Go2Pins: a framework for the LTL verification of Go programs

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

LRDE, EPITA, Kremlin-Bicêtre, France
{akirszenberg, amartin, hmoreau, renault}@lrde.epita.fr

Abstract. We introduce Go2Pins, a tool that takes a program written in Go and links it with two model-checkers: LTSMin [19] and Spot [7]. 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.
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.

1 Introduction & Motivation

The Go programming language was designed at Google in 2009 [16] to improve programming productivity in an era of multicore, networked machines and large codebases. Inspired by the idea of *Communicating Sequential Processes* (CSP) [17], 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. 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 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.

2 Go2Pins: Overview

This section describe 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 [19] or Spot [7].

Figure 2 provides more details about this 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) [19]. 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 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 $v1 *= h(n)$.

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

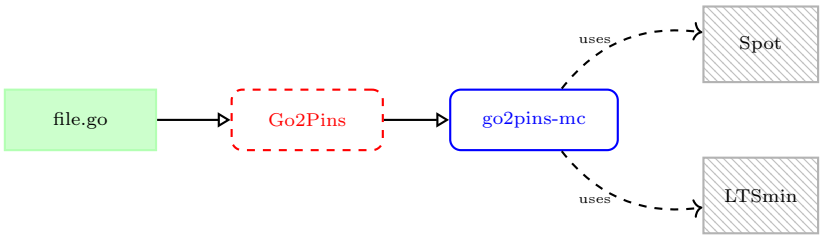

Fig. 1. Overview of Go2Pins. The input file is processed by Go2Pins which produces a binary called **go2pins-mc**. This binary can then be used to verify LTL formula using one of the two supported backends: Spot or LTSmin.

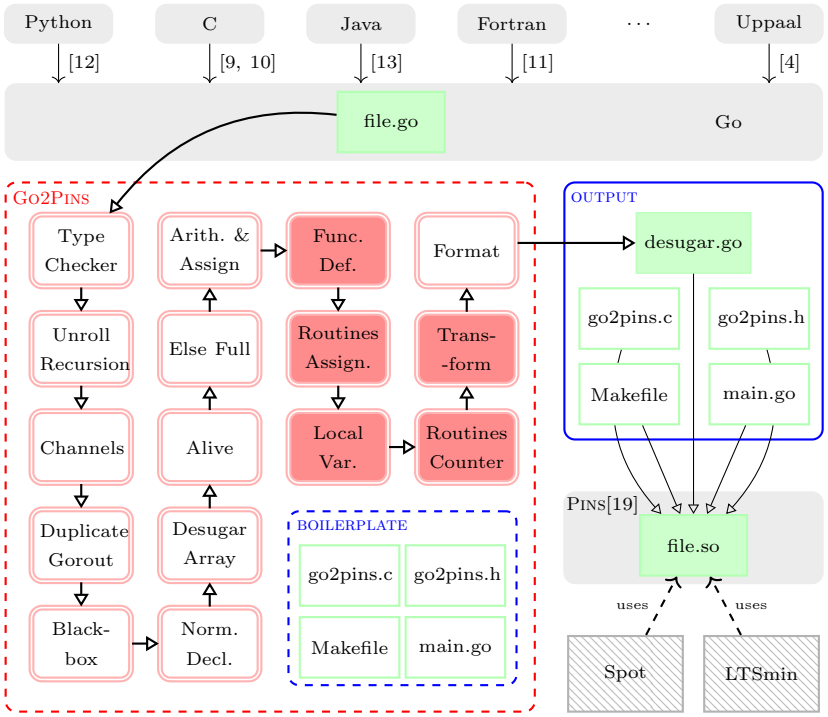

Fig. 2. Contributions of this paper (all except gray boxes). The dashed boxes represent the Go2Pins tool while the blue plain box represents the output directory produced by Go2Pins. The transformation steps are denoted by double shaped red boxes. Files grouped under the name *boilerplate* are copied 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.

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 static number of goroutines (i.e. no dynamic goroutine creation is yet supported). Notice that these kind of restrictions are common to most verification tools. Section 4 details how these restrictions can be bypassed.
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 work only with finite state space (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 3.2.
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 6).

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[2]. The illustration of this transformation is given in Listing 1.1 for an original program and Listing 1.2 and 1.3 for the transformed program.

[2] 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 1.1. 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 1.2. 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 }
```

Listing 1.3. Fibonacci translation (2/2)

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

Listing 1.4. Dispatch in Go2Pins

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

Listing 1.5. Successor computation

The first step of this translation is to build a (finite) state vector for the program given in Listing 1.1. 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 1.1 since it is generally handled directly by the micro-processor. 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: ⟨*fun-name*⟩*.caller* that indicates the return function and ⟨*fun-name*⟩*.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 ⟨*fun-name*⟩*.resX*, where X denotes the X^{th} return value.
4. Finally, each variable in the original program must be associated to an extra variable *isalive_*⟨*var-name*⟩. 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 1.2). Since this vector handles all values of all variables at a given time, it can be see as a snapshot of the system. Listings 1.2 and 1.3 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 1.5) delegates the processing to *GP2Entry* (Listings 1.4) which transitively[3] 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.

3.2 Handling Concurrency: Goroutines and Unbuffered Channels

The previous section presents 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 a communication features that avoid data races contrarily to shared variables.

In order to support goroutines, Go2Pins implements a scheduler. Indeed, at any moment, the main thread can progress as well as any active goroutine. 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 1.6 details this part of the scheduler (and finalize the code of Listing 1.4, line 48). It can be observed that the set of successor is only composed of a set of PINS vectors.

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

[3] 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.

```
final := []
for _, s := range  r { // walk all successors  and keep only valid ones
  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 1.6. Scheduler that synchronize operations on channels

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.
  detection 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.
  correctness of properties.

```
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 1.7. 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.

In order to illustrate the black-box transition technique, let us consider the example depicted in Listing 1.7. 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[4]. 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 a large software.

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.

Black-box transitions technique exploits this particularity by calling directly *math.Sqrt*. The returned value is then set in the slot corresponding to *a* in the PINS vector. More generally, black-box transitions technique automatically identifies external function calls, and directly insert these calls during the core translation described Section 3.1 [5]. To achieved this some manipulation of the PINS vector are required to fill the parameters of the function.

Thus black-box helps to reduce significantly the state-space of the program. For instance, the state-space of the program in Listing 1.7 has only 12 states which is low considering that the definition of both *math.Sqrt* and *fmt.Println* function are complex and are several hundred lines of code long[6].

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 [15] 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. Currently Go2Pins is limited to Integer variables. Nonetheless black-boxes transitions can check arbitrary complex code (for instance *math.Sqrt* or *fmt.Prinln*. Consequently, Go2Pins restrictions only applies to user code and not 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.

---

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

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

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

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 1.7 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 can think at first glance. A function can only be black-boxed if it respect the following rules:

1. None of its variable is referred 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 remove them from the original input and put 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 1.7 can be reduced from 12 states to 9 states (25% reduction).

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

Supporting depth-1 function using global variables. There are some situations where the aforementioned rule (3) is too restrictive (more details in Section 6). 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[7]: 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 and future work aims to investigate whether a solution to this problem exist.

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

5 Using Go2Pins on Go programs

This section provides the necessary commands to run and play with Go2Pins[8]. To download Go2Pins you can either fetch it and compile it from the git repository using:

`git clone https://gitlab.lrde.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.lrde.epita.fr/spot/go2pins`

Notice that Go2Pins have two dependencies you have to install by your own: LTSmin[9] and Spot[10]. Once this have been done, you can run Go2Pins on the example of Listing 1.7 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.
- `./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"'`

6 Benchmark

In order to test[11] Go2Pins we opted to translate industrial-inspired problems coming from the RERS challenge [28]. These reactive systems are represented through huge files written in C. To test the whole workflow of our approach, we first use C4Go [10] to translate them into Go, then apply the Go2Pins workflow.

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 [24], our benchmark is splitted in 25% pure guarantee,

[8] Under GPL (v3), available at `https://gitlab.lrde.epita.fr/spot/go2pins`
[9] `https://ltsmin.utwente.nl`
[10] `https://gitlab.lrde.epita.fr/spot/spot`
[11] Details of our benchmark and how to reproduce it are available at `https://www.lrde.epita.fr/~renault/benchs/SPIN-2021/results.html`

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 time-out and 200 Go memory limitation on a 24 cores Intel(R) Xeon(R) CPUX7460@ 2.66GHz with 256GB of RAM.

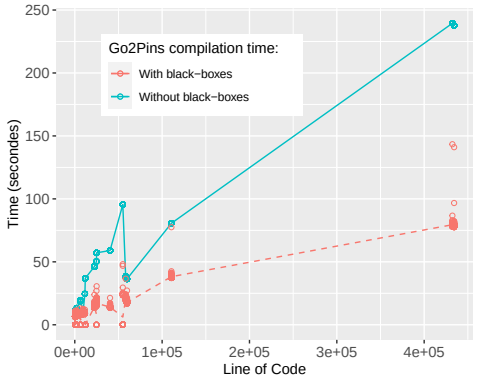

Fig. 3. Time required by Go2Pins to process and compile an input go program according to its number of line of code. Dots represent one computation in the benchmark (a pair model-formula), while lines join the mean of each series.

Figure 3 focuses 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 [12]. Two approaches are depicted: with or without the use of the black-box technique. Surprisingly, we can first observe that the use of black-boxes also reduce 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 reveal that Go2Pins uses 60% of this time while the Go compiler uses 40% of it. Consequently, there is a room for improvement in our tool. Finally one can observe huge variation for some models. These models have low number of line of code, but each line has complex operation: Go2Pins spends time to reduce these operations to atomic operations.

Figure 4 display 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 4 (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). These 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 [3, 8]. Further investigation could broaden the study of [2]. Finally, Figure 4 (d) show that the use of black-boxes help Spot to resolve empty products.

Figure 5 (a) and (b) displays the number of states and the number of transitions with or without black-boxes when using Spot. Figure 5 (c) and (d) depicts the same information for LTSmin. In explicit model checking these metrics are

[12] 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.

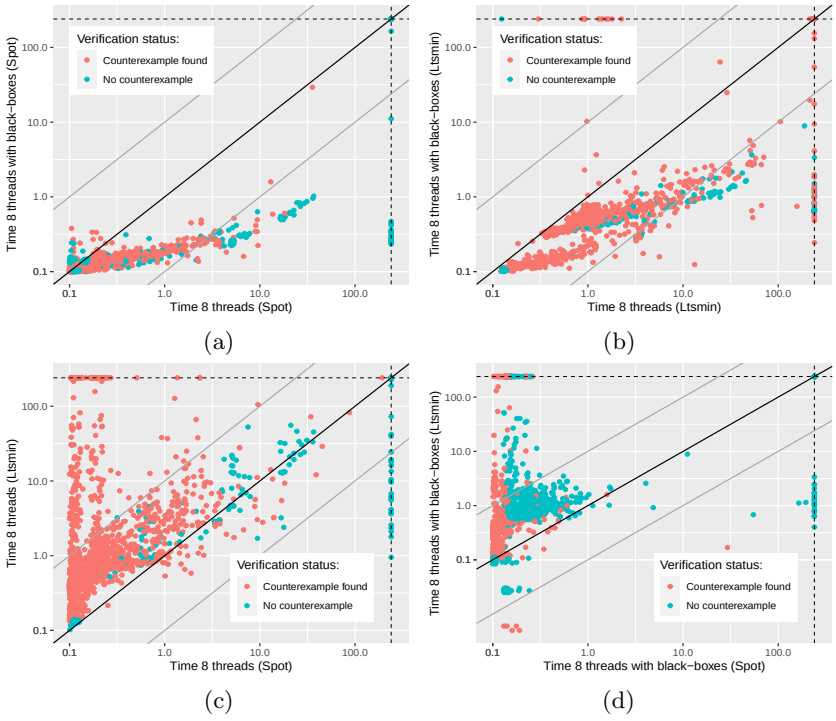

Fig. 4. Time comparison in $\log_{10}$ scale for each backend (Spot and LTSmin), with or without black-boxes. The dark line corresponds to identity while gray lines show the 10 factor speedup/slowdown. Dashed lines represent the 4 minutes timeout.

important: the runtime 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 of states and transitions is divided by 10 to 100.

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

Correctness. 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 work-flow. One should note that most of this 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[13].

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

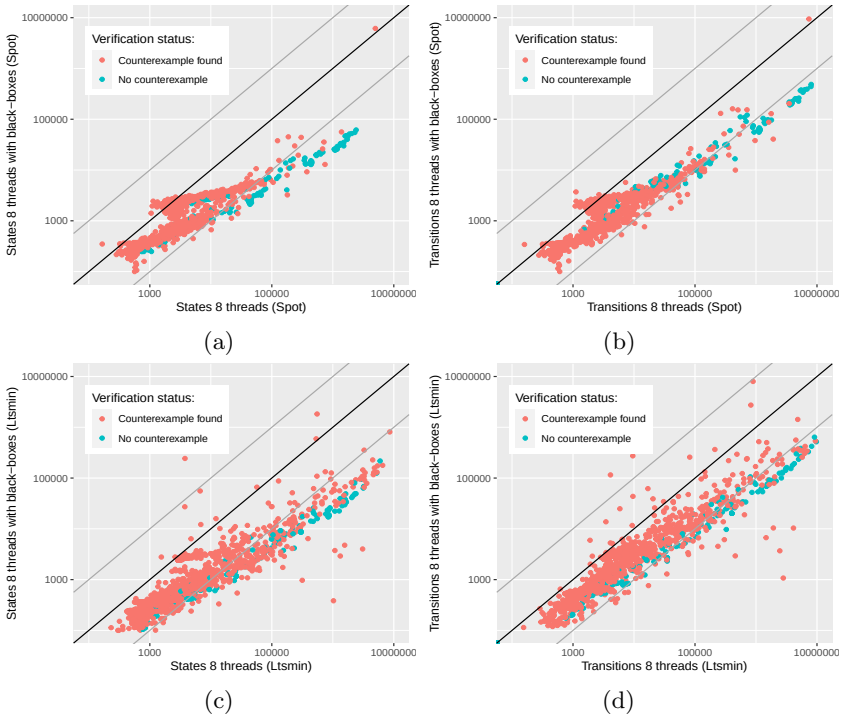

Fig. 5. States and transitions (with and without black-boxes) comparison for both Spot and LTSmin. The dark line corresponds to identity while gray lines show the 10 factor speedup/slowdown.

7 Related Work

The development of Go2Pins has been motivated by several empirical studies performed on the Go language [27, 5, 29]. Ray et al. [27] study the relation between types of bugs and multiple programming languages. Dilley and Lange [5] analyzed 865 Go projects in order to detect how channels are used in large Go projects. Tu et al. [29] 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 [23, 22, 21, 25, 6] focus on a static analysis of operations on channels. Liu et al. [23] developed a tool that detect statically patterns of bugs and fix them according to some strategies. The other approaches [22, 21, 25, 6] focus on extracting channels operations. This extraction is then used to to build models that are then verified for correctness. These 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 [4] aims to execute formal models by converting Uppaal programs into Go. Similarly Giunti [14] 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 [31] 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 [18] 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 [32] 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.

8 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 [5]). To handle industrial project we would like to support Partial Order Reductions (POR) [30, 26, 20]. 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.

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. 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.
3. 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.
4. J. Dekker, F. Vaandrager, and R. Smetsers. Generating a google go framework from an uppaal model. Master's thesis, Radboud University, August 2014.
5. 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.
6. 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`.
7. 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.
8. 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.
9. GitHub repository. C2Go: Migrate from C to Go. `https://godoc.org/rsc.io/c2go`, 2020.
10. GitHub repository. C4Go: Transpiling C code to Go code. `https://github.com/Konstantin8105/c4go`, 2020.
11. GitHub repository. Transpiling fortran code to golang code. `https://github.com/Konstantin8105/f4go`, 2020.
12. GitHub repository. Grumpy: Go running Python. `https://github.com/google/grumpy`, 2020.
13. GitHub repository. Java2Go: Convert Java code to something like Go. `https://github.com/dglo/java2go`, 2020.
14. M. Giunti. Gopi: Compiling linear and static channels in go. In *Coordination Models and Languages*, pp. 137–152, 2020. Springer.
15. P. Godefroid. Between testing and verification: Dynamic software model checking. In *DSSE'16*, vol. 45, pp. 99–116, april 2016.
16. 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.
17. C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, Inc., USA, 1985.
18. G. J. Holzmann. *The Spin Model Checker: Primer and Reference Manual*. Addison-Wesley, 2003.
19. 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.
20. 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.
21. 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.

22. 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.
23. 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.
24. Z. Manna and A. Pnueli. A hierarchy of temporal properties. In *PODC'90*, pp. 377–410, 1990. ACM.
25. N. Ng and N. Yoshida. Static Deadlock Detection for Concurrent Go by Global Session Graph Synthesis. In *CCC'16*, pp. 174–184. ACM, 2016.
26. D. Peled. Combining partial order reductions with on-the-fly model-checking. In *CAV'94*, vol. 818 of *LNCS*, pp. 377–390. Springer, 1994.
27. 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.
28. RERS challenge. Rigorous examination of reactive systems (RERS). `http://rers-challenge.org/2019/`, 2019.
29. T. Tu, X. Liu, L. Song, and Y. Zhang. Understanding real-world concurrency bugs in go. In *ASPLOS'19*, pp. 865–878, 2019.
30. A. Valmari. Stubborn sets for reduced state space generation. In *ICATPN'91*, vol. 618 of *LNCS*, pp. 491–515, 1991. Springer.
31. W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda. Model Checking Programs. In *ASE'03*, vol. 10, pp. 203–232. Springer, 2018.
32. A. Zaks and R. Joshi. Verifying Multi-threaded C Programs with SPIN. In *SPIN'08*, pp. 94–107, 2008.