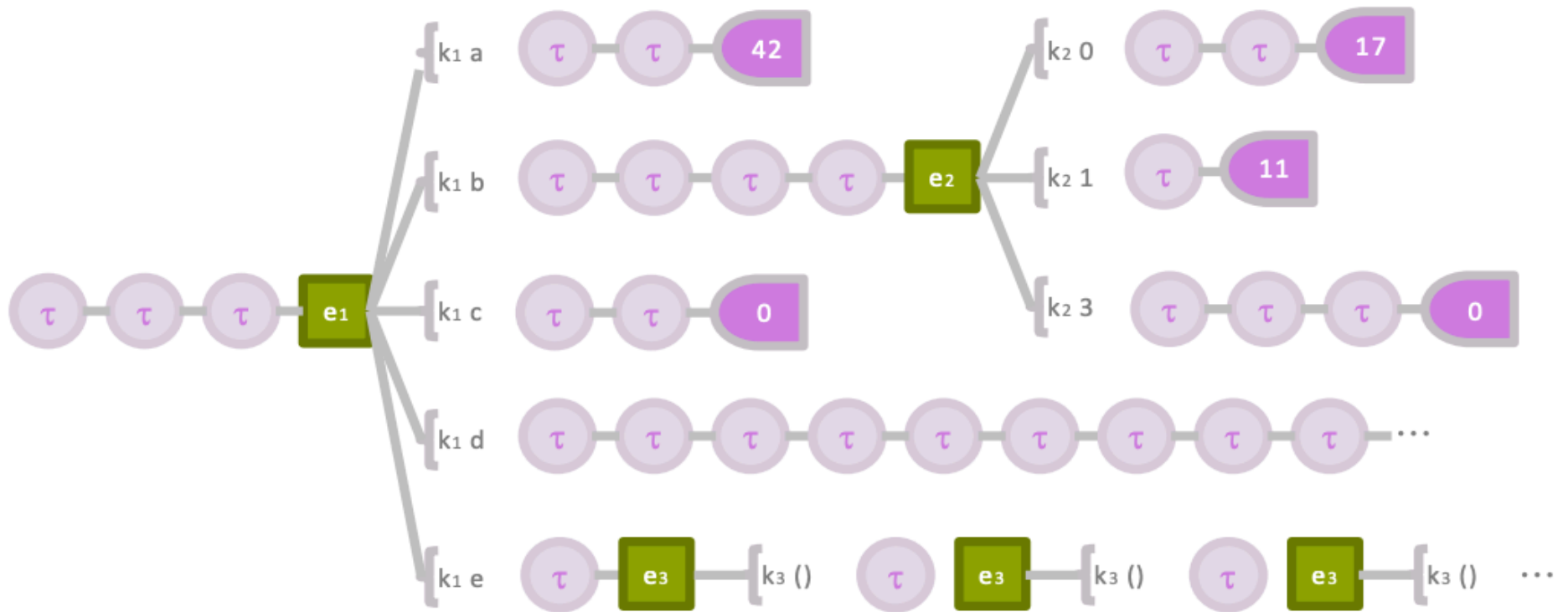
The DeepSpec Web Server

- Based on popular GNU libmicrohttpd library
 - Clean separation between core HTTP-level functionality and the specifics of particular web services
- Aimed at embedded web servers (E.g. IoT device controllers)
- Current implementation (in C)
 - Parsing / printing of core HTTP formats; basic GET functionality
- Current specification (in Coq): ditto, plus
 - PUT functionality
 - ETag support for bandwidth conservation
- Later:
 - Broader coverage of HTTP standard documents
 - TLS authentication
 - Support for database-backed web services

Meta Language

Interaction Trees

- “Abstract syntax trees” for computations
- Nodes labeled with constructors drawn from some set of **observable effects** and branching corresponding to possible results of observations
- Can be given a variety of semantics by supplying different interpretations for the effects
- Effects can be varied to model different views of a computation
 - e.g., “OS view” vs “network view”



Interaction Trees

```
CoInductive M (Event : Type -> Type) X :=  
  | Ret (x:X)  
  | Tau (k: M Event X).  
  | Vis {Y: Type} (e : Event Y) (k : Y -> M Event X)
```

An $M\ E\ X$ is the denotation of a program as a possibly infinite (coinductive) tree, parameterized over a type $Event$ of observable events where:

- **leaves** correspond to final **results** labeled with X ,
- **internal nodes** node are either
 - **internal events** (labeled Tau), or
 - **observable events** (labeled Vis , with a child for every element of the event's result type Y).

プログラムの行数 (Line of Code)

Component	Approximate LOC
Common: axioms for socket API	500
C code for echo server	140
Interaction tree for echo server	100
Hoare triples for functions in echo server	200
VST proofs of C-to-OS-level-spec	400
Coq proofs of OS-level-to-network-level	1000-2000 ?
Total	1500-2500ish
C code for web server	2880
Interaction tree for web server	2000?
Hoare triples for web server	4000?
VST proofs of C-to-OS-level-spec	8000?
Coq proofs of OS-level-to-network-level	20-40k?
Total	30-50k ???

次の「Feit-Thompsonの定理」のCoqによる証明と比較すると、そんなに膨大なわけではない。

「Feit-Thompsonの定理」のCoqによる証明

- 2004年、Georges Gonthierは、Coqを使って「四色問題」を解いた。
- Georges Gonthierは、また、2013年、有限群分類の重要な定理である、「Feit-Thompsonの定理」のCoqによる形式的証明を与えた。
- 15,000の定義、4,300の定理、17万行のソースからなるこの証明を、彼はチーム作業で、6年かけて完成させた。

From C to Interaction Trees

Specifying, Verifying, and Testing a Networked Server

Nicolas Koh et al.

2019

<http://www.cis.upenn.edu/~bcpierce/papers/deep-web-overview.pdf>

進行中のプロジェクト

DeepSpec

Summer School 2018 から

<https://github.com/DeepSpec/dsss18/blob/master/slides/dsss18-certikos2.pdf>



National Science Foundation Expedition in Computing

the science of deep specification

DeepSpec Summer School, July 2018, Lecture Video



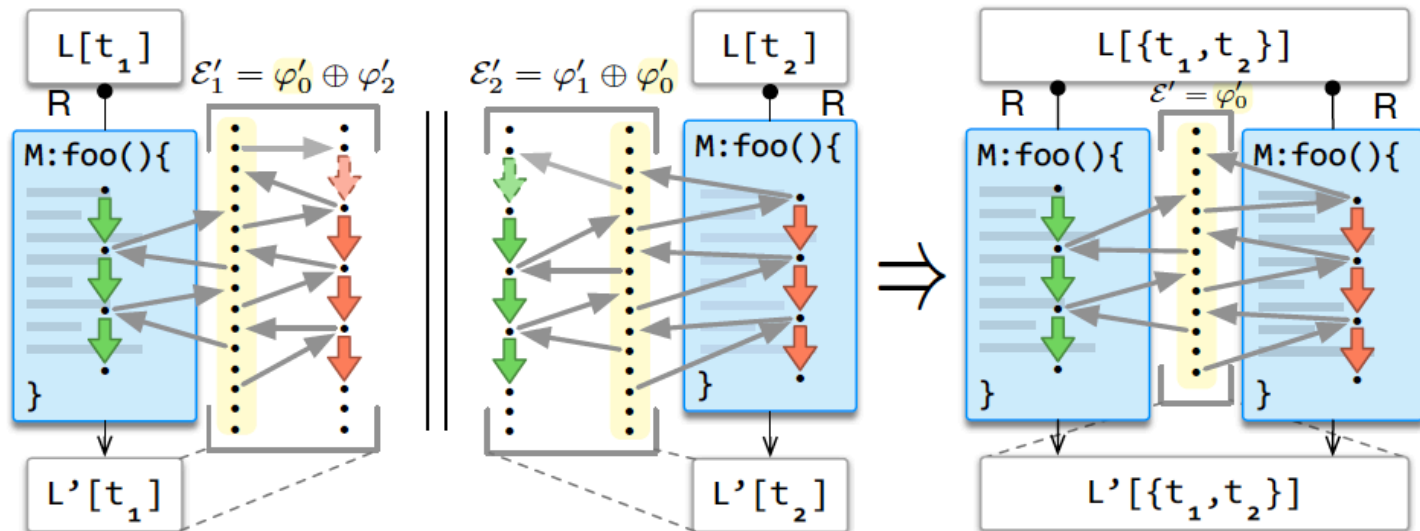
CertiKOS II
Certified Kit Operating System

Zhong Shao

www.deepspec.org

New Result: Certified Concurrent Abstraction Layers (CCAL) [PLDI'18]

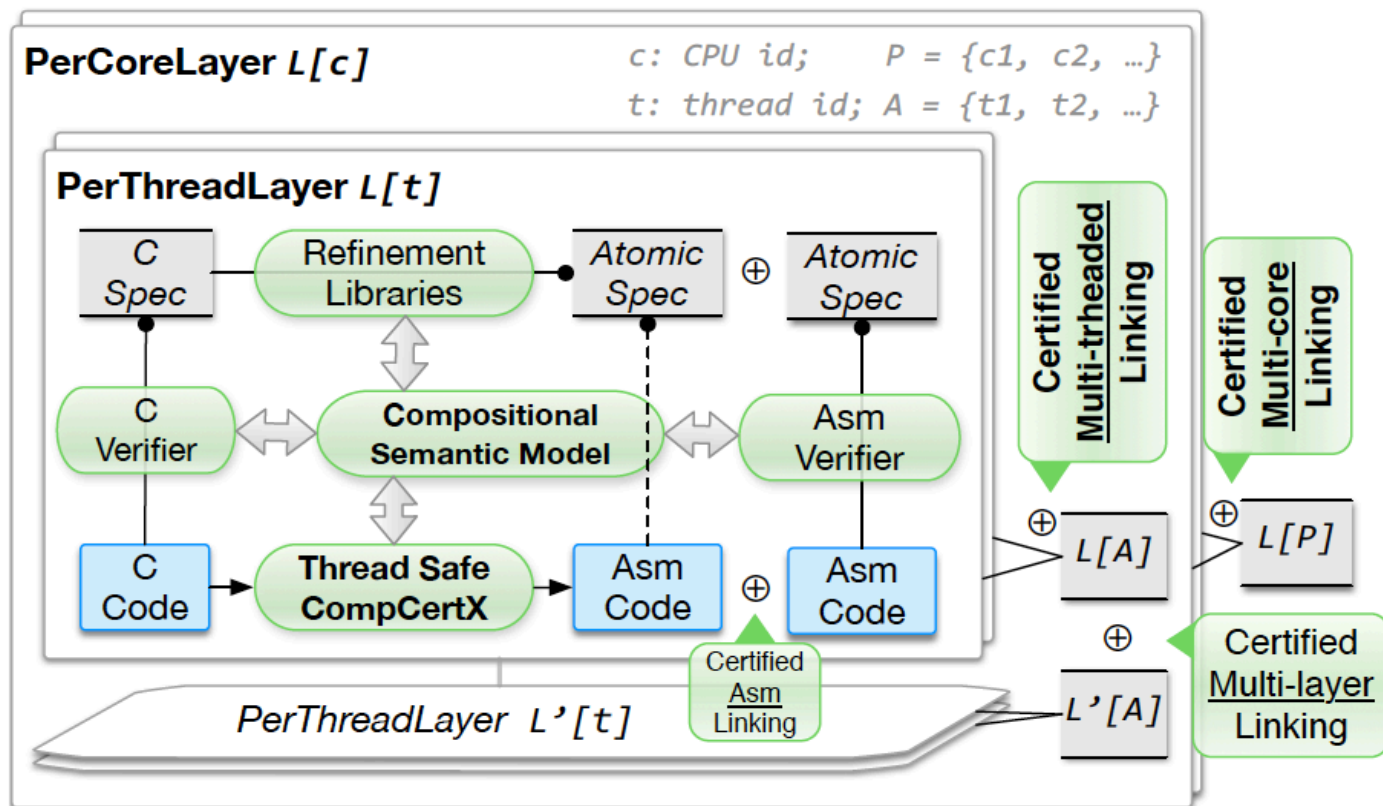
- **Key insight:** model each sequential/concurrent/networked object as a “game” (played against its *environment context*)
- Multicore & multithreaded linking $\leftrightarrow$ composing “games” $\leftrightarrow$ composing “game semantic” strategies
- *Thread-local* (or CPU-local) *reasoning* for CCAL layers (against their *environment contexts*)



New Result: Certified Concurrent Abstraction Layers (CCAL) [PLDI'18]

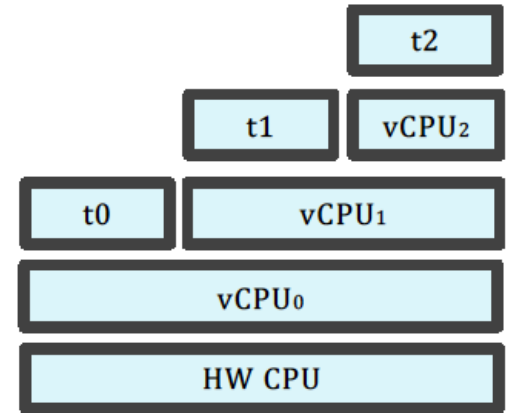
New programming toolkit w. certified multicore & multithreaded linking:

Composition = parallel composition + hiding (abstraction)

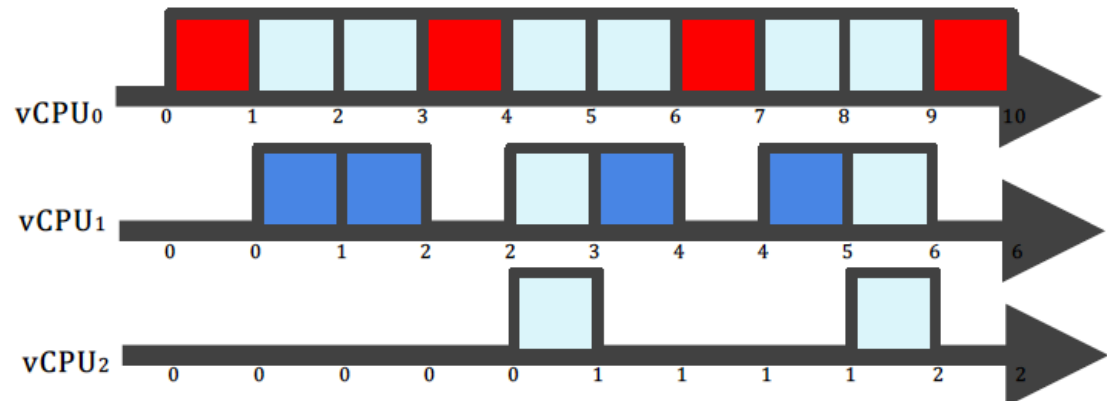


In Progress: Real-Time CertiKOS

- CertiKOS now models timer interrupts & timing registers.
- CertiKOS is now preemptive with a hybrid scheduler
 - Priority scheduling for real-time periodic threads
 - round-robin for batch tasks
- New information-flow security proof for CertiKOS with temporal and spatial partitions
 - **Virtual time:** time available for a specific priority level
 - **Virtual timeline:** mapping from physical time to virtual time



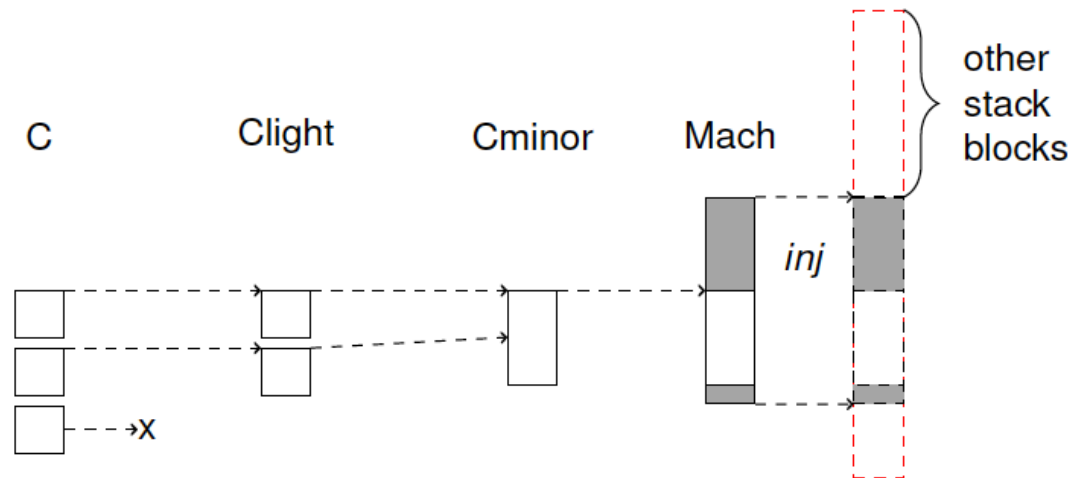
priority	period	WCET
0	3	1
1	5	2
2	9	1



In Progress: Stack-Aware CompCert

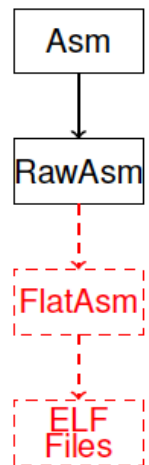
Our proposal: enrich the block-based memory model with an **abstract stack**:

- instrument all CompCert IR semantics w. a **stack requirement spec**
- Maintain an injection from stack blocks to a continuous stack block
- Enforce access control for private / public regions of each stack block



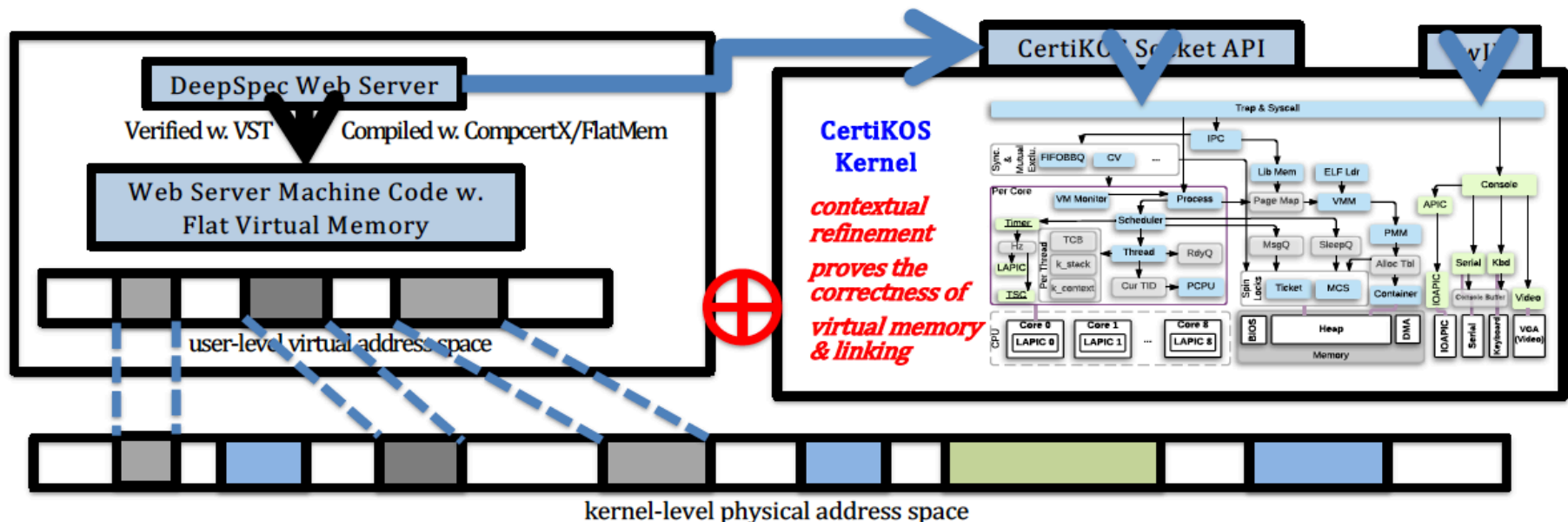
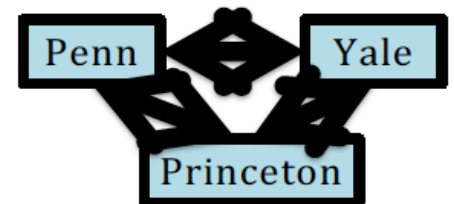
Benefits:

- Light-weight compositionality of heterogenous modules
- Merging the stack blocks into a finite contiguous space
- Verified translation from CompCert Asm into ELF files
 - Asm $\rightarrow$ RawAsm (merged stack blocks) $\rightarrow$ FlatAsm (flat mem) $\rightarrow$ machine code



In Progress: Connecting VST / DeepSpec Web Server with CertiKOS

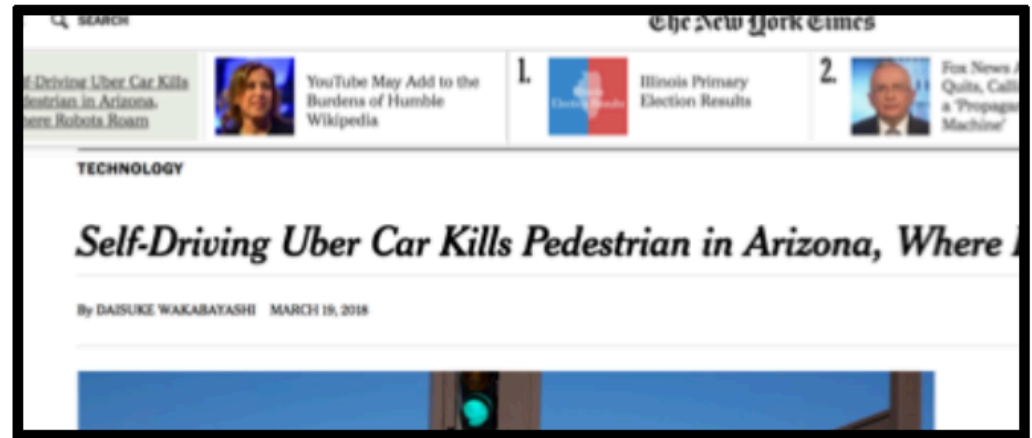
- DeepSpec web server now runs on CertiKOS blackbox (see the [demo](#))
- The web server will be verified using VST, while the socket API is specified as functions over CertiKOS abstract state
 - How to model the physical network?
 - What parts of CertiKOS should the web server be aware of?
 - How to make CertiKOS specs be compatible with VST specs?
- Integration of user-level flat virtual memory and CertiKOS kernel memory



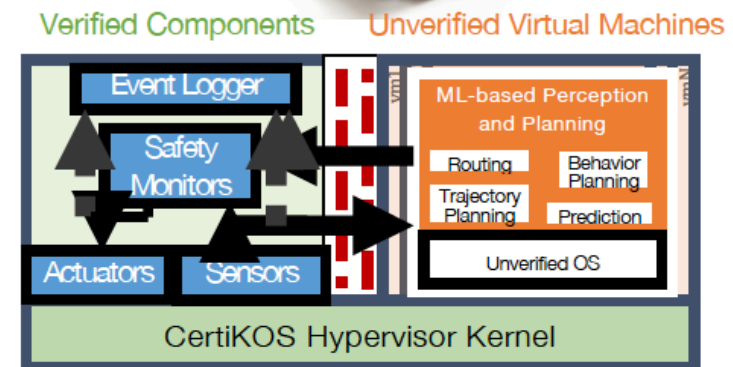
In Progress: Certified Self-Driving Cars

Goal: End-to-end verification of safety & security properties of smart cars

Machine learning enables self-driving, but does not guarantee safe-driving



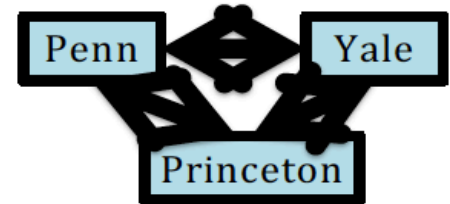
- **Model-based safety monitors**
 - e.g., safe distance, obstacle detection
 - Multi-agent verification (by rely-guarantee & CCAL)
- **Verified event logger for post-analysis**
 - E.g., accident investigation, maintenance
- **CertiKOS-based security guarantee**
 - Isolation for critical components
 - Real-time guarantee



Highlights of Future Plans

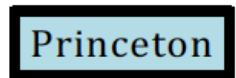
- Applications

- Certified DeepSpec web server demo
- Connecting DeepSpec server and VST with CertiKOS
- Certified self-driving car & quadcopter



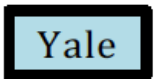
- Verification of user-level C programs using VST

- Concurrent separation logic for VST



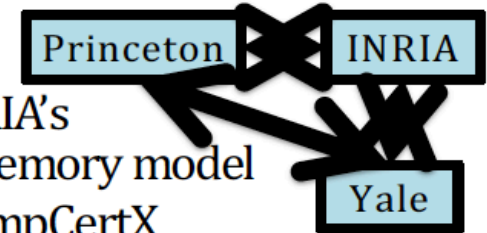
- Compositional concurrent OS and abstract machines

- General compositional model for concurrent & networked abstract machines
- Preemption, priority scheduling, and resource-aware CertiKOS
- CertiKOS port to ARM / x86 / RISC-V & latest Coq/CompCert



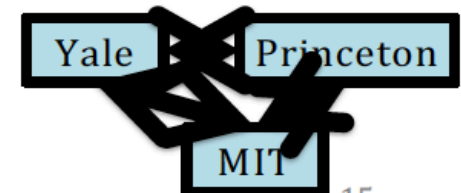
- User- & kernel-level semantic integration

- Connecting VST spec interface to CompCert w. Yale's & INRIA's
- CompCert extension for generating machine code w. flat memory model
- Compositional & Concurrent & Stack-aware CompCert/CompCertX



- Machine-level semantics

- Bridging CertiKOS/CompCertX LAsm & RISC-V semantics
- Provable resilience to side-channel attacks



DeepSpec 2019 Workshop Title

<https://pldi19.sigplan.org/home/deepspec-2019>

DeepSpec 2019 Workshop Title

☆ [Abstraction, Subsumption, and Linking in VST](#)

Lennart Beringer, Andrew Appel

☆ [Automated Formal Memory Consistency Verification of Hardware](#)

Yatin Manerkar

☆ [Closure Conversion is Safe for Space](#)

Zoe Paraskevopoulou, Andrew Appel

☆ [Coinductive Reasoning about Interaction Trees](#)

Chung-Kil Hur

☆ [Compositional Verification of Preemptive OS Kernels with Temporal and Spatial Isolation](#)

Mengqi Liu

☆ [Connecting Separation Logic with First-Order Reasoning on Memory](#)

William Mansky, Wolf Honore

☆ [Coverage Guided, Property Based Testing](#)

Leonidas Lampropoulos

☆ [Development of the RISC-V ISA Formal Specification](#)

Rishiyur Nikhil

☆ [Fast, Verified Partial Evaluation](#)

Adam Chlipala

☆ [Implementation and Verification of Modular Effectful Systems in Coq using FreeSpec](#)

Yann Régis-Gianas

☆ [Interaction Trees: Representing Recursive and Impure Programs in Coq](#)

Steve Zdancewic

☆ [Modular Correctness Proofs at the Hardware-Software Interface](#)

Joonwon Choi

☆ [Names, Places, and Things: Generic Traversals over Generic Syntax with Binding](#)

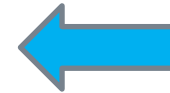
James McKinna

☆ [Overview of the DeepSpec Expedition and its Capstone Application](#)

Benjamin C. Pierce

☆ Project Oak: Control Data in Distributed Systems, Verify All The Things
Ben Laurie

☆ Project Updates from Participating Sites
Andrew Appel, Adam Chlipala, Zhong Shao



☆ Refinement-Based Game Semantics for CompCert
Jérémie Koenig

☆ Stack-Aware CompCert
Yuting Wang

☆ Typed Programming with Algebraic Effects (in terms of ambient values, functions, and control)
Daan Leijen

第三部

プログラムと論理



10/10 マルレク・サブゼミ 「プログラムと論理」
<https://program-logic.peatix.com/view>



木 10月10日 19:00

マルレク・サブゼミ 「プログラムと論理」

MISTLETOE OF TOKYO

★ 興味あり

第三部 **Agenda**

プログラムと論理

- 形式的手法の二つの起源
- 形式的手法をめぐる20世紀の「論争」
- ダイクストラの21世紀へのメッセージ
- 証明支援システム Coq
- Deep Specification – Software Foundation

形式的手法の二つの起源

現代の「形式的手法」の起源

- Deep Specificationのような、現代の「形式的手法」には、いずれも50年近く前の1970年代に遡る、二つの異なった「起源」がある。
 - 両者ともに「プログラムと論理」という問題意識は共通していたのだが、生まれた場所が異なっていた。一つは、計算機科学の領域で生まれ、もう一つは論理学＝数学の領域で生まれた。
 - 前者の代表は、ホーアやダイクストラである。彼らは、代入文やif文、シーケンシャルな実行や繰り返しのWhile文といった具体的なプログラムの構成と論理との関係を研究した。
 - 後者の代表は、ハワードやマーチン・レフである。彼らは、「型の理論」を作り上げ、抽象的だが、プログラムの実行は、ある定理の証明に他ならないという認識にたどり着いた。
-

-
- ただ、今から思えばの話だが、「プログラムと論理」という問題を扱うには、前者は、古典論理を自明のベースとしていて、論理の多様な性質への抽象的洞察が欠けていたし、後者は、自然と直観主義論理の立場に落ち着いたのだが、具体的なプログラム言語との接点は、必ずしも明確ではなかったように思う。（その意味では、計算機科学の分野では、ランポートによる「時制論理」の導入は、大きなブレークスルーだった。）
 - 様々な紆余曲折を経て、両者の統一は、21世紀に持ち込まれた。
-

- 「型の理論」については、先日のマルレク・サブゼミで、少し説明した。<https://www.marulabo.net/docs/type-theory-2/>
- ここでは、計算機科学の分野での「プログラムと論理」に関係した代表的な論文を上げておこう。
- "An axiomatic basis for computer programming"
C.A.R Hoare 1969年
<https://www.cs.cmu.edu/~crary/819-f09/Hoare69.pdf>
- "Guarded Commands, Nondeterminacy and Formal Derivation of Programs"
Edsger W. Dijkstra 1975年
<http://citeseerx.ist.psu.edu/viewdoc/download...>
- "A Temporal Logic of Actions"
Leslie Lamport 1990年
<http://lamport.azurewebsites.net/pubs/old-tla-src.pdf>

形式的手法をめぐる20世紀の「論争」

「証明」の役割について

形式的手法をめぐる「論争」

- 70年代のホーアやダイクストラのコンピュータ・サイエンスに「形式的手法」を導入しようという試みは、必ずしも広く受け入れられた訳ではなかった。
- 1979年、ACMは、その論文誌CACMに次のような内容の論文を掲載する。
- 「この論文では、プログラムの形式的検証は、たとえそれが得られたとしても、計算機科学とソフトウェア・エンジニアリングにおいては、数学での証明と同じ役割は果たす事はないと議論される。更に言えば、連続性の欠如、変化の不可避性、決定的に多くの現実のプログラムの仕様の複雑さは、形式的検証の過程を、正当化することも管理することも困難にする形式的検証の使いやすさは、プログラム言語の設計に大きな影響力を持つことはないように感じられる。」

ランポートの抗議

- この論文、“Social Processes and Proofs of Theorems and Programs” De Millo, Lipton and Perils
<https://www.cs.umd.edu/~gasarch/BLOGPAPERS/social.pdf> は、「プログラムの検証なんて、必ず失敗する。
"program verification is bound to fail. "」と断じたのだ。
 - ランポートは、直ちにACMの編集部に、抗議のメールを送る。
"Letter to the Editor
" <https://www.microsoft.com/.../publicati.../letter-to-the-editor/>
-

ランポートの抗議

Letter to the Editor -- Leslie Lamport

- In the May, 1979 CACM, De Millo, Lipton, and Perlis published an influential paper titled Social Process and Proofs of Theorems and Programs. This paper made some excellent observations. However, by throwing in a few red herrings, they came to some wrong conclusions about program verification. More insidiously, they framed the debate as one between a reasonable engineering approach that completely ignores verification and a completely unrealistic view of verification advocated only by its most naive proponents. (There were, unfortunately, quite a few such proponents.)

-
- The purpose of my letter was to express my dismay. I ironically suggested that they had succumbed to the arguments of De Millo, Lipton, and Perlis in their policy. As a result, my letter was published as a rebuttal to the De Millo, Lipton, and Perlis paper. No one seems to have taken it for what it was—a plea to alter the ACM algorithms policy to require that there be some argument to indicate that an algorithm worked.

「証明」の役割について

- ホーア、ダイクストラ、ランポートらの「形式的手法」に反対する先の論文の著者たちは、「証明」について、次のような考えを持っていた。
- 「数学では、証明の目的は、ある定理の正しさに対する人の確信を増大させることである。確かに、数学者がこの目的のための道具の一つとして、理論的に利用することができるのは、形式論理の長い連鎖である。
しかし、事実はそうではない。彼らが使う証明は、全く別の生き物なのだ。証明は、問題を解決さえない。
証明は、その名が示唆するものとは逆に、確信に向かう一つのステップにすぎないのだ。結局のところ、数学者がある定理に確信を持つか否かを決定するのは、社会的プロセスなのだ」と我々は信じている。」

Social Process ?

In mathematics, the aim is to increase one's confidence in the correctness of a theorem, and it's true that one of the devices mathematicians could in theory use to achieve this goal is a long chain of formal logic. But in fact they don't. What they use is a proof, a very different animal. Nor does the proof settle the matter; contrary to what its name suggests, **a proof is only one step in the direction of confidence. We believe that, in the end, it is a social process that determines whether mathematicians feel confident about a theorem** --- and we believe that, **because no comparable social process can take place among program verifiers, program verification is bound to fail. We can't see how it's going to be able to affect anyone's confidence about programs.**

「社会的なプロセス」

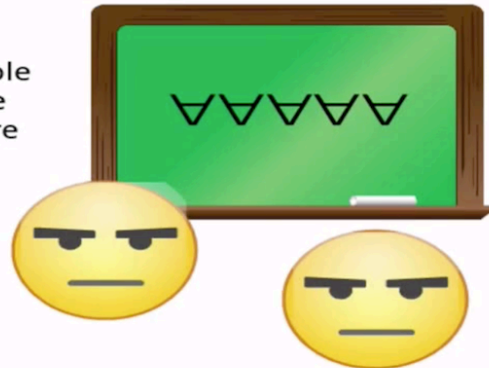
- 彼らの論文のタイトルに、「社会的なプロセス」とあったのは、こういう意味だったのだ。
 - 「プログラムの検証者たちの間に、数学と比較できるような社会的プロセスが存在しないのだから、プログラムの検証は、失敗するように定められていると、我々は信じている。プログラムについての誰かの確信に、証明がどのように影響を与えるのか、我々は見ることができないのだ。」
 - (僕の理解では、たいていの数学者にとって、「定理に対する確信」は、まず、最初に直観的に生まれるものだ。だから、それを証明しようとする。それに、数学者は、IT業界の人のように、コミュニティや学会に集まって「社会的プロセス」での合意をつくるのが好きなようには見えないのだが。)
-

-
- ❑ 数学論的には全くの愚論である。ACMは、計算機科学ではもっとも「権威ある」学会の一つなのだが、そこでも、一時期、こうした議論が行われていたことには注意が必要だと思う。
 - ❑ なぜなら、それは現実の多数の開発者の心情を代弁している可能性があるからである。
 - ❑ 一方、「形式的手法」の側は、ランポートだけでなく、あとで紹介するダイクストラを含めて、この論文が論文として学会誌に受容されたことにショックを受けたように思う。
 - ❑ 逆の意味で、40年近く経った今も、この論文は「有名」なのである。写真は、この論文を「紹介」する、チツパラ。
-

Q: Aren't These Proofs Too Boring for Mortals?

It is argued that formal verifications of programs, no matter how obtained, will not play the same key role in the development of computer science and software engineering as proofs do in mathematics. Furthermore the absence of continuity, the inevitability of change, and the complexity of specification of significantly many real programs make the formal verification process difficult to justify and manage.

– De Millo, Lipton, and Perlis,
“Social Processes and Proofs of
Theorems and Programs,” CACM, 1979



12



34C3.
tUwAt!

ダイクストラの21世紀へのメッセージ

「ライプニッツの夢の魔力の下で」 ダイクストラの最後のメッセージ

- ダイクストラは、21世紀の初めまで生き、2002年になくなる。最晩年の2001年に彼があらわした「ライプニッツの夢の魔力の下で」"Under the spell of Leibniz's Dream" は、自身の生涯を振り返りつつ、コンピュータ・サイエンスについての彼の立場を語った、いわば彼の「遺言」ともいえる文書である。

<https://www.cs.utexas.edu/.../transcript.../EWD12xx/EWD1298.html>

「ライブニッツの夢の魔力の下で」 ダイクストラの最後のメッセージ

- 論点は多岐にわたるが、いずれも刺激的である。
 - コンピュータ・サイエンスでの「理論」と「実践」の関係
 - コンピュータ・サイエンスとコンピュータ企業の関係
 - 大学の役割とその変化
 - プログラミングは「誰でもできる」のか？
 -
 - この文書が「刺激的」なのは、我々多数の「通念」とは、ほとんど逆のことを彼が述べているからである。おそらく反発する人も多いだろう。
-

「ライブニッツの夢の魔力の下で」 ダイクストラの最後のメッセージ

- ここでは、それらを紹介することはないが、こうした主張がコンピュータの世界に存在していることを知ることは、「未来の変化」を展望した時に、大きな意味があると僕は考えている。なぜなら、未来は現在の連続的延長上にあるとは限らず、変化は、現在の「常識」のいくつかを覆すだろうから。
 - 文章は平易である。論旨も明確である。興味ある人は、是非、目を通していただければと思う。
 - ここでは、この文書で「形式的手法」について、彼が述べていることのみを紹介したい。いずれも大事な指摘である。
 - ほとんど20年前の文書だが、これらの指摘は、現代の「形式的手法」が進もうとしている道を、的確に指し示している。
-

「ライブニッツの夢の魔力の下で」 ダイクストラの最後のメッセージ

1. 全ての場合を尽くすテストが不可能な時には、すなわち、ほとんどの場合はそうなのだが、我々の信頼は、(それが機械的なものであろうとなかろうと)証明の上にのみ基礎付けられる。
 2. 我々が、それを信ずべき明確な方法のないプログラムは、疑わしい価値しか持たない。
 3. プログラムは、その正しさについての議論が分かりやすく、かつ、不必要に複雑にならないように構造化されるべきである。
 4. 証明が与えられている時、その証明によって正当化されるプログラムを導き出す方が、プログラムが与えられて、それを正当化する証明を構成するより、はるかに簡単である。
-

証明支援システム Coq

形式的手法の Lingua Franca

Coqの開発

- ❑ Coqの開発は、Thierry Coquand とGérard Huetによって1984年から、INRIA (Institut National de Recherche en Informatique et en Automatique フランス国立情報学自動制御研究所) で開始された。その後、Christine Paulin-Mohringが加わり、40人以上の協力者がいる。
- ❑ 最初の公式のリリースは、拡張された素朴な帰納タイプのCoC 4.10で1989年。1991年には、Coqと改名された。
- ❑ それ以来、ユーザーのコミュニティは増大を続けており、豊かな型付き言語として、また、インタラクティブな定理証明システムとして、Coqのコンセプトのオリジナリティとその様々な特徴に魅了されている。

-
- 2013年には、ACMのSIGPLANのProgramming Languages Software Awardを授賞した。
 - Coqは、数学・セマンティックス・プログラムの検証の、形式的な正確性の保証の新しい時代への移行の中で、本質的に重要な役割を果たしているのである。
-

関連リンク

- ❑ **The Coq Proof Assistant**
<https://coq.inria.fr/>
- ❑ **Coq Github**
<https://github.com/coq/coq/>
- ❑ **Coq Reference Manual**
<https://coq.inria.fr/distrib/current/refman/>
- ❑ **“Certified Programming with Dependent Types”**
<http://adam.chlipala.net/cpdt/>
- ❑ **“Formal Reasoning About Programs”**
<http://adam.chlipala.net/frap/>

Deep Specificationと Software Foundations

<https://deepspec.org/page/SF/>

Volume 1

Logical Foundations is the entry-point to the series. It covers functional programming, basic concepts of logic, computer-assisted theorem proving, and Coq.



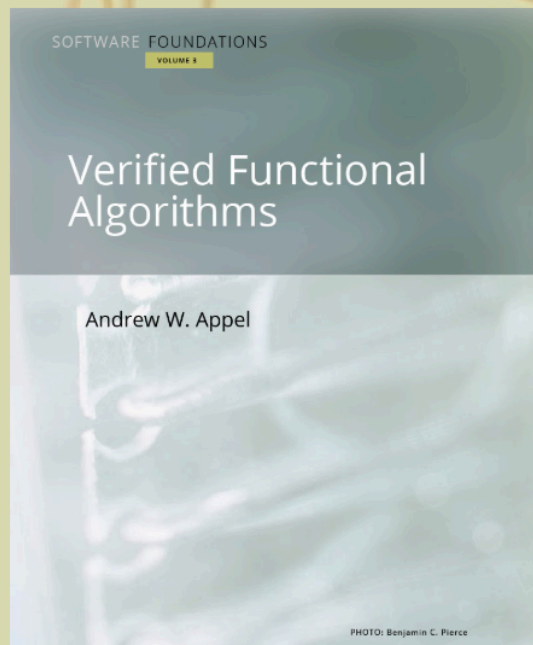
Volume 2

Programming Language Foundations surveys the theory of programming languages, including operational semantics, Hoare logic, and static type systems.



Volume 3

Verified Functional Algorithms shows how a variety of fundamental data structures can be specified and mechanically verified.



Volume 4

QuickChick: Property-Based Testing in Coq introduces tools and techniques for combining randomized property-based testing with formal specification and proof in the Coq ecosystem.



Software Foundations

□ **Volume 1: Logical Foundations**

<https://softwarefoundations.cis.upenn.edu/lf-current/toc.html>

PDF版

<https://softwarefoundations.cis.upenn.edu/lf-current/lf.tgz>

□ **Volume 2: Programming Language Foundations**

<https://softwarefoundations.cis.upenn.edu/plf-current/toc.html>

PDF版

<https://softwarefoundations.cis.upenn.edu/plf-current/plf.tgz>

Software Foundations

□ **Volume 3: Verified Functional Algorithms**

<https://softwarefoundations.cis.upenn.edu/vfa-current/toc.html>

PDF版

<https://softwarefoundations.cis.upenn.edu/vfa-current/vfa.tgz>

□ **Volume 4: QuickChick: Property-Based Testing in Coq**

<https://softwarefoundations.cis.upenn.edu/qc-current/toc.html>

PDF版

<https://softwarefoundations.cis.upenn.edu/qc-current/qc.tgz>
