Interference freedom

In computer science, **interference freedom** is a technique for proving partial correctness of concurrent programs with shared variables. Hoare logic had been introduced earlier to prove correctness of sequential programs. In her PhD thesis[1] (and papers arising from it [2][3]) under advisor David Gries, Susan Owicki extended this work to apply to concurrent programs.

Concurrent programming had been in use since the mid 1960s for coding operating systems as sets of concurrent processes (see, in particular, Dijkstra.[4]), but there was no formal mechanism for proving correctness. Reasoning about interleaved execution sequences of the individual processes was difficult, was error prone, and didn't scale up. Interference freedom applies to *proofs* instead of execution sequences; one shows that execution of one process cannot interfere with the correctness proof of another process.

A range of intricate concurrent programs have been proved correct using interference freedom, and interference freedom provides the basis for much of the ensuing work on developing concurrent programs with shared variables and proving them correct. The Owicki-Gries paper *An axiomatic proof technique for parallel programs I* [2] received the 1977 ACM Award for best paper in programming languages and systems.[5][6]

Note. Lamport [7] presents a similar idea. He writes, "After writing the initial version of this paper, we learned of the recent work of Owicki.[1][2]" His paper has not received as much attention as Owicki-Gries, perhaps because it used flow charts instead of the text of programming constructs like the **if** statement and **while** loop. Lamport was generalizing Floyd's method [8] while Owicki-Gries was generalizing Hoare's method.[9] Essentially all later work in this area uses text and not flow charts. Another difference is mentioned below in the section on Auxiliary variables.

Dijkstra's Principle of non-interference

Edsger W. Dijkstra introduced the *principle of non-interference* in EWD 117, "Programming Considered as a Human Activity", written about 1965.[10] This principle states that: The correctness of the whole can be established by taking into account only the exterior specifications (abbreviated *specs* throughout) of the parts, and not their interior construction. Dijkstra outlined the general steps in using this principle:

1. Give a complete spec of each individual part.
2. Check that the total problem is solved when program parts meeting their specs are available.
3. Construct the individual parts to satisfy their specs, but independent of one another and the context in which they will be used.

He gave several examples of this principle outside of programming. But its use in programming is a main concern. For example, a programmer using a method (subroutine, function, etc.) should rely only on its spec to determine what it does and how to call it, and *never* on its implementation.

Program specs are written in Hoare logic, introduced by Sir Tony Hoare,[9] as exemplified in the specs of processes S1 and S2:

{pre-S1} {pre-S2}
S1 S2
{post-S1} {pre-S2}

Meaning: If execution of Si in a state in which precondition pre-Si is true terminates, then upon termination, postcondition post-Si is true.

Now consider concurrent programming with shared variables. The specs of two (or more) processes S1 and S2 are given in terms of their pre- and post-conditions, and we assume that implementations of S1 and S2 are given that satisfy their specs. But when executing their implementations in parallel, since they share variables, a race condition can occur; one process changes a shared variable to a value that is not anticipated in the proof of the other process, so the other process does not work as intended.

Thus, Dijkstra's Principle of non-interference is violated.

In her PhD thesis of 1975 [1] in Computer Science, Cornell University, written under advisor David Gries, Susan Owicki developed the notion of *interference freedom*. If processes S1 and S2 satisfy interference freedom, then their parallel execution will work as planned. Dijkstra called this work the first significant step toward applying Hoare logic to concurrent processes.[11] To simplify discussions, we restrict attention to only two concurrent processes, although Owicki-Gries[2][3] allows more.

Interference freedom in terms of proof outlines

Owicki-Gries[2][3] introduced the *proof outline* for a Hoare triple {P}S{Q}. It contains all details needed for a proof of correctness of {P}S{Q} using the axioms and inference rules of Hoare logic. (This work uses the assignment statement x:= e, if-then and if-then-else statements, and the while loop.) Hoare alluded to proof outlines in his early work; for interference freedom, it had to be formalized.

A proof outline for {P}S{Q} begins with precondition P and ends with postcondition Q. Two assertions within braces { and } appearing next to each other indicates that the first must imply the second.

Example: A proof outline for {P} S {Q} where S is:
x:= a; **if** e **then** S1 **else** S2

{P}
{P1[x/a]}
x:= a;
{P1}
if e **then** {P1 ∧ e}
S1
{Q1}
else {P1 ∧ ¬ e}
S2

{Q1}

{Q1}
{Q}

$P \Rightarrow P1[x/a]$ must hold, where $P1[x/a]$ stands for $P1$ with every occurrence of x replaced by a. (In this example, $S1$ and $S2$ are basic statements, like an assignment statement, **skip**, or an **await** statement.)

Each statement T in the proof outline is preceded by a precondition $pre\text{-}T$ and followed by a postcondition $post\text{-}T$, and $\{pre\text{-}T\}T\{post\text{-}T\}$ must be provable using some axiom or inference rule of Hoare logic. Thus, the proof outline contains all the information necessary to prove that $\{P\}S\{Q\}$ is correct.

Now consider two processes $S1$ and $S2$ executing in parallel, and their specs:

{pre-S1}	{pre-S2}
S1	S2
{post-S1}	{post-S2}

Proving that they work suitably in parallel will require restricting them as follows. Each expression E in $S1$ or $S2$ may refer to at most one variable y that can be changed by the other process while E is being evaluated, and E may refer to y at most once. A similar restriction holds for assignment statements $x := E$.

With this convention, the only indivisible action need be the memory reference. For example, suppose process $S1$ references variable y while $S2$ changes y. The value $S1$ receives for y must be the value before or after $S2$ changes y, and not some spurious in-between value.

Definition of Interference-free

The important innovation of Owicki-Gries was to define what it means for a statement T not to interfere with the *proof* of $\{P\}S\{Q\}$. If execution of T cannot falsify any assertion given in the proof outline of $\{P\}S\{Q\}$, then that proof still holds even in the face of concurrent execution of S and T.

Definition. Statement T with precondition $pre\text{-}T$ does not interfere with the proof of $\{P\}S\{Q\}$ if two conditions hold:

(1) $\{Q \wedge pre\text{-}T\}\ T\ \{Q\}$
(2) Let S' be any statement within S but not within an await statement (see later section). Then $\{pre\text{-}S' \wedge pre\text{-}T\}\ T\ \{pre\text{-}S'\}$.

Read the last Hoare triple like this: If the state is such that both T and S' can be executed, then execution of T is not going to falsify $pre\text{-}S'$.

Definition. Proof outlines for $\{P1\}S1\{Q1\}$ and $\{P2\}S2\{Q2\}$ are interference-free if the following holds. Let T be an await or assignment statement (that does not appear in an await) of process $S1$. Then T does not interfere with the proof of $\{P2\}S2\{Q2\}$. Similarly for T of process $S2$ and $\{P1\}S1\{Q1\}$.

Statements cobegin and await

Two statements were introduced to deal with concurrency. Execution of the statement **cobegin** S1 // S2 **coend** executes S1 and S2 in parallel. It terminates when both S1 and S2 have terminated.

Execution of the **await** statement **await** B **then** S is delayed until condition B is true. Then, statement S is executed as an indivisible action—evaluation of B is part of that indivisible action. If two processes are waiting for the same condition B, when it becomes true, one of them continues waiting while the other proceeds.

The **await** statement cannot be implemented efficiently and is not proposed to be inserted into the programming language. Rather it provides a means of representing several standard primitives such as semaphores—first express the semaphore operations as **await**s, then apply the techniques described here.

Inference rules for **await** and **cobegin** are:

await

$$\frac{\{P \wedge B\}\ S\ \{Q\}}{\{P\}\ \textbf{await}\ B\ \textbf{then}\ S\ \{Q\}}$$

cobegin

$$\frac{\{P1\}\ S1\ \{Q1\},\ \{P2\}\ S2\ \{Q2\}\quad \text{interference-free}}{\{P1 \wedge P2\}\ \textbf{cobegin}\ S1//S2\ \textbf{coend}\ \{Q1 \wedge Q2\}}$$

Auxiliary variables

An *auxiliary variable* does not occur in the program but is introduced in the proof of correctness to make reasoning simpler —or even possible. Auxiliary variables are used only in assignments to auxiliary variables, so their introduction neither alters the program for any input nor affects the values of program variables. Typically, they are used either as program counters or to record histories of a computation.

Definition. Let AV be a set of variables that appear in S' only in assignments x:= E, where x is in AV. Then AV is an *auxiliary variable set* for S'.

Since a set AV of auxiliary variables are used only in assignments to variables in AV, deleting all assignments to them doesn't change the program's correctness, and we have the inference rule AV elimination:

$$\frac{\{P\}\ S'\ \{Q\}}{\{P\}\ S\ \{Q\}}$$

AV is an auxiliary variable set for S'. The variables in AV do not occur in P or Q. S is obtained from S' by deleting all assignments to the variables in AV.

Instead of using auxiliary variables, one can introduce a program counter into the proof system, but that adds complexity to the proof system.

Note: Apt [12] discusses the Owicki-Gries logic in the context of *recursive* assertions, that is, *effectively computable* assertions. He proves that all the assertions in proof outlines can be recursive, but that this is no longer the case if auxiliary variables are used only as program counters and not to record histories of computation. Lamport, in his similar work,[7] uses assertions about token positions instead of auxiliary variables, where a token on an edge of a flow chart is akin to a program counter. There is no notion of a history variable. This indicates that Owicki-Gries and Lamport's approach are not equivalent when restricted to recursive assertions.

Deadlock and termination

Owicki-Gries[2][3] deals mainly with partial correctness: {P} S {Q} means: If S executed in a state in which P is true terminates, then Q is true of the state upon termination. However, Owicki-Gries also gives some practical techniques that use information obtained from a partial correctness proof to derive other correctness properties, including freedom from deadlock, program termination, and mutual exclusion.

A program is in deadlock if all processes that have not terminated are executing **await** statements and none can proceed because their **await** conditions are false. Owicki-Gries provides conditions under which deadlock cannot occur.

Owicki-Gries presents an inference rule for total correctness of the **while** loop. It uses a bound function that decreases with each iteration and is positive as long as the loop condition is true. Apt *et al* [13] show that this new inference rule does not satisfy interference freedom. The fact that the bound function is positive as long as the loop condition is true was not included in an interference test. They show two ways to rectify this mistake.

A simple example

Consider the statement:

```
{x=0}
cobegin await true then x:= x+1
        // await true then x:= x+2
coend
{x=3}
```

The proof outline for it:

```
{x=0}
S: cobegin
            {x=0}
            {x=0 ∨ x=2}
            S1: await true then x:= x+1
            {Q1: x=1 ∨ x=3}
      //
            {x=0}
            {x=0 ∨ x=1}
```

S2: **await true then** x:= x+2
{Q2: x=2 ∨ x=3}
coend
{(x=1 ∨ x=3) ∧ (x=2 ∨ x=3)}
{x=3}

Proving that S1 does not interfere with the proof of S2 requires proving two Hoare triples:

(1) {(x=0 ∨ x=2) ∧ (x=0 ∨ x=1} S1 {x=0 ∨ x=1}
(2) {(x=0 ∨ x=2) ∧ (x=2 ∨ x=3} S1 {x=2 ∨ x=3}

The precondition of (1) reduces to x=0 and the precondition of (2) reduces to x=2. From this, it is easy to see that these Hoare triples hold. Two similar Hoare triples are required to show that S2 does not interfere with the proof of S1.

Suppose S1 is changed from the **await** statement to the assignment x:= x+1. Then the proof outline does not satisfy the requirements, because the assignment contains two occurrences of shared variable x. Indeed, the value of x after execution of the **cobegin** statement could be 2 or 3.

Suppose S1 is changed to the **await** statement **await true then** x:= x+2, so it is the same as S2. After execution of S, x should be 4. To prove this, because the two assignments are the same, two auxiliary variables are needed, one to indicate whether S1 has been executed; the other, whether S2 has been executed. We leave the change in the proof outline to the reader.

Examples of formally proved concurrent programs

A. Findpos. Write a program that finds the first positive element of an array (if there is one). One process checks all array elements at even positions of the array and terminates when it finds a positive value or when none is found. Similarly, the other process checks array elements at odd positions of the array. Thus, this example deals with **while** loops. It also has no **await** statements.

This example comes from Barry K. Rosen.[14] The solution in Owicki-Gries,[2] complete with program, proof outline, and discussion of interference freedom, takes less than two pages. Interference freedom is quite easy to check, since there is only one shared variable. In contrast, Rosen's article[14] uses Findpos as the single, running example in this 24 page paper.

An outline of both processes in a general environment:

cobegin producer: ...
await in-out < N **then skip**;
add: b[in **mod** N]:= next value;
markin: in:= in+1;
...
//
consumer: ...
await in-out > 0 **then skip**;
remove: this value:=
b[out **mod** N];

```
            markout: out:= out+1;
coend
...
```

B. Bounded buffer consumer/producer problem. A producer process generates values and puts them into bounded buffer b of size N; a consumer process removes them. They proceed at variable rates. The producer must wait if buffer b is full; the consumer must wait if buffer b is empty. In Owicki-Gries,[2] a solution in a general environment is shown; it is then embedded in a program that copies an array c[1..M] into an array d[1..M].

This example exhibits a principle to reduce interference checks to a minimum: Place as much as possible in an assertion that is invariantly true everywhere in both processes. In this case the assertion is the definition of the bounded buffer and bounds on variables that indicate how many values have been added to and removed from the buffer. Besides buffer b itself, two shared variables record the number in of values added to the buffer and the number out removed from the buffer.

C. Implementing semaphores. In his article on the THE multiprogramming system,[4] Dijkstra introduces the semaphore sem as a synchronization primitive: sem is an integer variable that can be referenced in only two ways, shown below; each is an indivisible operation:

1. P(sem): Decrease sem by 1. If now sem < 0, suspend the process and put it on a list of suspended processes associated with sem.

2. V(sem): Increase sem by 1. If now sem ≤ 0, remove one of the processes from the list of suspended processes associated with sem, so its dynamic progress is again permissible.

The implementation of P and V using **await** statements is:

```
P(sem):
        await true then
            begin sem:= sem-1;
                if sem < 0 then
                    w[this process]:= true
            end;
            await ¬w[this process]
                then skip

V(sem):
        await true then
            begin sem:= sem+1;
                if sem ≤ 0 then
                    begin choose p such
                                                that w[p];
                                        w[p]:= false
                    end
            end
```

Here, w is an array of processes that are waiting because they have been suspended; initially, w[p]=**false** for every process p. One could change the implementation to always waken the longest suspended process.

D. On-the-fly garbage collection. At the 1975 Summer School Marktoberdorf, Dijkstra discussed an on-the-fly garbage collector as an exercise in understanding parallelism. The data structure used in a conventional implementation of LISP is a directed graph in which each node has at most two outgoing edges, either of which may be missing: an outgoing left edge and an outgoing right edge. All nodes of the graph must be reachable from a known root. Changing a node may result in unreachable nodes, which can no longer be used and are called *garbage*. An on-the-fly garbage collector has two processes: the program itself and a garbage collector, whose task is to identify garbage nodes and put them on a free list so that they can be used again.

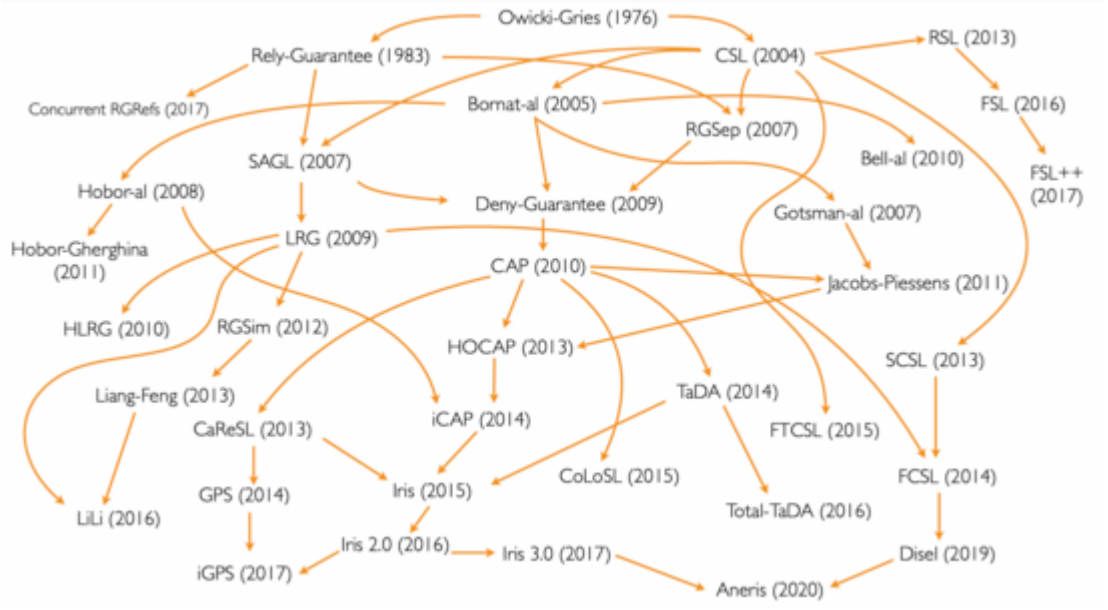
Gries felt that interference freedom could be used to prove the on-the-fly garbage collector correct. With help from Dijkstra and Hoare, he was able to give a presentation at the end of the Summer School, which resulted in an article in CACM.[15]

E. Verification of readers/writers solution with semaphores. Courtois et al[16] use semaphores to give two versions of the readers/writers problem, without proof. Write operations block both reads and writes, but read operations can occur in parallel. Owicki[17] provides a proof.

F. Peterson's algorithm, a solution to the 2-process mutual exclusion problem, was published by Peterson in a 2-page article.[18] Schneider and Andrews provide a correctness proof.[19]

Dependencies on interference freedom

The image below, by Ilya Sergey, depicts the flow of ideas that have been implemented in logics that deal with concurrency. At the root is interference freedom. The file *CSL-Family-Tree* (https://ilyasergey.net/assets/other/CSL-Family-Tree.pdf) (PDF) contains references. Below, we summarize the major advances.



- **Rely-Guarantee**. 1981. Interference freedom is not compositional. Cliff Jones[20][21] recovers compositionality by abstracting interference into two new predicates in a spec: a

rely-condition records what interference a thread must be able to tolerate and a guarantee-condition sets an upper bound on the interference that the thread can inflict on its sibling threads. Xu *et al* [22] observe that Rely-Guarantee is a reformulation of interference freedom; revealing the connection between these two methods, they say, offers a deep understanding about verification of shared variable programs.

- **CSL**. 2004. Separation logic supports local reasoning, whereby specifications and proofs of a program component mention only the portion of memory used by the component. Concurrent separation logic (CSL) was originally proposed by Peter O'Hearn,[23][24] We quote from:[23] "the Owicki-Gries method[2] involves explicit checking of non-interference between program components, while our system rules out interference in an implicit way, by the nature of the way that proofs are constructed."
- **Deriving concurrent programs**. 2005-2007. Feijen and van Gasteren[25] show how to use Owicki-Gries to design concurrent programs, but the lack of a theory of progress means that designs are driven only by safety requirements. Dongol, Goldson, Mooij, and Hayes have extended this work to include a "logic of progress" based on Chandy and Misra's language Unity, molded to fit a sequential programming model. Dongel and Goldson[26] describe their logic of progress. Goldson and Dongol[27] show how this logic is used to improve the process of designing programs, using Dekker's algorithm for two processes as an example. Dongol and Mooij[28] present more techniques for deriving programs, using Peterson's mutual exclusion algorithm as one example. Dongol and Mooij[29] show how to reduce the calculational overhead in formal proofs and derivations and derive Dekker's algorithm again, leading to some new and simpler variants of the algorithm. Mooij[30] studies calculational rules for Unity's *leads-to* relation. Finally, Dongol and Hayes[31] provide a theoretical basis for and prove soundness of the process logic.
- **OGRA**. 2015. Lahav and Vafeiadis strengthen the interference freedom check to produce (we quote from the abstract) "OGRA, a program logic that is sound for reasoning about programs in the release-acquire fragment of the C11 memory model." They provide several examples of its use, including an implementation of the RCU synchronization primitives.[32]
- **Quantum programming**. 2018. Ying *et al* [33] extend interference freedom to quantum programming. Difficulties they face include intertwined nondeterminism: nondeterminism involving quantum measurements and nondeterminism introduced by parallelism occurring at the same time. The authors formally verify Bravyi-Gosset-König's parallel quantum algorithm solving a linear algebra problem, giving, they say, for the first time an unconditional proof of a computational quantum advantage.
- **POG**. 2020. Raad *et al* present POG (Persistent Owicki-Gries), the first program logic for reasoning about non-volatile memory technologies, specifically the Intel-x86.[34]

Texts that discuss interference freedom

- *On A Method of Multiprogramming*, 1999.[25] Van Gasteren and Feijen discuss the formal development of concurrent programs entirely on the idea of interference freedom.
- *On Current Programming*, 1997.[35] Schneider uses interference freedom as the main tool in developing and proving concurrent programs. A connection to temporal logic is given, so arbitrary safety and liveness properties can be proven. Control predicates obviate the need for auxiliary variables for reasoning about program counters.
- *Verification of Sequential and Concurrent Programs*, 1991,[36] 2009.[37] This first text to cover verification of structured concurrent programs, by Apt *et al*, has gone through several editions over several decades.
- *Concurrency Verification: Introduction to Compositional and Non-Compositional Methods*, 2112.[38] De Roever *et al* provide a systematic and comprehensive introduction to

compositional and non-compositional proof methods for the state-based verification of concurrent programs

Implementations of interference freedom

- 1999: Nipkow and Nieto present the first formalization of interference freedom and its compositional version, the rely-guarantee method, in a theorem prover: Isabelle/HOL.[39][40]
- 2005: Ábrahám's PhD thesis provides a way to prove multithreaded Java programs correct in three steps: (1) Annotate the program to produce a proof outline, (2) Use their tool *Verger* to automatically create verification conditions, and (3) Use the theorem prover PVS to prove the verification conditions interactively.[41][42]
- 2017: Denissen[43] reports on an implementation of Owicki-Gries in the "verification ready" programming language Dafny.[44] Denissen remarks on the ease of use of Dafny and his extension to it, making it extremely suitable when teaching students about interference freedom. Its simplicity and intuitiveness outweighs the drawback of being non-compositional. He lists some twenty institutions that teach interference freedom.
- 2017: Amani *et al* combine the approaches of Hoare-Parallel, a formalisation of Owicki-Gries in Isabelle/HOL for a simple while-language, and SIMPL, a generic language embedded in Isabelle/HOL, to allow formal reasoning on C programs.[45]
- 2022: Dalvandi *et al* introduce the first deductive verification environment in Isabelle/HOL for C11-like weak memory programs, building on Nipkow and Nieto's encoding of Owicki–Gries in the Isabelle theorem prover.[46]
- 2022: This webpage [47] describes the Civl verifier for concurrent programs and gives instructions for installing it on your computer. It is built on top of Boogie, a verifier for sequential programs. Kragl et al [48] describe how interference freedom is achieved in Civl using their new specification idiom, *yield invariants*. One can also use specs in the rely-guarantee style. Civl offers a combination of linear typing and logic that allows economical and local reasoning about disjointness (like separation logic). Civl is the first system that offers refinement reasoning on structured concurrent programs.
- 2022. Esen and Rümmer developed TRICERA,[49] an automated open-source verification tool for C programs. It is based on the concept of constrained Horn clauses, and it handles programs operating on the heap using a theory of heaps. A web interface to try it online is available. To handle concurrency, TRICERA uses a variant of the Owicki-Gries proof rules, with explicit variables to added to represent time and clocks.

References

1. Owicki, Susan S. (August 1975). *Axiomatic Proof Techniques for Parallel Programs* (https://hdl.handle.net/1813/6393) (PhD thesis). Cornell University. hdl:1813/6393 (https://hdl.handle.net/1813%2F6393). Retrieved 2022-07-01.
2. Owicki, Susan; Gries, David (25 June 1976). "An axiomatic proof technique for parallel programs I" (https://doi.org/10.1007/BF00268134). *Acta Informatica*. **6** (4). Berlin: Springer (Germany): 319–340. doi:10.1007/BF00268134 (https://doi.org/10.1007%2FBF00268134). S2CID 206773583 (https://api.semanticscholar.org/CorpusID:206773583).
3. Owicki, Susan; Gries, David (May 1976). "Verifying properties of parallel programs: an axiomatic approach" (https://doi.org/10.1145%2F360051.360224). *Communications of the ACM*. **19** (5): 279–285. doi:10.1145/360051.360224 (https://doi.org/10.1145%2F360051.360224). S2CID 9099351 (https://api.semanticscholar.org/CorpusID:9099351).

4. Dijkstra, E.W. (1968), "The structure of the 'THE'-multiprogramming system", *Communications of the ACM*, **11** (5): 341–346, doi:10.1145/363095.363143 (https://doi.org/10.1145%2F363095.363143), S2CID 2021311 (https://api.semanticscholar.org/CorpusID:2021311)
5. "Susan S Owicki" (https://awards.acm.org/award_winners/owicki_1219047). Awards.acm.org. Retrieved 2022-07-01.
6. "David Gries" (https://awards.acm.org/award-winners/GRIES_1028422). Awards.acm.org. Retrieved 2022-07-01.
7. Lamport, Leslie (March 1977). "Proving the correctness of multiprocess programs" (https://ieeexplore.ieee.org/document/1702415). *IEEE Transactions on Software Engineering*. **SE-3** (2): 125–143. doi:10.1109/TSE.1977.229904 (https://doi.org/10.1109%2FTSE.1977.229904). S2CID 9985552 (https://api.semanticscholar.org/CorpusID:9985552).
8. Floyd, Robert W. (1967). "Assigning Meanings to Programs" (https://people.eecs.berkeley.edu/~necula/Papers/FloydMeaning.pdf) (PDF). In Schwartz, J.T. (ed.). *Mathematical Aspects of Computer Science* (https://books.google.com/books?id=ynigSICJflYC). Proceedings of Symposium on Applied Mathematics. Vol. 19. American Mathematical Society. pp. 19–32. ISBN 0821867288.
9. Hoare, C. A. R. (October 1969). "An axiomatic basis for computer programming" (https://doi.org/10.1145%2F363235.363259). *Communications of the ACM*. **12** (10): 576–580. doi:10.1145/363235.363259 (https://doi.org/10.1145%2F363235.363259). S2CID 207726175 (https://api.semanticscholar.org/CorpusID:207726175).
10. "Programming Considered as a Human Activity" (https://www.cs.utexas.edu/~EWD/ewd01xx/EWD117.PDF) (PDF). *E. W. Dijkstra Archive*. University of Texas.
11. Dijkstra, Edsger W. (1982). "EWD 554: A personal summary of the Gries-Owicki Theory". *Selected Writings on Computing: A Personal Perspective*. Monographs in Computer Science. Springer-Verlag. pp. 188–199. ISBN 0387906525.
12. Apt, Krzysztof R. (June 1981). "Recursive assertions and parallel programs" (https://doi.org/10.1007/BF00289262). *Acta Informatica*. **15** (3): 219–232. doi:10.1007/BF00289262 (https://doi.org/10.1007%2FBF00289262). S2CID 42470032 (https://api.semanticscholar.org/CorpusID:42470032).
13. Apt, Krzysztof R.; de Boer, Frank S.; Olderog, Ernst-Rüdiger (1990). "Proving termination of parallel programs". In Gries, D.; Feijen, W.H.J.; van Gasteren, A.J.M.; Misra, J. (eds.). *Beauty is Our Business* (https://link.springer.com/book/10.1007/978-1-4612-4476-9). Monographs in Computer Science. New York: Springer Verlag. pp. 0–6. doi:10.1007/978-1-4612-4476-9 (https://doi.org/10.1007%2F978-1-4612-4476-9). ISBN 978-1-4612-8792-6. S2CID 24379938 (https://api.semanticscholar.org/CorpusID:24379938).
14. Rosen, Barry K (1976). "Correctness of parallel programs: The Church-Rosser Approach" (https://doi.org/10.1016%2F0304-3975%2876%2990032-3). *Theoretical Computer Science*. **2** (2): 183–207. doi:10.1016/0304-3975(76)90032-3 (https://doi.org/10.1016%2F0304-3975%2876%2990032-3).
15. Gries, David (December 1977). "An exercise in proving parallel programs correct" (https://doi.org/10.1145%2F359897.359903). *Communications of the ACM*. **20** (12): 921–930. doi:10.1145/359897.359903 (https://doi.org/10.1145%2F359897.359903). S2CID 3202388 (https://api.semanticscholar.org/CorpusID:3202388).
16. Courtois, P.J.; Heymans, F.; Parnas, D.L. (October 1971). "Concurrent control with "readers" and "writers" " (https://doi.org/10.1145%2F362759.362813). *Communications of the ACM*. **14** (10): 667–668. doi:10.1145/362759.362813 (https://doi.org/10.1145%2F362759.362813). S2CID 7540747 (https://api.semanticscholar.org/CorpusID:7540747).
17. Owicki, Susan (August 1977). *Verifying concurrent programs with shared data classes* (http://i.stanford.edu/pub/cstr/reports/csl/tr/77/147/CSL-TR-77-147.pdf) (PDF) (Technical report). Digital Systems Laboratory, Stanford University. 147. Retrieved 2022-07-08.

18. Peterson, Gary L. (June 1981). "Myths about the mutual exclusion problem" (https://dx.doi.org/10.1016/0020-0190%2881%2990106-X). *IPL*. **12** (3): 115–116. doi:10.1016/0020-0190(81)90106-X (https://doi.org/10.1016%2F0020-0190%2881%2990106-X).
19. Schneider, Fred B.; Andrews, Gregory R. (1986). "Concepts for concurrent programming" (https://doi.org/10.1007/BFb0027049). In J.W. Bakker; W.P. de Roever; G. Rozenberg (eds.). *Current Trends in Concurrency*. Lecture Notes in Computer Science. Vol. 224. Noordwijkerhout, the Netherlands: Springer Verlag. pp. 669–716. doi:10.1007/BFb0027049 (https://doi.org/10.1007%2FBFb0027049). ISBN 978-3-540-16488-3.
20. Jones, C.B. (June 1981). *Development Methods for Computer Programs including a Notion of Interference* (http://www.cs.ox.ac.uk/files/9025/PRG-25.pdf) (PDF) (DPhil thesis). Oxford University.
21. Jones, Cliff B. (1983). R.E.A. Mason (ed.). *Specification and design of (parallel) programs*. 9th IFIP World Computer Congress (Information Processing 83). North-Holland/IFIP. pp. 321–332. ISBN 0444867295.
22. Xu, Qiwen; de Roever, Willem-Paul; He, Jifeng (1997). "The Rely-Guarantee method for verifying shared variable concurrent programs" (https://doi.org/10.1007/BF01211617). *Formal Aspects of Computing*. **9** (2): 149–174. doi:10.1007/BF01211617 (https://doi.org/10.1007%2FBF01211617). S2CID 12148448 (https://api.semanticscholar.org/CorpusID:12148448).
23. O'Hearn, Peter W. (2004-09-03). "Resources, Concurrency and Local Reasoning" (https://link.springer.com/book/10.1007/b100113). In P. Gardner; N. Yoshida (eds.). *CONCUR 2004 -- Concurrency Theory*. CONCUR 2004. London, UK: Springer Verlag Berlin, Heidelberg. pp. 49–67. doi:10.1007/b100113 (https://doi.org/10.1007%2Fb100113). ISBN 978-3-540-28644-8. Retrieved 2022-07-06.
24. O'Hearn, Peter (2007). "Resources, Concurrency and Local Reasoning" (http://www.cs.ucl.ac.uk/staff/p.ohearn/papers/concurrency.pdf) (PDF). *Theoretical Computer Science*. **375** (1–3): 271–307. doi:10.1016/j.tcs.2006.12.035 (https://doi.org/10.1016%2Fj.tcs.2006.12.035).
25. Van Gasteren, A.J.M.; Feijen, W.H.J. (1999). Gries, David; Schneider, Fred B. (eds.). *On A Method Of Multiprogramming*. Monographs in Computer Science. Springer-Verlag New York Inc. p. 370. doi:10.1007/978-1-4757-3126-2 (https://doi.org/10.1007%2F978-1-4757-3126-2). ISBN 978-1-4757-3126-2. S2CID 13607884 (https://api.semanticscholar.org/CorpusID:13607884).
26. Dongol, Brijesh; Goldson, Doug (2006) [January 12, 2005]. "Extending the theory of Owicki and Gries with a logic of progress" (https://doi.org/10.2168%2Flmcs-2%281%3A6%292006). *Logical Methods in Computer Science*. **2**. Centre pour la Communication Scientifique Directe (CCSD). arXiv:cs/0512012v3 (https://arxiv.org/abs/cs/0512012v3). doi:10.2168/lmcs-2(1:6)2006 (https://doi.org/10.2168%2Flmcs-2%281%3A6%292006). S2CID 302420 (https://api.semanticscholar.org/CorpusID:302420).
27. Goldson, Doug; Dongol, Brijesh (January 2005). "Concurrent Program Design in the Extended Theory of Owicki and Gries" (https://dl.acm.org/doi/pdf/10.5555/1082260.1082265). In Mike Atkinson; Frank Dehne (eds.). *CATS '05: Proc 2005 Australasian Symp on Theory of Computing*. Vol. 41. Australian Computer Society, Inc. pp. 41–50.
28. Dongol, Brijesh; Mooij, Arjan J (July 2006). "Progress in deriving concurrent programs: emphasizing the role of stable guards" (https://link.springer.com/content/pdf/10.1007/11783596_11). In Tarmo Uustalu (ed.). *MPC'06: Proc. 8th Int. Conf. on Mathematics of Program Construction*. Vol. 41. Kuressaare, Estonia: Springer Verlag, Berlin, Heidelberg. pp. 14–161. doi:10.1007/11783596_11 (https://doi.org/10.1007%2F11783596_11).
29. Dongol, Brijesh; Mooij, Arjan J (2008). "Streamlining progress-based derivations of concurrent program" (https://dl.acm.org/doi/pdf/10.1007/s00165-007-0037-4). *Formal Aspects of Computing*. **20** (2): 141–160. doi:10.1007/s00165-007-0037-4 (https://doi.org/10.1007%2Fs00165-007-0037-4). S2CID 7024064 (https://api.semanticscholar.org/CorpusID:7024064).

30. Mooij, Arjan J. (November 2007). "Calculating and composing progress properties in terms of the leads-to relation". In Michael Butler; Michael G. Hinchey; María M. Larrondo-Petrie (eds.). *ICFEM'07: Proc. Formal Engineering Methods 9th Int. Conf. on Formal Methods and Software Engineering*. Boca Raton, Florida: Springer Verlag, Berlin, Heidelberg. pp. 366–386. ISBN 978-3540766483.
31. Dongol, Brijesh; Hayes, Ian (April 2007). *Trace semantics for the Owicki-Gries theory integrated with the progress logic from UNITY* (https://core.ac.uk/download/pdf/14986002.pdf) (PDF) (Technical report). University of Queensland. SSE-2007-02.
32. Lahav, Ori; Vafeiadis, Viktor (2015). "Owicki-Gries reasoning for weak memory models" (https://link.springer.com/chapter/10.1007/978-3-662-47666-6_25). In Halldórsson, M.; Iwama, K.; Kobayashi, N.; Speckmann, B. (eds.). *Automata, Languages, and Programming. ICALP 2015*. ICALP 2015. Lecture Notes in Computer Science. Vol. 9135. Berlin, Heidelberg: Springer. pp. 311–323. doi:10.1007/978-3-662-47666-6_25 (https://doi.org/10.1007%2F978-3-662-47666-6_25).
33. Ying, Mingsheng; Zhou, Li; Li, Yangjia (2018). "Reasoning about Parallel Quantum Programs". arXiv:1810.11334 (https://arxiv.org/abs/1810.11334) [cs.LO (https://arxiv.org/archive/cs.LO)].
34. Raad, Azalea; Lahav, Ori; Vafeiadis, Viktor (13 November 2020). "Persistent Owicki-Gries reasoning: a program logic for reasoning about persistent programs on Intel-x86". *Proceedings of the ACM on Programming Languages*. Vol. 4. ACM. pp. 1–28. doi:10.1145/3428219 (https://doi.org/10.1145%2F3428219). hdl:10044/1/97398 (https://hdl.handle.net/10044%2F1%2F97398).
35. Schneider, Fred B. (1997). Gries, David; Schneider, Fred B. (eds.). *On Concurrent Programming*. Graduate Texts in Computer Science. Springer-Verlag New York Inc. doi:10.1007/978-1-4612-1830-2 (https://doi.org/10.1007%2F978-1-4612-1830-2). ISBN 978-1-4612-1830-2. S2CID 9980317 (https://api.semanticscholar.org/CorpusID:9980317).
36. Apt, Krzysztof R.; Olderog, Ernst-Rüdiger (1991). Gries, David (ed.). *Verification of Sequential and Concurrent Programs*. Texts in Computer Science. Springer-Verlag Germany.
37. Apt, Krzysztof R.; Boer, Frank S.; Olderog, Ernst-Rüdiger (2009). Gries, David; Schneider, Fred B. (eds.). *Verification of Sequential and Concurrent Programs* (http://link.springer.com/10.1007/978-1-84882-745-5). Texts in Computer Science (3rd ed.). Springer-Verlag London. p. 502. Bibcode:2009vscp.book.....A (https://ui.adsabs.harvard.edu/abs/2009vscp.book.....A). doi:10.1007/978-1-84882-745-5 (https://doi.org/10.1007%2F978-1-84882-745-5). ISBN 978-1-84882-744-8.
38. de Roever, Willem-Paul; de Boer, Willem-Paul; Hanneman, Ulrich; Hooman, Jozef; Lakhnech, Yassine; Poel, Mannes; Zwiers, Job (2012). Abramsky, S. (ed.). *Concurrency Verification: Introduction to Compositional and Non-Compositional Methods*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press USA. p. 800. ISBN 978-0521169325.
39. Nieto, Leonor Prensa (2002-01-31). *Verification of Parallel Programs with the Owicki-Gries and Rely-Guarantee Methods in Isabelle/HOL* (https://tumb1.biblio.tu-muenchen.de/publ/diss/allgemein.html) (PhD thesis). Technischen Universitaet Muenchen. p. 198. Retrieved 2022-07-05.
40. Nipkow, Tobias; Nieto, Leonor Prensa (1999-03-22). "Owicki/Gries in Isabelle/HOL". In J.P. Finance (ed.). *Fundamental Approaches to Software Engineering*. FASE 1999. Lecture Notes in Computer Science. Vol. 1577. Berlin Heidelberg: Springer Verlag. pp. 188–203. doi:10.1007/978-3-540-49020-3_13 (https://doi.org/10.1007%2F978-3-540-49020-3_13). ISBN 978-3-540-49020-3.
41. Ábrahám, Erika (2005-01-20). *An Assertional Proof System for Multithreaded Java - Theory and Tool Support* (https://hdl.handle.net/1887/584) (PhD thesis). Universiteit Leiden. p. 220. hdl:1887/584 (https://hdl.handle.net/1887%2F584). ISBN 9090189084. Retrieved 2022-07-05.

42. Ábrahám, Erika; Boer, Frank, S., de; Roever, Willem-Paul, de; Martin, Steffen (2005-02-25). "An assertion-based proof system for multithreaded Java" (https://doi.org/10.1016%2Fj.tcs.2004.09.019). *Theoretical Computer Science*. **331** (2–3). Elsevier: 251–290. doi:10.1016/j.tcs.2004.09.019 (https://doi.org/10.1016%2Fj.tcs.2004.09.019).
43. Denissen, P.E.J.G (November 2017). *Extending Dafny to Concurrency: Owicki-Gries style program verification for the Dafny program verifier* (https://research.tue.nl/en/studentTheses/extending-dafny-to-concurrency) (Masters thesis). Eindhoven University of Technology.
44. "Dafny Programming Language" (https://dafny.org). Retrieved 2022-07-20.
45. Amani, S.; Andronick, J.; Bortin, M.; Lewis, C.; Rizkallah, C.; Tuong, J. (16 January 2017). Yves Bertot; Viktor Vafeiadid (eds.). *COMPLX: A verification framework for concurrent imperative programs* (https://dx.doi.org/10.1145/3018610.3018627). CPP 2017: Proc 6th ACM SIGPLAN Conference on Certified Programs and Proof. Paris, France: ACM. pp. 138–150. doi:10.1145/3018610.3018627 (https://doi.org/10.1145%2F3018610.3018627). ISBN 978-1-4503-4705-1.
46. Dalvandi, Sadegh; Dongol, Brijesh; Doherty, Simon; Wehrheim, Heike (February 2022). "Integrating Owicki–Gries for C11-Style memory models into Isabelle/HOL" (https://doi.org/10.1007%2Fs10817-021-09610-2). *Journal of Automated Reasoning*. **66**: 141–171. arXiv:2004.02983 (https://arxiv.org/abs/2004.02983). doi:10.1007/s10817-021-09610-2 (https://doi.org/10.1007%2Fs10817-021-09610-2). S2CID 215238874 (https://api.semanticscholar.org/CorpusID:215238874).
47. "Civl: A verifier for concurrent programs" (https://civl-verifier.github.io). Retrieved 2022-07-22.
48. Kragl, Bernhard; Qadeer, Shaz; Henzinger, Thomas A. (2020). "Refinement for structured concurrent programs". In S. Lahiri; C. Wang (eds.). *CAV 2020: Computer Aided Verification*. Lecture Notes in Computer Science. Vol. 12224. Springer Verlag. doi:10.1007/978-3-030-53288-8_14 (https://doi.org/10.1007%2F978-3-030-53288-8_14). ISBN 978-3-030-53288-8.
49. Esen, Zafer; Rümmer, Philipp (October 2022). "TRICERA Verifying C Programs Using the Theory of Heaps". In A. Griggio; N. Rungta (eds.). *Proc. 22nd Conf. on Formal Methods in Computer-Aided Design – FMCAD 2022*. TU Wien Academic Press. pp. 360–391. doi:10.34727/2022/isbn.978-3-85448-053-2_45 (https://doi.org/10.34727%2F2022%2Fisbn.978-3-85448-053-2_45).
