Go2Pins: A framework for the LTL verification of Go programs (Extended Version)

Alexandre Kirszenberg, Antoine Martin, Hugo Moreau, Étienne Renault

Noname manuscript No.
(will be inserted by the editor)

Go2Pins: a framework for the LTL verification of Go programs (Extended Version)

Alexandre Kirszenberg, Antoine Martin, Hugo Moreau, and Etienne Renault

Abstract We introduce Go2Pins, a tool that takes a program written in Go and links it with two model-checkers: LTSMin [26] and Spot [10]. Go2Pins is an effort to promote the integration of both formal verification and testing inside industrial-size projects. With this goal in mind, we introduce *black-box transitions*, an efficient and scalable technique for handling the Go runtime. This approach, inspired by hardware verification techniques, allows easy, automatic and efficient abstractions. Go2Pins also handles basic concurrent programs through the use of a dedicated scheduler. Moreover, in order to efficiently handle recursive programs, we introduce PSL_{REC}, a formalism that augments PSL without changing the complexity of the underlying verification process.

In this paper we demonstrate the usage of Go2Pins over benchmarks inspired by industrial problems and a set of LTL formulae. Even if Go2Pins is still at the early stages of development, our results are promising and show the the benefits of using black-box transitions. This paper also shows how Go2Pins is able to work efficiently on two bugs coming from industrial problems Kubernetes and Trillian.

1 Introduction & Motivation

The Go programming language was designed at Google in 2009 [23] to improve programming productivity in an era of multicore, networked machines and large codebases. Inspired by the idea of *Communicating Sequential Processes* (CSP) [24], designers focused on two principles: (1) having lightweight and easy to create threads (called goroutines) and, (2) promoting communication across threads by explicit messaging (through channels) rather than by shared memory. Even if other languages have also been designed to tackle similar problems (OCCAM and ERLANG), Go is probably the first large scale, widely used, industrial language to integrate these distinctive CSP features.

Previously (and except for OCCAM and ERLANG), mainly academic formal languages, implementing variations around the notion of CSP, have been developed: PROMELA, UPPAAL, DVE, GAL, CSP_{M}, etc. These languages have been built as a support for developing verification tools and their associated theory but have seldom been used in the industry.

The main idea defended in this paper is to consider the Go language not only as a disruptive, efficient, industrial, statically typed, compiled programming language but also as a good candidate for the specification and verification of asynchronous systems. Indeed, most of the time formal languages are only used for modeling and verification while the actual implementation of the system is done in another language for efficiency. This switch between languages is error-prone since bugs can be introduced at each level. Moreover, most formal languages do not have associated compilers or interpreters: this is annoying since the only way to test the validity of the model is to express the desired behaviors through a temporal logic [1].

This paper tackles these problems by introducing Go2Pins: a Go-based unified framework for testing, modeling, verification, and efficient implementation of systems. This paper also introduces black-box transitions (see Section 4), an efficient and scalable technique for handling the Go runtime. This approach, inspired by

EPITA Research Laboratory (LRE), Kremlin-Bicêtre, France
E-mail: {alexandre.kirszenberg, antoine4.martin, hugo.moreau, etienne.renault}@epita.fr

[1] Note that in the particular context of CSP, validity can also be checked using refinement.

hardware verification techniques, allows easy, automatic and efficient abstractions. Even if this idea is not new (premises of this technique are available the SPIN model checker), we extend it to be automatic, and then well suited for verifying large software systems.

In addition to the above (common with our previous paper [27] published at the SPIN'21 conference), this paper gives more clarification about the translation process (Section 3.1) and show how to add the support of structures, interfaces, and duck-typing in Go2Pins. This paper also test Go2Pins against two industrial problems (Section 9) and compares it to latest advances in the verification of Go programs (Section 8 and 9). Finally, this paper also introduces PSL_{REC}, a new formalism that augment PSL (a superset of LTL) without changing the complexity of the underlying verification process (Section 5 and 7).

2 Go2Pins: Overview

This section describes our journey towards the verification of Go programs. Figure 1 describes an overview of Go2Pins: the program to verify is processed by Go2Pins which produces a binary called **go2pins-mc**. This binary can then be used to verify any LTL formula (over the input program) using one of the two supported backends: LTSMin [26] or Spot [10]. These backends have been selected because (1) LTSMin is a widely used parallel verification tool that supports both explicit and symbolic model-checking and (2) because Spot is known to be an efficient modular framework for developing verification-based tools and techniques.

Figure 2 provides more details about the Go2Pins approach. At coarse grain, the input program is processed by the core of our tool and then translated into the Partitioned Next-State Interface (PINS) [26]. This interface exposes two functions: one for retrieving the initial state of the system, and one for computing the successors of a state. Any program that exposes this interface is thereby compatible with any (explicit or symbolic) model checking solution that supports it (for instance LTSmin or Spot). Then, Go2Pins produces a set of files that are compiled together to build the **go2pins-mc** binary. We opted for this workflow since (1) it provides more flexibility, (2) it can be easily extended and (3) our code remains in the Go realm (useful for black-box transitions, see Section 4).

At fine grained level, our approach behaves like a transpiler[2] that translates the input Go program into an output Go program that respects the PINS interface. This transformation has many advantages. First, it benefits from both the reflexivity and the standard library of the Go language. The reflexivity lets us avoid the development of the classic toolchain of a transpiler (scanner, parser, AST, etc.), while the use of the standard library lets us avoid redeveloping concepts such as Control Flow Graph, Call Graph, etc. The second benefit of our approach is the ease of building abstractions (see Section 4).

Figure 2 shows that Go2Pins processes the input program in steps. Each one modifies the Abstract Syntax Tree (AST) in order to desugar a specific feature. For instance, the *Arith&Assign* step decomposes complex arithmetic operations into consecutive elementary ones. For instance `v1 := 3 * g(n) * h(n)` is translated into three instructions: `v1 := 3`, then `v1 *= g(n)` and finally `v1 *= h(n)`. Thus, this step does not change the semantics of the original program but simplifies it in order to be used by model-checkers.

With this workflow, it is easy to test each step. For almost all steps presented in Figure 2, we can just apply the step on some input, run the modified program and check that the behavior stay unchanged.

Among the various steps in Go2Pins, some are of special interest:

1. **TypeChecker**. Ensures, via type deduction, that the current limitations of Go2Pins are respected. Currently Go2Pins is limited to unbuffered channels, Integer variables and a static number of goroutines (i.e. dynamic goroutine creation is not yet supported). Note that these kind of restrictions are common to most verification tools. Section 4 details how these restrictions can be by-passed.
2. **Core (*Func. Def.* to *Transform*)**. This is the core of Go2Pins: it translates the program into a structure that can easily be adapted to match the PINS interface (more details in Section 3.1).
3. **Recursion.** Since Go2Pins only works with finite state spaces (with possibly infinite behaviours), a specific attention must be paid to recursion. This step unrolls each function up to a limit fixed by the user. Since the depth of recursion is fixed, only bounded verification can be done on recursive programs.
4. **DuplicateGoroutines.** This step adds the support for goroutines, i.e. multi-threaded programs. This is achieved by the implementation of a scheduler that returns all the possible interleavings from a given state. More details can be found in Section 3.3 and Section 9.

[2] In compiler realm, a difference is made between *compilers* that usually produce a directly usable artifact whereas transpilers produce another form of source code.

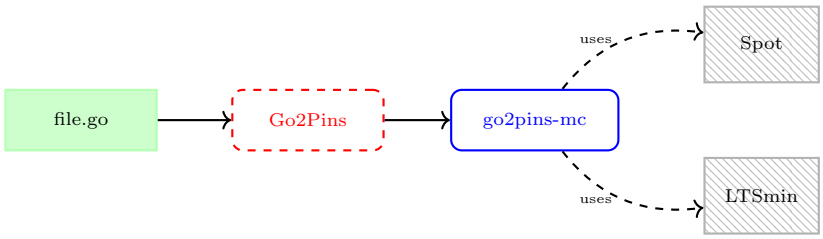


Fig. 1 Overview of Go2Pins. The input file is processed by Go2Pins which produces a binary called **go2pins-mc**. This binary can then be used to verify LTL formula using one of the two supported backends: Spot or LTSmin.

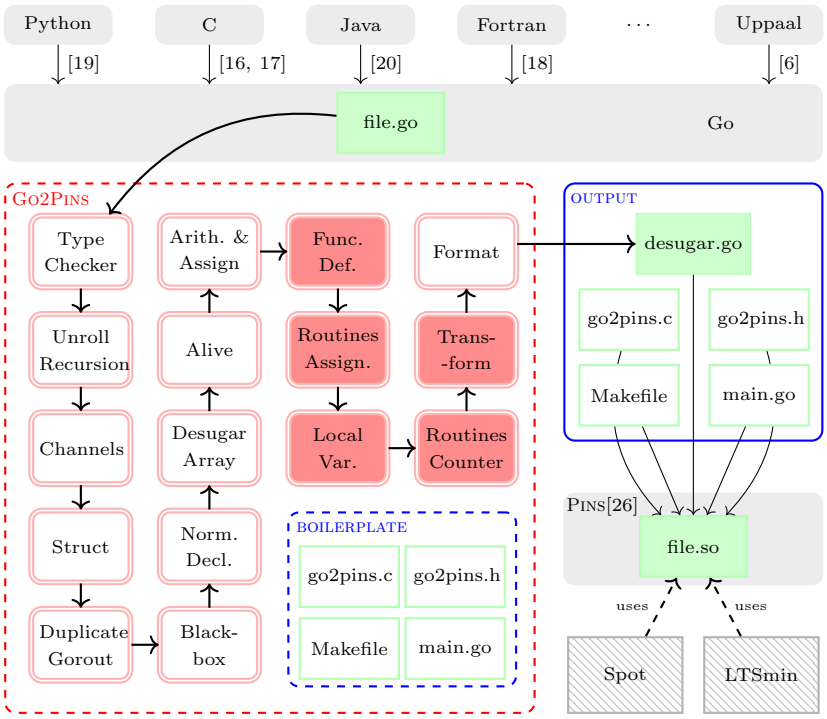


Fig. 2 Contributions of this paper (all except gray boxes). The dashed boxes represent the Go2Pins tool while the blue plain box represents the output directory produced by Go2Pins. The transformation steps are denoted by double shaped red boxes. Files grouped under the name *boilerplate* are copied as-is into the output directory. These files are generic and handle communication between the desugared program and the mandatory functions to respect the PINS interface.

5. **Black-Box.** This module reduces the state space explosion problem by fusing consecutive transitions into a single one (more details Section 4).

Fortuitous behaviour of our approach. During the conception of our tool, we were advised that a lot of transpilers targeting Go exist. Some of these tools were developed by the Go Team in order to translate some parts of the Go compiler (originally written in C) into Go. Thus, our workflow transitively supports model-checking these mainstream languages (details in Figure 2 and Section 7).

3 Implementation Details

3.1 Core translation: *Func. Def.* to *Transform*

The core of Go2Pins (steps *Func. Def.* to *Transform* of Figure 2) translates the input program into a structure that can be easily adapted to match the PINS interface. This interface exposes two functions: one for retrieving the initial state of the system (represented by a vector of N integer variables), and one for computing the successors of a state[3]. The illustration of this transformation is given in Listing 1 for an original program and Listing 2 and 3 for the transformed program.

[3] Model checkers represent the model as a Kripke structure. These two functions are enough to provide a Kripke view of a Go program.

```
1  func fibo(n int) int {
2    n0 := 0
3    n1 := 1
4    for i := 0; i < n; i++ {
5      n2 := n0 + n1
6      n0 = n1
7      n1 = n2
8    }
9    return n1
10 }
11
12 func main() {
13     fibo(5)
14 }
```

Listing 1 Fibonacci computation in Go

```
1  type state [15]int
2
3  func G2PF_fibo(s state) state{
4    switch s.LabelCounter {
5      case 0: goto label0
6      //...
7      case 12: goto label12
8    }
9    label0: // n0 := 0
10     s.fibo.n0 = 0
11     s.LabelCounter = 1
12     s.fibo.isalive = 1
13     return s
14   //...
15   label12: // return n1
16   s.fibo.res0 = s.fibo.n1
17   s.fibo.FunctionCounter =
18       s.fibo.caller
19   s.fibo.LabelCounter =
20       s.fibo.callerLabel
21   return s
22 }
```

Listing 2 Fibonacci translation (1/2)

```
23 func G2PF_main(s state) state {
24   switch s.LabelCounter {
25     case 0: goto label0
26     //...
27     case 2: goto label2
28   }
29   label0:
30     s.fibo.n = 5
31     s.fibo.caller =
32        s.FunctionCounter
33     s.fibo.callerLabel = 2
34     s.FunctionCounter = 1
35     s.LabelCounter = 0
36     return s
37   //...
38   label2:
39   //...
40 }
```

Listing 3 Fibonacci translation (2/2)

```
41 func G2PEntry(src state) []state {
42   r := make([]state, 0)
43   r := append(res, G2PF_main(src))
44   // From here it's the scheduler
45   // detailed Section 3.2
46   // Build all valid successors
47   for _, g := range goroutines {
48     r = append(r, g.Fun(src))
49   }
50   // See Listing 1.6
51   return r
52 }
```

Listing 4 Dispatch in Go2Pins

```
53 func get_successors(src state,
54        cb CB /*Callback*/) int {
55
56   // Compute all successors
57   dsts := G2PEntry(src)
58
59   // Call the model checker
60   // callback for each succ
61   for _, dst := range dsts {
62     CB(cb, dst)
63   }
64 }
```

Listing 5 Successor computation

The first step of this translation is to build a (finite) state vector for the program given in Listing 1. To build this vector, we must compute the total number of variables that are used. Here, four variables n, n0, n1 and i are displayed but Go2Pins requires extra-variables:

1. The *program counter* indicating the line currently executed. This information is hidden in Listing 1 since it is generally handled directly by the microprocessor. For the sake of clarity we opted for a two variables representation of this counter: a variable *FunctionCounter* that indicates the current function, and a variable *LabelCounter* that indicates the current instruction.
2. Another piece of information that is usually tracked at the assembly level is the *return address*, i.e., the position where the execution should continue after a **return** statement (or the end of the function). As previously two variables per function are used: $\langle fun\text{-}name \rangle.caller$ that indicates the return function and $\langle fun\text{-}name \rangle.callerLabel$ that specifies the instruction in this function.
3. When a function returns one or multiple values, a placeholder for these values should be available. Indeed, since these values may be used in various contexts (assignments, comparisons, etc.), the placeholder will represent them until their final use is detected. As a consequence, Go2Pins uses X placeholder variables $\langle fun\text{-}name \rangle.resX$, where X denotes the X^{th} return value.
4. Finally, each variable in the original program is associated to an extra variable $isalive\_\langle var\text{-}name \rangle$. This is required in order to handle complex initialization

such as $a := f()$. In this assignment the value of a is only known after the evaluation of $f()$. Since the PINS interface represents the program as a vector of integers, a default value must be fixed for all variables (here 0). As a consequence, a model-checking procedure may fail by considering this default value. Thus, the extra variable indicates whether or not the variable a has already been initialized. Due to lack of space, this transformation is not depicted here but would appear in line 14.

To respect the PINS interface, the previous variables are collapsed into a vector of integers (line 1, Listing 2). Since this vector handles all values of all variables at a given time, it can be see as a snapshot of the system. Listings 2 and 3 also detail the other modifications performed during the **core translation** (for the sake of clarity names are explicit, while our translation manipulates indexes: for instance, *s.fibo.res0* is then translated into *s[2]*):

- Each name has been changed to *G2PF_⟨fun-name⟩* and its parameters have been replaced by a single parameter: the state vector representing the actual status of the execution (line 3 and 23).
- Each instruction of the original program has been extracted into a dedicated block of code (see lines 9–12 or 14–20 for an example). This block is accessible from a switch statement at the beginning of the function (lines 4–8 or 24–28). This switch uses the *LabelCounter* to detect the instruction to execute and then jump to the corresponding block.
  This transformation in blocks relies on the computation of *Basic Blocks* and *Control Flow Graph* (CFG). *Basic Blocks* are sequences of instructions without jumps (conditional or not) while the *Control Flow Graph* is a graph that represents all of the execution paths of the function and links each basic block to its potential successors. For the purpose of our tool we restrict basic blocks to contain only one instruction of the original program. As a consequence, the CFG represents the successors of each instruction. With this CFG, each basic block can now be augmented to update *FunctionCounter* and *LabelCounter*. In particular, moving inside a function modifies the *LabelCounter* (line 11) while a call to another function modifies both variables(line 16–19 and 24–35) . For instance, line 9 details the modification of the *LabelCounter* while lines 14 to 17 modifies both counters since they represents the original return statement.

The last step of the translation aggregates all the previous transformations in order to fit the PINS interface. With this architecture, the PINS *get_successors* (Listing 5) delegates the processing to *GP2Entry* (Listings 4) which transitively[4] delegates to the current function *G2PF_⟨fun-name⟩*. This strategy preserves (with a minimal overhead) the structure of the original program which is helpful for debugging or producing traces during the verification procedure.

Discussion. The translation schemes depicted previously are very technical and may be confusing for a reader non familiar with both the Go language and the architecture of Go2Pins. In order to clarify these schemes, Figure 3 describes, at an higher level, some of the aforementioned translations. The goal of Go2Pins is to provide a Kripke view of a program written in Go. Consequently, each translation scheme builds a part of a Kripke structure. By applying repeatedly these schemes, from the *main* entry point we are able to build the whole Kripke structure.

Let us first focus on the first translation which represents the assignment of a complex expression α in the variable a. In this scheme, the orange states represent the statement(s) that precedes the current translation while gray diamonds represent statements/expressions that must be recursively translated. In order to translate $a = \alpha$, α must be translated first. Then the value computed by α must be assigned in a. This is achieved by using the $\Theta(var\_name)$ notation. Since a state is represented as an integer array, this notation returns the index of the *var_name* variable in this array. Then, the variable $a.is\_alive$ is set to true (the constant 1), indicating that the variable a is now initialized. The last state (white state), only represents the next statement to be translated[5].

The second translation scheme details the translation of a conditional statement. Similarly to the previous scheme, *cond*, α, and β must be recursively translated. Even if Go2Pins produces both the α and the β branches, the resulting Kripke structure will only have one of the two branches, depending on the value of *cond*.

The scheme for the *for* statement is straightforward: it only splits the various parts of the statement and combines all together. Similarly to the previous schemes, this translation will build a sub-Kripke structure that is "linear", i.e. one instruction only produces a sequence of states. The only translations that are not linear are translation schemes for concurrency primitives (more details section 3.3).

[4] This is achieved by building one last extra function: *G2PMain* (see line 42). This function takes a state vector as a parameter and returns an initialized state vector during the first call. Then, this function dispatches the processing of the computation to the function under execution.

[5] In practice, this state represents the increment of the program counter.

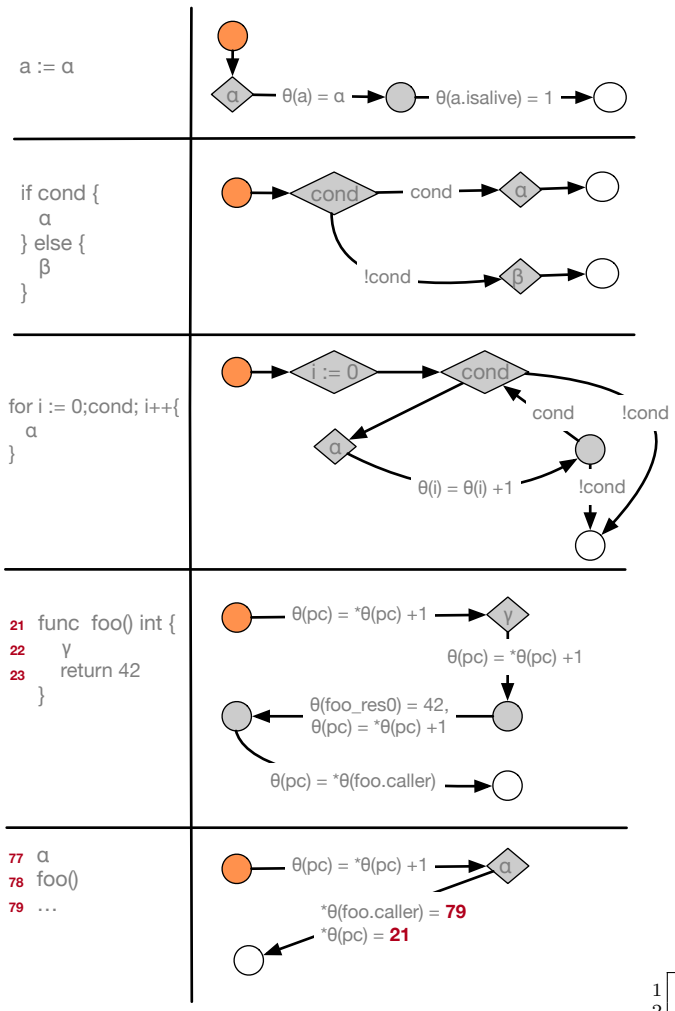


Fig. 3 Description of some translation schemes. Orange circle denotes the current state, white circle the instruction after the one in translation, grey diamond represents not yet translated patterns while grey circles represent intermediate states.

The two last schemes are dedicated to the translation of functions and sequences of instructions. In order to have a better visualisation, we also annotate lines by their identifiers (red numbers) and describe here how the *program counter* (*pc*) is managed (while the previous translations hide this management [6]). One can observe that, every edge increments the program counter ($*\Theta(pc)$ means the value stored at *pc*). One can also note that the final instruction retrieved the value of *foo.caller*. A closer look to the last schemes, reveals that this value is fixed to the instruction to be executed after the call to the *foo* function.

[6] Here we merge *LabelCounter* and *FunctionCounter* for the sake of clarity.

3.2 Support for Stuctures and Interfaces

At first glance, structures (*struct*) in Go look very similar to C structures: they provide an easy mechanism to pack collection of typed fields (variables) together. Nonetheless, structs are also, and mainly, used for object-oriented programming.

In Go, the object-oriented paradigm is very different from the one of C++ or Java. Indeed, Go architects preferred (1) composition over inheritance and (2) interfaces over subclasses. Since object-oriented paradigm is mostly the implementation of the **is-a** concept it can be implemented in many different ways. The C++ or Java implementation requires to define a full hierarchy while the Go implementation prefer compositions/interfaces that are more flexible. An interface is a contract of implicit behaviors. Using interfaces has the advantage of making code more economical, more readable, provides good APIs between packages and reduces repetition. The design choices of Go, commonly referred as Duck-Typing [7], can be viewed as a usage-based structural equivalence between a given object and the requirements of a type. In other words an object is of a given type if it has all methods and properties required by that type. Listing 6 presents an example of duck-typing that displays alternatively "Hello there!" and "Bark bark!".

```
1  type Speaker interface {
2          Say()
3  }
4
5  type Human struct {
6          age int
7  }
8
9  type Dog struct {
10 }
11
12 func (h Human) Say() {
13         fmt.Println("Hello there!", h.age)
14 }
15
16 func (h Dog) Say() {
17         fmt.Println("Bark bark!")
18 }
19
20 func main() {
21         var s Speaker
22         for i := 0; i < 20; i++ {
23                 if i%2 == 0 {
24                         s = Dog{}
25                 } else {
26                         s = Human{i}
27                 }
28                 s.Say()
29         }
30 }
```

Listing 6 Duck Typing in action.

[7] or structural typing

In order to handle such program, Go2Pins must catch two things: (1) structs definitions (lines 5 to 10) and (2) interfaces (lines 1-3, 12-18, and 28).

1. **Supporting structs definitions.** Go2Pins performs a first pass that collects information about structures, e.g. the various fields, their types, their size, etc. With this information, we are then able able to calculate the *exact* size of our struct, by simply counting each the size (in integers) of each field [8]. During this process, we keep compute one unique identifier for each structure.
   Then a second pass translates structures as arrays. Each access to one fields of the structure is then rewritten as an array subscript. For instance, *s.age* is translated as *s[1]*. Note that the *age* field is not at index 0 but 1 since index 0 contains metadata (the unique identifier).
2. **Supporting interfaces.** With the previous translation, Go2Pins must now handle interfaces. Concretely this means that each virtual dipatch [9] must be replaced by something that simulates the genericity of interfaces.
   This is achieved by implementing a dispatcher that exploits the meta-information computed during the previous step. Listing 7 details such a dipatcher lines 9 to 15.

```
1  var (  // Structs
2      Human_struct int = 1
3      Dog_struct int = 2
4  )
5
6  //...
7
8  func Speaker_Dispatcher_Say(s [2]int) {
9      if s[0] == Human_struct {
10         Human_Say(s)
11     } else if s[0] == Dog_struct {
12         Dog_Say(s)
13     } else {
14         panic("Unreachable point: undefined struct")
15     }
16 }
17
18 // ...
```

Listing 7 Meta-information and dispatcher for structs and interfaces

In practice, structures with different size can match the same interface. In this case, Go2Pins only consider the biggest one for the implementation of the dispatcher. In other words, smaller structures will be enlarged and zero-filled in order to match the biggest structure used for a given interface.

[8] Note that Go2Pins handles the same way, nested structures.

[9] Virtual dispatches are calls to function *Say()*

3.3 Handling Concurrency: Goroutines and Unbuffered Channels

The previous sections present the core translation for sequential programs. Nonetheless the main application of model checking is the verification of concurrent programs where bugs are hard to find and reproduce. The concurrency in Go is provided through two elements: *goroutines* and *channels*. Goroutines are triggered by the **go** instruction and spawn lightweight threads. Channels are communication features that, contrarily to shared variables, avoid data races. In order to support goroutines, Go2Pins implements a scheduler. Indeed, at any moment, both the main thread as well as any active goroutine can progress. An *active goroutine* is a goroutine that (1) has been spawned by the **go** keyword and, (2) that is not yet finished. Consequently, this status is stored in the state vector (so that the scheduler can arrange the various goroutines). Additionally, since each goroutine needs its own recursive stack, a preprocessing phase is required to reserve slots for each function that could be called by each goroutine. This processing is similar to the one done for unrolling recursive functions.

Support for channels also requires to have dedicated slots in the state vector. These slots catch goroutines that are about to perform a synchronization operation through the channel. As soon as our scheduler detects two of these goroutines, a synchronization is triggered. In other words the scheduler ensures a simultaneous progress of the two goroutines. Listing 8 details this part of the scheduler (and finalize the code of Listing 4, line 48). It can be observed that the set of successor is only composed of a set of PINS vectors.

```
final := []
// walk all successors  and keep only valid ones
for _, s := range  r {
  if ∃ one channel with (at least) a pending
       read and a pending write {
    tmp := generate all read/write synchronizations
           on this channel
    final = append(final, tmp)
  } else if s has no pending operations
           on channels {
    final = append(final, s)
  }
}
r = final
```

Listing 8 Scheduler that synchronizes operations on channels

4 Abstraction with Black-Box Transitions

4.1 Overview of black-box transitions

The main problem that arises when verifying large (concurrent) software systems is the state-space explosion problem since all of the details must be represented to catch all possible behaviors. One way to tackle this problem is to use approximations that remove some irrelevant details in order to reduce the size of the state space. Two kind of approximations exist:

- **over-approximations** contain more behaviors than the full system. Thus, if there is no error in an over-approximation, then there is no error in the full system. On the other hand if an error is found in an over-approximation it can be spurious. Over-approximations cannot prove presence of errors.
- **under-approximations** contain less behaviors than the full system. Thus if there is an error in an under-approximation, then this error is real error in the full system. On the other hand, absence of errors in an under-approximation does not imply absence of errors in the full systems. Under-approximations cannot prove absence of errors.

```
1  package main
2
3  import "fmt"
4  import "math"
5
6  func foo(n int) int {
7    return n * 2
8  }
9
10 func main() {
11   a := int(math.Sqrt(42))
12   a = a + foo(a)
13   fmt.Println(a)
14 }
```

Listing 9 Simple computations

In this paper, we introduce the **black-box transitions** technique in order to overcome limitations of both over and under-approximations. The underlying idea is to *automatically* build a representation of the program that abstracts away all behaviors irrelevant for the verification procedure while keeping effectiveness for proving correctness of properties or finding errors. This technique can be seen as a *structural reductions* and takes its roots in the work of Lipton [32] and Berthelot [2]. Nowadays, these reductions are still considered as an attractive way to alleviate the state explosion problem [28, 3]. Structural reductions strive to fuse structurally "adjacent" events into a single atomic step, leading to less interleaving of independent events and less observable behaviors in the resulting system. While structural reductions traditionnaly work only on "user-code", the black-box transition technique proposed here goes further by (1) avoiding complex computation for the abstraction and (2) allowing to fuse events/functions in imported code.

In order to illustrate the black-box transition technique, let us consider the example depicted in Listing 9. This example only performs arithmetical operations: it first calls *math.Sqrt* (line 11) which is part of the Go standard library and then calls *foo* (line 12) which is a local function. The result is then printed (line 13). Suppose now that we want to check the (correct) LTL property *FG 'a > 1'*, which express that a will end to be strictly greater than 1.

Trying to verify this property over this program is hard due to lines 11 and 13. Indeed since both of these lines are calls to functions that belong to the Go standard library, the source code of these functions is not available[10]. Consequently the translation depicted in Section 3.1 will not work. More generally this problem occurs with any Go program that links with an external library. This problem is annoying since this is a common situation in any large system.

Fortunately, when checking *FG 'a > 1'*, we are only interested in (1) the value of the variable a and (2) the value returned by the *math.Sqrt* function. All the details of the *math.Sqrt* functions are irrelevant for the verification procedure.

The black-box transitions technique exploits this particularity by directly calling *math.Sqrt*. The returned value is then set in the slot corresponding to a in the PINS vector. More generally, the black-box transitions technique automatically identifies external function calls, and directly insert the result of these calls during the core translation described Section 3.1 [11]. To achieved this some manipulation of the PINS vector are required to fill the parameters of the function.

The example of Listing 9 depicts black-box functions with integer parameters. Nonetheless Go2Pins supports passing variables as arguments. In this latter case, Go2Pins simply copy the correct slots of the PINS vector into the formal arguments of the function. Since Go2Pins does not (yet) support references, there is no consistency problems.

Thus, black-box helps to reduce significantly the state-space of the program. For instance, the state-space of the program in Listing 9 has only 12 states which is low considering that the definition of both *math.Sqrt*

[10] The runtime of programming language is traditionally provided as a dynamic library.

[11] Notice that this technique is only possible since Go2Pins is developed in Go and produces Go files.

and *fmt.Println* function are complex and are several hundred lines of code[12].

Discussion. Black-boxes address the state space explosion problem by fusing multiple transitions (here, external library function calls) into a single one. Thus, black-boxes assume the correctness of these external functions calls. The verification of these functions is then delegated to the writer of the external library who can opt to use testing or model-checking. Consequently, the developer can only focus on verifying its own code and on providing a high quality software. This strategy follows the idea of Godefroid [22] who states that some part of the software can be checked by model-checking while some part can be checked by testing. This strategy is interesting since it can progressively be integrated into all existing project in order to increase the quality of the project.

Remark on Go2Pins limitations. Like many model-checking approaches (ATL+ for instance that distinguish programs and instances) Go2Pins is currently limited to Integer variables, and compositions (structures) of Integer variables. Nonetheless black-box transitions can check arbitrary complex code (for instance *math.Sqrt* or *fmt.Println*. Consequently, Go2Pins restrictions only apply to user code and not to imported code.

Blackbox and LTL verification. One drawback of abstraction methods (such as Partial Order Reductions) is the compatibility with the LTL Next operator. Since blackbox transitions collapse successive transitions into one based only on the observed atomic propositions, the use of the Next operator is possible without altering the verification results. In other word this technique only removes the noise from the verification procedure.

A word on side effects. Black-box transitions are not limited to pure functions and also work with functions containing side effects. For instance, call to *fmt.Println* is fully supported. The only drawback of our method is that we will observe the result of calling *fmt.Println* during the verification procedure.

4.2 User-defined black-box transitions

It is legitimate to ask whether the black-box transition technique could also be applied to user code. A closer look to Listing 9 shows that the *foo* function could also be black-boxed if we are only interested in the value of the variable *a*.

Go2Pins can automatically detect such functions. The computation of functions that can be black-boxed is more complex than we realize at first glance. A function can only be black-boxed if it respects the following rules:

1. None of its variables is referred to during the verification process
2. It only calls functions that can be black-boxed
3. It does not manipulate global variables

A more precise definition could be stated but would require to compute all the possible executions paths. Since this may be costly we opted for this conservative approximation which is enough in most cases, and can be easily computed.

Once all black-boxed functions are detected, Go2Pins removes them from the original input and puts them into a dedicated package. By achieving this, Go2Pins is back to the situation described in the previous section. Thus, user defined functions can now be black-boxed. With this approach the state space of the program in Listing 9 can be reduced from 12 states to 9 states (25% reduction).

Hence, with this approach, an automatic abstraction, restricted to only behavior mandatory for the verification, is built.

Supporting depth-1 functions using global variables. There are some situations where the aforementioned rule (3) is too restrictive (more details in Section 7). Consider for example a simple function f that modifies a global variable v. Let us now suppose that we want f to be black-boxed. A simple rewriting system can be used to catch this situation. The function f is moved in the blackbox package and rewritten to accept one more argument: a reference to the actual PINS vector. Then every access to global variables is modified to reference the correct slot in the PINS vector. This technique works well but has a severe limitation[13]: we cannot have a black-box function g that will call f. In other words, g will never be considered as black-box. This is too restrictive since some part of g could be nonetheless abstracted away: e.g. all lines that are not required for the computation of f. Future work aims to investigate whether a solution to this problem exist.

5 PSL$_{\text{REC}}$: Sugaring PSL for Recursive Programs

[12] The interested reader may look the definition of:
`https://golang.org/src/fmt/print.go`
`https://golang.org/src/math/sqrt.go`

[13] Another restriction concern the use of the LTL Next operator. Indeed, if the blackboxed function has multiple modification of one variable, only the later one will be visible.

Go2Pins handles recursion by unrolling k times the recursive functions (with k fixed by the user). This unrolling produces, for each step of the recursion, new functions, with new names. Thus, it can be seen as an explicit representation of the recursive stack. Such unrolling with depth 2 is detailled in Listing 10 and 11. A closer look to function $facto\_2$ reveals that we opted to raise an exception when the maximum depth is reached. To achieve that, we keep track of the level of recursion and replace all recursive calls by calls to **panic**. The only problem with this translation is from the user point of view: the variable *facto.n* no longer exists since it has been replaced by multiple $facto_i.n$ variables, with i the level of recursion. Thus, during the verification process, the user can only check properties involving $facto_i.n$ and not *facto.n*.

To ease this manipulation, we propose to sugar PSL, an industrial Property Specification Language [11] and a superset of LTL syntax. We opted for PSL since Go2Pins targetsindustrial settings[14]. We sugar PSL with two new operators **any** and **all**. We refer this new version of PSL: PSL_{REC}. It aims to translate automatically atomic propositions expressed on the (recursive) source code into their substituted names. For instance, the LTL property F **any**(*"facto.n == 0"*) would be translated into F (*"*$facto_1$*.n == 0" || "*$facto_2$*.n == 0"*). The PSL property {**all**(*"facto.a == 1"*)`[*]`}`[]`->F **any**(*"facto.n == 0"*) would be translated into {(*"*$facto_1$*.a == 1" && "*$facto_2$*.a == 1"*)`[*]`}`[]`->F(*"*$facto_1$*.n == 0" || "*$facto_2$*.n == 0"*) With this translation, the semantics of the original formulae is preserved (i.e., for the PSL formula, F **any**(*"facto.n == 0"*) must be true every time the regular expression **all**(*"facto.a == 1"*)* is matched.).

Let k be the maximum recursion depth. Let $a \in AP$ be an atomic proposition. For $i \in [1..k]$, let us denote by a_i the occurrence of a in the i^{th} step of the recursion. Let σ be an infinite sequence and $\sigma(0)$ be the first element of this sequence. Let us introduce two new operators such that: $\sigma \models$ **any**(a) iff $\exists i \in [1..k], a_i \in \sigma(0)$ and $\sigma \models$ **all**(a) iff $\forall i \in [1..k], a_i \in \sigma(0)$.

Remark. We strongly believe that the PSL_{REC} formalism can also be used outside Go2Pins. Consequently, we implement a tool called *pslrec* that takes as an input: the PSL formula and a JSON describing the recursion depth for each variables. This tool is distributed as a part of Go2Pins but we keep it external in order to make it reusable in another context.

[14] Note that only the Spot backend supports PSL

```
func facto(n int) int {
  if n == 0 {
    return 1
  } else {
    return n * facto(n-1)
  }
}
```

Listing 10 Factorial function

```
func facto_1(n int) int {
  if n == 0 {
    return 1
  } else {
    return n * facto_2(n-1)
  }
}
func facto_2(n int) int {
  if n == 0 {
    return 1
  } else {
    panic("Max Depth Reached")
  }
}
```

Listing 11 Two level unrolling of the *facto* function of Listing 10

6 Using Go2Pins on Go programs

This section provides the necessary commands to run and play with Go2Pins[15]. To download Go2Pins you can either fetch it and compile it from the git repository using:

```
git clone https://gitlab.lre.epita.fr/spot/go2pins.git
    && make
```

or you can use the package manager of Go using the following command. In this case, the tool will be installed directly in your $GOBIN directory.

```
go get gitlab.lre.epita.fr/spot/go2pins
```

Notice that Go2Pins has two dependencies you have to install by your own: LTSmin[16] and Spot[17]. Once this has been done, you can run Go2Pins on the example of Listing 9 using `go2pins -f listing.1.7.go`
The previous command produced an *out* directory containing the **go2pins-mc** binary. This binary can then be used for model-checking the original program.

- `./out/go2pins-mc -list-variables` lists all variables you can use for LTL model-checking. One can observe that each variable is prefixed by the package name and the function name.
- `./out/go2pins-mc -kripke-size` computes the state space of the program. You should obtain 12 states visited as aforementioned.

[15] Under GPL (v3), available at `https://gitlab.lre.epita.fr/spot/go2pins`
[16] `https://ltsmin.utwente.nl`
[17] `https://gitlab.lre.epita.fr/spot/spot`

- `./out/go2pins-mc -ltl 'FG "main_main_a > 1"' -backend spot -nb-threads 1` runs the command of Section 4 with one thread using the Spot backend. You should observe an extra display 18, that corresponds to black-boxing *fmt.Println* .

Finally, if you want to blackbox the *foo* function, you have to regenerate the *out* directory and rerun the verification process. Go2Pins offers a shortcut to perform both actions simultaneously

```
go2pins -f -blackbox-fn="auto"
        listing.1.7.go 'FG "main_main_a > 1"'
```

Discussion. Go2Pins progressively desugar the original program into a program that respects the PINS interface. With this workflow it could be sometimes hard to follow the various transformations and their side effects in the original program. Consequently, we added an option **-debug** that prints out the result of each translation into a dedicated file (located into the `out/debug` directory). This option ease the verification and the debug of each pass.

Notice that our primary goal was to build a verification tool that compares the results of the original program with the results of the transformed program after each step of the translation [18]. This approach works well but triggers "declared but not used" errors when trying to compile the transformed programs. Indeed, our translations introduce extra-variables that are deservedly considered by the Go compiler, as "unused variables". Since the Go language specification forbids this, there is no flag for compiling without this error..Consequently, if you want to check each transformation, you have to compile your own Go compiler and get rid of this message by commenting lines 67-69 of [13].

7 Benchmark

7.1 Evaluation of Go2Pins

In order to test[19] Go2Pins we opted to translate industrial-inspired problems coming from the RERS challenge [38]. These reactive systems are represented through huge files written in C. To test the whole workflow of our approach, we first use C4Go [17] to translate them into Go, then apply the Go2Pins workflow.

[18] This approach is valid at least until we reach the core-translation

[19] Details of our benchmark and how to reproduce it are available at `https://www.lre.epita.fr/~renault/benchs/SPIN-2021/results.html`

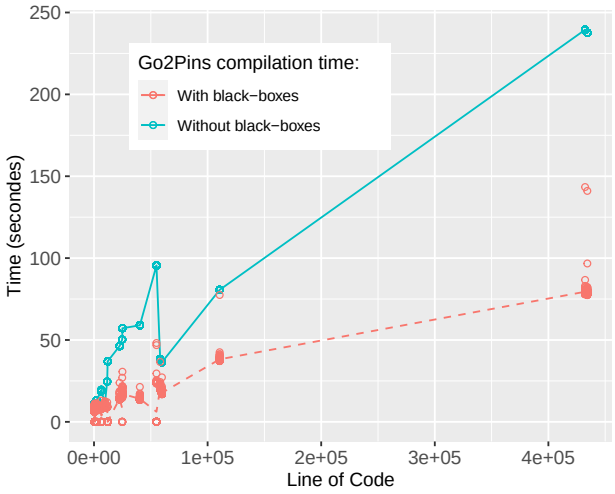


Fig. 4 Time required by Go2Pins to process and compile an input go program according to its number of line of code. Dots represent one computation in the benchmark (a pair model-formula), while lines join the mean of each series.

The RERS challenge comes with a set of LTL formulae. Consequently, our benchmark is composed of 41 models (1 909 345 LOC) and 5 064 formulae. Among these 5 064 formulae 35% are verified and 65% are violated. Regarding the hierarchy of Manna and Pnueli [34], our benchmark is split in 25% pure guarantee, 44% pure safety, 2% pure obligation, 12% pure persistence, 12% pure recurrence, and 5% pure reactivity. Finally all experiments were run with a 4 minutes timeout and 200 GB memory limitation on a 24 cores Intel(R) Xeon(R) CPUX7460@ 2.66GHz with 256GB of RAM.

In order to handle this benchmark, Go2Pins must be able to simulate an environment. Indeed, the RERS challenge simulates reactive systems where the environment (a non controllable component) impacts the results of the verification process. In order to model this non controllable component, RERS opted to read arbitrary finite sequence of bytes (characters) from the standart input (over a specified alphabet). To ensure the validity of the system over all traces, we have to check each individual inputs. One option would be to provide a trace generator that generates all finite sequence from a given alphabet. This solution is not satisfactory since it would require user action to plug the two tools together. A closer look to the structure of RERS file show that we can avoid any user actions. Indeed, he alphabet is specified directly in the file and the environment is simulated by a specific statement (that read standard input). Therefore we adapted Go2Pins to track this specific statement: every time this statement is triggered not only one but many successors are produced, one per potential value of the environment (i.e. the alphabet).

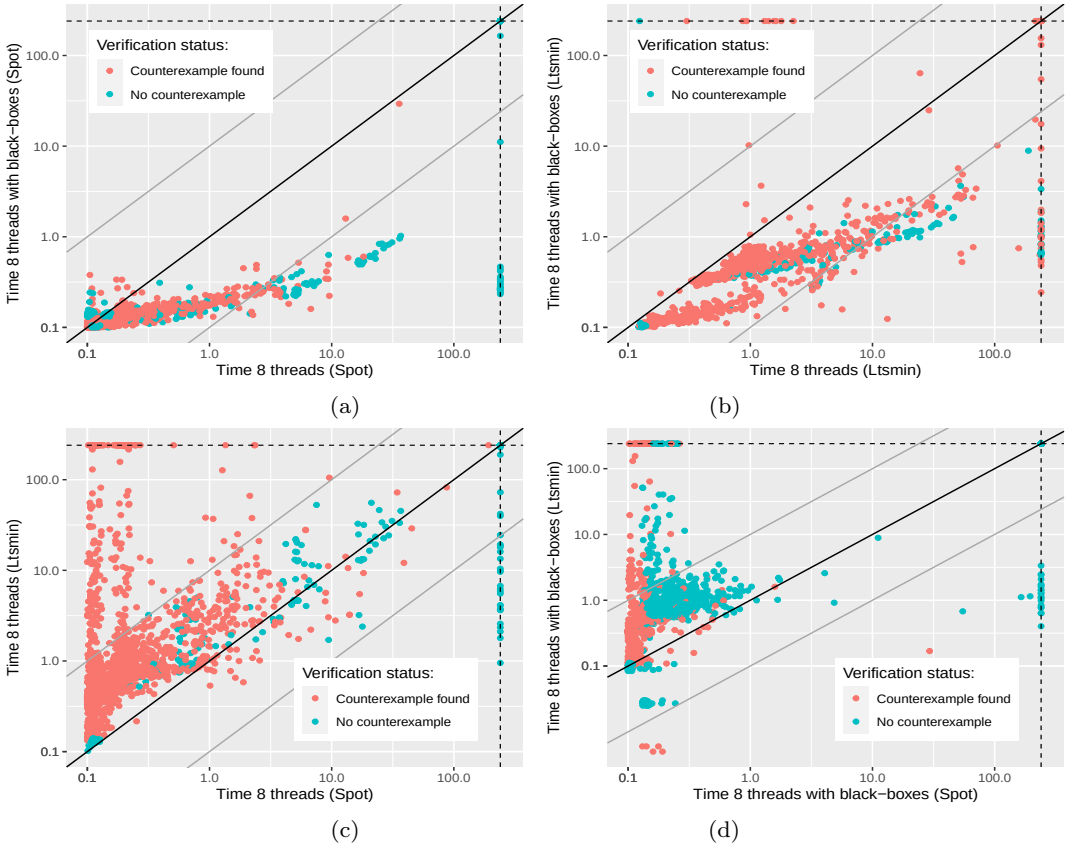


Fig. 5 Time comparison in $\log_{10}$ scale for each backend (Spot and LTSmin), with or without black-boxes. The dark line corresponds to identity while gray lines show the 10 factor speedup/slowdown. Dashed lines represent the 4 minutes timeout.

Figure 4 focuses first on the scalability of Go2Pins. This figure details the time required by Go2Pins to translate and compile the files of the benchmark. For each pair model-formula a dot is displayed while lines join the mean of each series [20]. Two approaches are depicted: with or without the use of the black-boxes technique. Surprisingly, we can first observe that the use of black-boxes also reduces the processing time. Since our approach decomposes each statement in atomic operations, the use of black-box will produce smaller files that are easily processed by the go compiler. Thus, with the black-box technique, our tool process around 5000 line per second. A closer look to these results reveals that Go2Pins uses 60% of this time while the Go compiler uses 40% of it. Consequently, there is room for improvement in our tool. Finally one can observe huge variation for some models. These models have a low number of lines of code, but each line has complex operation: Go2Pins spends time to reduce these operations to atomic operations.

[20] In our benchmarks multiples programs have the same number of line of code (LOC). A serie is defined as all computations, i.e. one per formula, w.r.t. a specific LOC.

Figure 5 displays the time required to process the whole benchmark by both Spot and LTSmin. In (a) and (b) it can be observed that the use of black-boxes significantly improves both Spot and LTSmin. Figure 5 (c) and (d) display the comparison between Spot and LTSmin on this benchmark. Without black-boxes, Spot outperform to find counterexamples while LTSmin seems better to find empty products (the hardest ones). This difference could come either from the type of Büchi automaton used (which differ between Spot and LTSmin default configurations) or from the default emptiness check algorithm used [5, 12]. Further investigation could broaden the study of [4]. Finally, Figure 5 (d) shows that the use of black-boxes helps Spot to resolve empty products.

Figure 6 (a) and (b) display the number of states and the number of transitions with or without black-boxes when using Spot. Figure 6 (c) and (d) depict the same information for LTSmin. In explicit model checking these metrics are important: the runtime is proportional to the number of transitions explored while the memory consumption is proportional to the number of states. For both Spot and LTSmin, the number

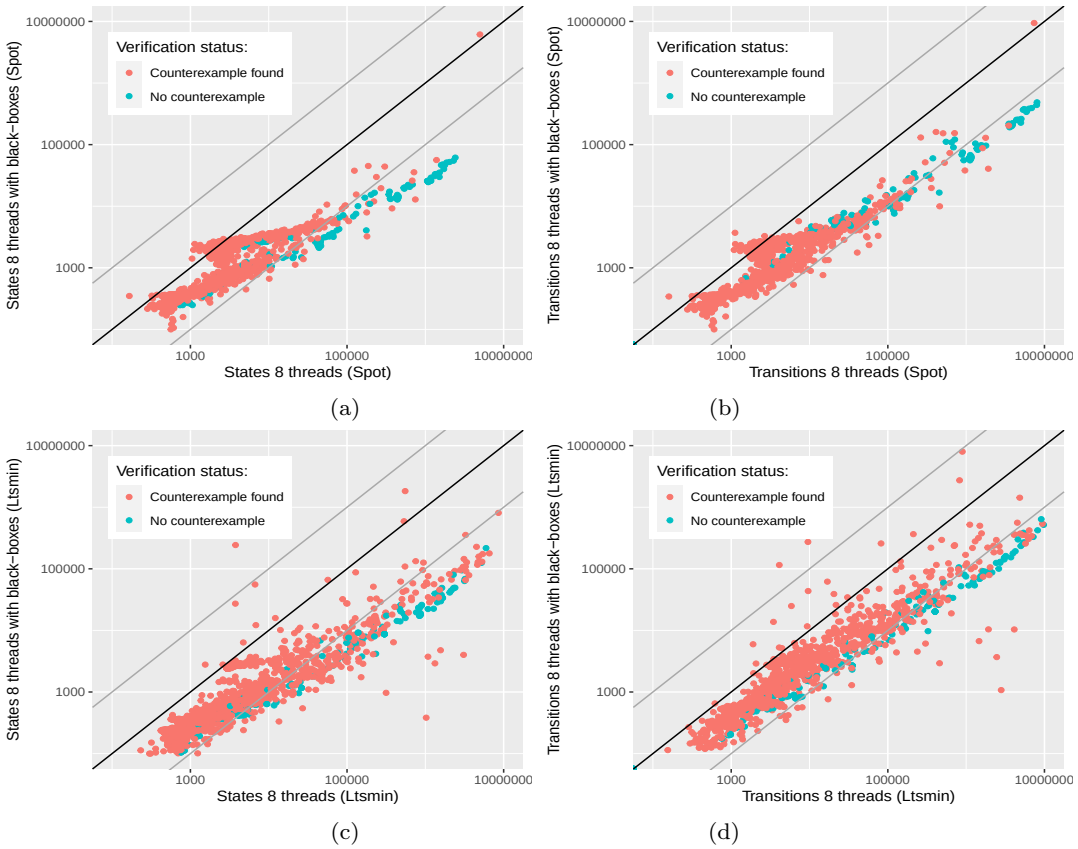


Fig. 6 States and transitions (with and without black-boxes) comparison for both Spot and LTSmin. The dark line corresponds to identity while gray lines show the 10 factor speedup/slowdown.

of states and transitions is divided by 10 to 100 when using black-boxes.

In conclusion, the black-box technique helps to reduce both preprocessing and verification runtime.

Validation. We also opted to test our approach using the RERS benchmark in order to ensure correctness of our implementation. Indeed this benchmark fully specifies 10 models through exactly 964 LTL formulae. These pairs (models, formulae) describe all lines that are (or not) reachable in the input file. In addition to the tests developed during the conception of our tool, these specific models confirm the validity of our workflow. One should note that most of these files are unprocessable within the 4 minutes timeout restriction. For black-box transitions, we compare all obtained results to the 5 064 original results. Also note that we plan to translate the BEEM database, used by Spin and DiVinE2.4 in order to increase the confidence in our tool[21].

7.2 Evaluation of PSL_{REC}

Since PSL_{REC} could be used in multiple contexts, this section aims at evaluating the impact of PSL_{REC} desugaring on the translation time of LTL[22] formulas to Büchi automata. In order to test it over an extensive benchmark, we opted to use LTL formulas extracted from the RERS benchmark, and modified them by sugaring arbitrary atomic proposition with the **all** operator. Any atomic proposition sugared with the newly introduced **any** or **all** operators will be replaced with k equivalent atomic propositions to support verification of a recursion of depth k. This can greatly increase the complexity of the translation of the formula to a Büchi automaton. Figure 7 shows a comparison of the translation time between Spot and LTL2BA (used by default for the LTSmin backend). Spot shows overall better performance, especially when increasing the depth of the recursion (from 5 to 30). Additionally, we measured the size of the automata produced by both translations. Spot seems to consistently produce au-

[21] We also plan to translate the Promela database `http://www.albertolluch.com/research/promelamodels` to Go in order to compare with other verification tools

[22] Although PSL_{REC} itself supports PSL syntax, only pure LTL formulas were used for this evaluation since LTSMin's default translation backend only supports LTL syntax.

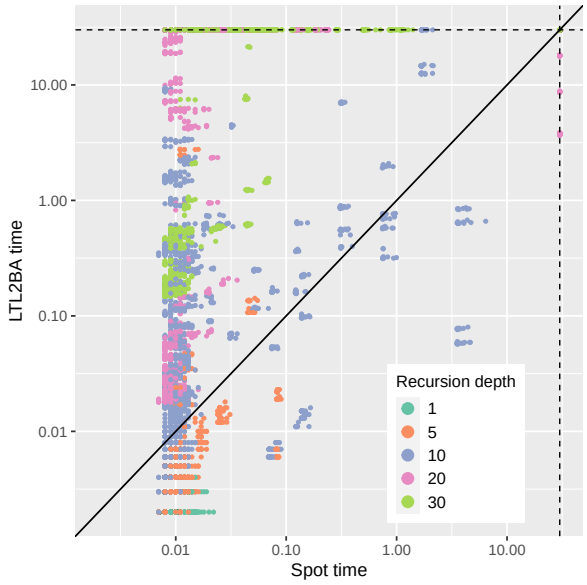


Fig. 7 Translation time comparison in seconds for Spot and LTL2BA over a set of formulas extracted from the RERS challenge and modified by PSL_{REC}. Dashed lines represent the 30 seconds timeout.

tomata with fewer states. Note that these experiments translates LTL formulae into Büchi automata, but Spot could produce smaller automata (transition-based generalized Büchi automta). When using Go2Pins, we therefore advise either to use the Spot backend or to use Spot as a translation tool for the LTSmin backend (when verifying programs that make use of recursion).

8 Related Work

The development of Go2Pins has been motivated by several empirical studies performed on the Go language [37, 7, 39]. Ray et al. [37] study the relation between types of bugs and multiple programming languages. Dilley and Lange [7] analyzed 865 Go projects in order to detect how channels are used in large Go projects. Tu et al. [39] study 171 real-world concurrency bugs in Go.

To our knowledge, the LTL-verification of full and unmodified Go programs has never been studied. Many studies [33, 31, 30, 35, 8] focus on a static analysis of operations on channels. Liu et al. [33] developed a tool that detect statically patterns of bugs and fix them according to some strategies. The other approaches [31, 30, 35, 8] focus on extracting channels operations. This extraction is then used to to build models that are then verified for correctness. Between SPIN'21 (the first publication related to Go2Pins) and this paper, several studies focused on the verification of Go program. Dilley and Lange [9] proposed an elegant approach (and an associated tool called Gomela) that extract operations on channel to build a promela model. This model is then checked using the SPIN model checker. Focusing on the channels operations is interesting since it build an abstraction but has two main problems: false positives/negatives can be raised, and not all LTL properties can be expressed on the programs since some variables are abstracted away. Another, not yet published, work has been made by Zhong et al. [44]. Their tool to identify concurrency bugs in Go applications via dynamic binary analysis. In other words, their Bingo tool activates tracepoints on the binary to be able to detect bugs. Nonetheless, the source code is not yet available and therefore, we were not able to see if their technique could be compatible with some LTL verification. All the aforementioned studies mainly focuses on concurrency problem by checking data-races, communication patterns or deadlocks. Focusing only on channels operation helps to build small models that are processable by verification tools. In this paper we developed a broader approach since (1) we are able to check all LTL properties, (2) we are not restricted to channels operations and (3) we developed a the black-box technique that helps to fight combinatorial explosion without restricting ourself to only channels communications.

Another approach [6] aims to execute formal models by converting Uppaal programs into Go. Similarly Giunti [21] proposed to map pi-calculus specifications of static channels into Go executable programs. Our workflow avoids such transformations, since programs can be executed and verified as-is.

Handling the standard library is a real problem for software verification tools. JPF [41] requires providing the source code of the standard library and relies on a Virtual Machine. The idea of black-box transitions, that naturally handle the standard library, has never been proposed to our knowledge. The closest idea is the one of Spin [25] that is able to execute multiple instructions atomically (see **atomics**, **d_steps** and **c_code** keywords). Since this approach is not automatic and relies on a model written in Promela, it is not well suited for verifying large software systems. One should note that approaches based on the LLVM bytecode also exist. The first one [43] links with Spin for handling concurrency while the second one [1] requires a program expressed in C++. In contrast to our approach, no model can be extracted.

9 Toward the Verification of Real-World Programs

In this section, we test Go2Pins over some buggy programs adapted from real world applications. These programs come from the GoBench benchmark [42] that

Listing 12 Negative counter bug adapted from [14]

```
1  package main
2  import "sync"
3
4  var wg sync.WaitGroup
5  var listSize = 5
6  var someList []int = []int{-2, -5, -3, -1, 0}
7
8  func process() {
9    // code omitted working with someList
10   wg.Done()
11 }
12
13 func main() {
14   // Fixup error by removing line 19 and
15   // uncommenting the following line
16   // wg.Add(listSize)
17   for i := 0; i < listSize; i++ {
18     go process()
19     wg.Add(1) // Error
20   }
21   wg.Wait()
22 }
```

list a variety of concurrency issues detected either in popular open source applications or in kernel programs. We also compare Go2Pins to Gomela on these programs.

9.1 Kubernetes Negative Counter

Listing 12 details a typical safety bug that was found in kubernetes [14]. This program uses *WaitGroups* that are popular shared memory structures dedicated for synchronizing multiple goroutines. *WaitGroups* can be used through three primitives: *Add(n)* that declares (or augments) the number of goroutines to wait, *Done* that specifies that some goroutine has finished its work, and *Wait* that is blocking until the number of gouroutines to wait has not been reached. Note that *Waitgroups* trigger a runtime error if the counter they handled becomes negative.

In this example, the main thread spawns several goroutines (line 18) and invokes *wg.Add(1)* just after. Then, each goroutine performs computations on the list before invoking *wg.Done()* (line 10). The problem in this snippet of code comes from the line 19. In an execution where the first worker goroutine finishes its job quickly, it may decrement *wg* before it is incremented, and thus trigger a run-time error.

In order to verify this program Go2Pins must be adapted to support WaitGroups. Thanks to the modular architecture of Go2Pins, we provide this support by writing a single transformation step of only 150 (documented) lines [23] . This demonstrates how simple the extension of Go2Pins can be. The transformation step is straightforward: all WaitGroup variables are converted into an integer variable, every *Add(n)* invocations just increases the variable by n, every *Done* decreases the variable by 1, and every *Wait* operation is translated in a (while-based) spin loop.

Then, checking that a given WaitGroup *wg* will not trigger an error can be done by checking the LTL formula 'G "wg $\geq$ 0"'. Checking this formula against Listing 12, raises a counterexample in less than 40 ms (for a 3 seconds compilation time). Therefore this Kubernetes bug can be detected by Go2Pins.

By comparison checking this program [24] with the state-of-the-art Gomela tool is less straightforward. Indeed Gomela computes different values (by default 3) for the *listSize* in order to build various abstractions. Then each abstraction is verified, each one taking approximatively 1.8 seconds. In order to be more confident in the results, the user may specify the range of values for *listSize*. All abstractions will report the bug, except one, when *listSize* equals to zero. In this case, no goroutine is spawned, and therefore no bug is found. Even if this information is correct, it will never occurs in the program to check. Thus the user may be confused by this extra-information. Also note that applying the fix of line 14-16 for Listing 12 raises false positives with Gomela while Go2Pins outputs directly the correct answer (by exploring a state-space of 682 state and 2746 transitions in 20 ms).

9.2 Trillian Blocking Bug

Listing 13 presents an adapted version of a bug found in the Trillian [15] project, a verifiable data store developed at Google.

At lines 23 to 25, this program starts by allocating one token per available CPU. Each token is then stored in the *limitCh* buffered channel. By opposition to unbuffered channels described Section 3.3, buffered channels are also blocking channels but contains a fixed size buffer that can be filled before becoming blocking. For instance, such a buffer of size 2 will authorize two insertions without removal before becoming blocking for a new insertion.

Once all tokens have been inserted in *limitCh*, the program spawns goroutines, according to the size of the verifiable data store. Therefore, each goroutine executes

[23] This transformation can be found at `https://gitlab.lre.epita.fr/spot/go2pins/-/blob/master/transform/waitgroup.go`

[24] An adaptation of this program w.r.t the constraints of Gomela

Listing 13 Blocking bug adapted from [15]

```
1  package main
2
3  var cpus int = 2
4  var store_size int = 3
5  var n int = 4
6
7  var limitCh chan int =  make(chan int, cpus)
8  var ch chan int = make(chan int, n)
9
10 func wait_and_close() int {
11   exit_code := 1
12   // code omitted, close channel, etc.
13   return exit_code // SUCCESS
14 }
15
16 func process() {
17   a := <-limitCh // get a token
18   s := 42 // complex computation ommitted here
19   ch<-s
20   limitCh <- a // return token
21 }
22
23 func main() {
24   for i := 0; i < cpus; i++ {
25     limitCh <- 1
26   }
27
28   for i := 0; i < store_size; i++ {
29     go process()
30   }
31
32   for i := 0; i < n; i++ {
33     tmp := <- ch
34     // code omitted
35   }
36
37   result := wait_and_close()
38 }
```

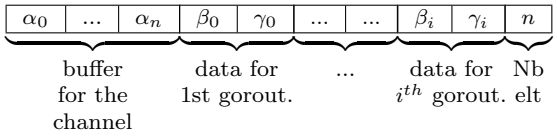
the *process* function that (1) retrieves a token, (2) performs some computation, (3) send the results of this computation into the *ch* buffered channel, and (4) release the token.

Finally, the main thread (lines 31 to 34) aggregates the values computed by the various goroutines and finalizes the program (line 36).

This program has a subtle bug that originates in lines 4 and 5. Indeed, this program works perfectly when *store_size* and n have the same values or when *store_size* is greater than n[25]. The blocking bug comes when n is greater than *store_size*. In this particular case lines 32 to 35 become blocking as soon as *store_size* elements have been been retrieved from *ch*. Indeed, since *ch* will contain at most *store_size* elements (line 19), line 33 becomes blocking after that.

[25] This is handled by some omitted code line 34

In order to verify this program Go2Pins must be adapted to support both *spawning loops* and *buffered channels*. Spawning loops, i.e. loops whose purpose is only to launch goroutines (see lines 28 to 30) can be easily supported. Indeed, fixed size loops can be unrolled by duplicating the content of the loop. This simple transformation can be done with only 200 lines of code. The support of buffered channels is more complex. While unbuffered channels act as a *rendez-vous* between two goroutines, buffered channels must handle all the possible interleaving (w.r.t. the size of the channel) before becoming blocking. Thus, the representation of the channel requires more metadata. The structure of this representation is depicted below



where α_0 to α_n represents the buffer associated to the channel, (β_j, γ_j) represents metadata of goroutine j (information about the operation, and about the value), and n represents the number of elements in the buffer. With this information, the scheduler of Listing 8 can be adapted for generating all possibles permutations. For instance if one goroutine wants to write 42, while the second wants to write 51 and the third one 69, then our scheduler will produce 6 possibles orders.

With these two modifications, Go2Pins is now ready to verify the behavior of the program depicted in Listing 13. Checking that no goroutine is blocked forever can be done by checking the LTL property "'GF "result == 51"', i.e. the main thread always finishes. Checking this formula raises a counterexample in immediately; therefore this Trillian bug can be detected by Go2Pins. Similarly to the Kubernetes bug, Gomela is also able to detect this deadlock but needs to compute different values for *cpus*, *store_size*, and n. Each of these verifications takes 1.6 seconds, leading to a total verification of almost 1 minute (to compare with the 50 ms needed by spot in addition to the 3.5 compilation time).

Technical issue. In the original snippet of code, the variable *cpus* (line 3) is initialized to *runtime.NumCPU()*. Thanks to the blackboxes, Go2Pins could work with this definition. Nonetheless, this solution is not satisfactory since the results of the verification procedure may depend on the underlying architecture. For instance on a two-cores architecture, verifying that FG "cpus == 2"' would return *true* while checking the same property on a four-cores architecture would return *false*.

One potential solution would be to follow a strategy similar to the one of Gomela, i.e. to pick random values for *cpus* an to verify all the properties for each random value. This is not ideal since it may raise false positives.

Another solution could be to follow the idea used to handle the RERS benchmark. In other words, consider *runtime.NumCPU()* as a special case that does not produce a single value but a set of values and therefore produce a set of successors. Even if this idea seems attractive, it will have a significant impact on the size of the state space, and does not solve the problem: if *runtime.NumCPU()* returns a set with all values up to 16, an architecture of 18 cores may lead to similar problems. Finding an adequate solution is left for a future work.

Discussion. Through these two industrial examples, it seems that Gomela and Go2Pins are complementary. On one hand, the abstractions provided by Gomela seems to be more adapted to find potential bugs in large code bases, but could raise false positives. On the other hand, Go2Pins provides exact results but needs to be adapted to handle new concepts. Our feeling is that checking correctness of Go programs could be done in two steps: (1) use Gomela to detect potential bug, (2) then use Go2Pins to rule out false positive. Indeed, since Gomela abstraction are very relaxed, one can efficiently detect false positives. Then, Go2Pins can check the veracity of the previous verdict by achieving an exhaustive verification.

10 Conclusion

This paper introduces Go2Pins, the first tool developed for LTL model-checking over Go programs. It relies on the idea that the Go language is a good candidate for specifying, verifying and building asynchronous systems. Go2Pins uses the PINS interface to link with an ecosystem of model-checkers and model-checking techniques. This paper also introduces black-box transitions to tackle the combinatorial explosion problem. Our benchmark has proven the efficiency of this technique by reducing by more than a factor the size of the state-spaces. Moreover, this technique provides an easy way to support features that are not yet supported by Go2Pins.

Future work aims to support more Go features in order to analyze the structure of the state space of industrial problems (following up the static empirical study of Dilley and Lange [7]). To handle industrial project we would like to support Partial Order Reductions (POR) [40, 36, 29]. Currently only LTSmin supports POR through the use of dependencies matrixes. We plan to compute these matrixes directly into Go2Pins and to integrate POR into Spot. We also would to like to study the relation between black-boxes and POR.

Additionally we would like to go deeper in the development of the black-box technique. For huge functions that cannot be black-boxed we could nonetheless find sequences of instructions that could be fused. Moreover we would like to investigate whether the black-box technique could be generalized to handle any-depth functions with global side-effects.

Finally, our tool only performs verification without fairness since both LTSmin and Spot require fairness to be expressed in the LTL-formula. Nonetheless, expressing fairness directly in Go2Pins could help to reduce state-space size. In order to achieved that, we would like first perform a study of source code of the go scheduler, then we would like to see if we can adapt their code directly inside of Go2Pins in order to avoid adding complexity in the LTL formulae.

References

1. Z. Baranova, J. Barnat, K. Kejstova, T. Kucera, H. Lauko, J. Mrazek, P. Rockai, and V. Still. Model checking of C and C++ with DIVINE 4. In *ATVA'17*, vol. 10482 of *LNCS*, pp. 201–207. Springer, 2017.
2. G. Berthelot. Checking properties of nets using transformation. In *Applications and Theory in Petri Nets*, vol. 222 of *LNCS*, pp. 19–40. Springer, 1985.
3. B. Berthomieu, D. Le Botlan, and S. Dal Zilio. Counting Petri net markings from reduction equations. *International Journal on Software Tools for Technology Transfer*, Apr. 2019.
4. F. Blahoudek, A. Duret-Lutz, V. Rujbr, and J. Strejček. On refinement of Büchi automata for explicit model checking. In *SPIN'15*, vol. 9232 of *LNCS*, pp. 66–83. Springer, Aug. 2015.
5. V. Bloemen and J. van de Pol. Multi-core scc-based ltl model checking. In *HVC'16*, vol. 10028 of *LNCS*, pp. 18–33. Springer, Nov. 2016.
6. J. Dekker, F. Vaandrager, and R. Smetsers. Generating a google go framework from an uppaal model. Master's thesis, Radboud University, August 2014.
7. N. Dilley and J. Lange. An empirical study of messaging passing concurrency in go projects. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER'19)*, pp. 377–387, 2019.
8. N. Dilley and J. Lange. Bounded verification of message-passing concurrency in go using promela and spin. *Electronic Proceedings in Theoretical Computer Science*, 314: 34–45, Apr 2020. URL `http://dx.doi.org/10.4204/EPTCS.314.4`.
9. N. Dilley and J. Lange. Automated verification of go programs via bounded model checking. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 1016–1027, 2021.
10. A. Duret-Lutz, A. Lewkowicz, A. Fauchille, T. Michaud, E. Renault, and L. Xu. Spot 2.0 — a framework for LTL

and ω-automata manipulation. In *ATVA'16*, vol. 9938 of *LNCS*, pp. 122–129. Springer, Oct. 2016.
11. C. Eisner and D. Fisman. *A Practical Introduction to PSL*. Series on Integrated Circuits and Systems. Springer, 2006.
12. S. Evangelista, A. Laarman, L. Petrucci, and J. van de Pol. Improved multi-core nested depth-first search. In *ATVA'12*, vol. 7561 of *LNCS*, pp. 269–283. Springer, 2012.
13. GitHub repository. Go Compiler. https://github.com/golang/go/blob/04fb929a5b7991ed0945d05ab8015c1721958d82/src/go/types/stmt.go#L67-L69, .
14. GitHub repository. Kubernetes generate node map bug. https://github.com/kubernetes/kubernetes/blob/d70ee902fddc682863a3cc4f0d8eac0223ebf70b/test/e2e/storage/vsphere/nodemapper.go#L62, .
15. GitHub repository. Trillian preload bug. https://github.com/kubernetes/kubernetes/blob/d70ee902fddc682863a3cc4f0d8eac0223ebf70b/test/e2e/storage/vsphere/nodemapper.go#L62, .
16. GitHub repository. C2Go: Migrate from C to Go. https://godoc.org/rsc.io/c2go, 2020.
17. GitHub repository. C4Go: Transpiling C code to Go code. https://github.com/Konstantin8105/c4go, 2020.
18. GitHub repository. Transpiling fortran code to golang code. https://github.com/Konstantin8105/f4go, 2020.
19. GitHub repository. Grumpy: Go running Python. https://github.com/google/grumpy, 2020.
20. GitHub repository. Java2Go: Convert Java code to something like Go. https://github.com/dglo/java2go, 2020.
21. M. Giunti. Gopi: Compiling linear and static channels in go. In *Coordination Models and Languages*, pp. 137–152, 2020. Springer.
22. P. Godefroid. Between testing and verification: Dynamic software model checking. In *DSSE'16*, vol. 45, pp. 99–116, april 2016.
23. R. Griesemer, R. Pike, K. Thompson, I. Taylor, R. Cox, J. Kim, and A. Langley. Hey! ho! let's go! https://opensource.googleblog.com/2009/11/hey-ho-lets-go.html, 2009.
24. C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, Inc., USA, 1985.
25. G. J. Holzmann. *The Spin Model Checker: Primer and Reference Manual*. Addison-Wesley, 2003.
26. G. Kant, A. Laarman, J. Meijer, J. van de Pol, S. Blom, and T. van Dijk. Ltsmin: High-performance language-independent model checking. In *TACAS'15*, pp. 692–707, April 2015.
27. A. Kirszenberg, A. Martin, H. Moreau, and E. Renault. Go2Pins: A framework for the LTL verification of Go programs. In *SPIN'21*, vol. 12864 of *LNCS*, pp. 140–156, May 2021. Springer.
28. A. Laarman. Stubborn transaction reduction. In *NFM*, vol. 10811 of *LNCS*, pp. 280–298. Springer, 2018.
29. A. Laarman, E. Pater, J. van de Pol, and H. Hansen. Guard-based partial-order reduction. *International Journal on Software Tools for Technology Transfer*, pp. 1–22, 2014.
30. J. Lange, N. Ng, B. Toninho, and N. Yoshida. Fencing off Go: Liveness and Safety for Channel-based Programming. In *POPL'17*, pp. 748–761. ACM, 2017.
31. J. Lange, N. Ng, B. Toninho, and N. Yoshida. A Static Verification Framework for Message Passing in Go using Behavioural Types. In *CSE'18*, pp. 1137–1148. ACM, 2018.
32. R. J. Lipton. Reduction: A method of proving properties of parallel programs. *Commun. ACM*, 18(12):717–721, 1975.
33. Z. Liu, S. Zhu, B. Qin, H. Chen, and L. Song. Automatically detecting and fixing concurrency bugs in go software systems. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, vol. 11, pp. 2227–2240, 2016.
34. Z. Manna and A. Pnueli. A hierarchy of temporal properties. In *PODC'90*, pp. 377–410, 1990. ACM.
35. N. Ng and N. Yoshida. Static Deadlock Detection for Concurrent Go by Global Session Graph Synthesis. In *CCC'16*, pp. 174–184. ACM, 2016.
36. D. Peled. Combining partial order reductions with on-the-fly model-checking. In *CAV'94*, vol. 818 of *LNCS*, pp. 377–390. Springer, 1994.
37. B. Ray, D. Posnett, V. Filkov, and P. Devanbu. A large scale study of programming languages and code quality in github. In *SIGSOFT'14*, pp. 155–165, 2014.
38. RERS challenge. Rigorous examination of reactive systems (RERS). http://rers-challenge.org/2019/, 2019.
39. T. Tu, X. Liu, L. Song, and Y. Zhang. Understanding real-world concurrency bugs in go. In *ASPLOS'19*, pp. 865–878, 2019.
40. A. Valmari. Stubborn sets for reduced state space generation. In *ICATPN'91*, vol. 618 of *LNCS*, pp. 491–515, 1991. Springer.
41. W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda. Model Checking Programs. In *ASE'03*, vol. 10, pp. 203–232. Springer, 2018.
42. T. Yuan, G. Li, J. Lu, C. Liu, L. Li, and J. Xue. Gobench: A benchmark suite of real-world go concurrency bugs. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pp. 187–199, 2021.
43. A. Zaks and R. Joshi. Verifying Multi-threaded C Programs with SPIN. In *SPIN'08*, pp. 94–107, 2008.
44. C. Zhong, Q. Zhao, and X. Liu. Bingo: Pinpointing concurrency bugs in go via binary analysis, 2022.