Bull. Nov. Comp. Center, Comp. Science, 47 (2023), 1-9
@ 2023 NCC Publisher

An approach to using Golang programs for the specification and verification of distributed systems

Evgeny Bodin

Abstract. The work is devoted to an attempt to use Golang programs for the specification and verification of distributed systems. The A.P. Ershov Institute of Informatics Systems, SB RAS, has been developing this approach for years. A distributed system described in terms of an SDL specification, which is first translated to a Dynamic-REAL (dREAL) specification, and then to a Promela specification. All these languages (SDL, dREAL, and Promela) are based on the channel concept. Since the Go language (Golang) shares the same feature, we had an idea to use it in some manner. One way is to use Golang to describe distributed systems. However, it is more practical to try to formally verify the already existing Golang applications by translating them into dREAL specifications (as it is done with SDL specifications) or directly into Promela specifications. The paper compares the aforesaid languages and presents a case study of a specification of a simplified ATM network specified in terms of a Golang program.

Keywords: distributed systems verification, distributed systems analysis, translation, SDL, SPIN, Dynamic-REAL, Golang.

Introduction

The universal popularity of distributed systems makes their validating (testing and verification) extremely important. Testing does not allow us to find all defects in a system or a program; formal verification enables us to *prove* that a program is correct (in other words, that it has some explicitly specified properties).

For years, the A.P. Ershov Institute of Informatics Systems, SB RAS, has been developing the following approach: a distributed system is described in terms of an SDL specification, which is translated to a Dynamic-REAL (dREAL) specification, subsequently translated to a Promela specification.

All these languages (SDL [1, 2], dREAL [3, 4], and Promela [5]) share the channel concept. Since the Go language (Golang) also has this feature, we had an idea to use it somehow.

One possible way is to use Golang to specify distributed systems.

Another way is to try to formally verify the already existing Golang applications by translating them into dREAL specifications (as it is done with SDL specifications) or directly into Promela specifications. Nicolas Dilley and Julien Lange applied this approach: they chose a fragment of the Go language, named it MiniGo, and developed a GOMELA tool [6, 7] to translate MiniGo programs into Promela specifications. Later, they evolved it into a tool-chain for an automated verification of Go programs [8, 9].

The paper compares the languages mentioned above and presents a case study of a specification of an ATM network, which is simplified and specified in terms of a

Golang program. Section 1 gives an overview of our SRDSVer3 verification tool. Section 2 presents an example used for illustrating the verification of distributed systems. Section 3 presents a Golang program corresponding to the SDL specification of the previous paper [4].

Though Promela has been used for specifying ATMs [10, 11], these specifications are for less complex cases.

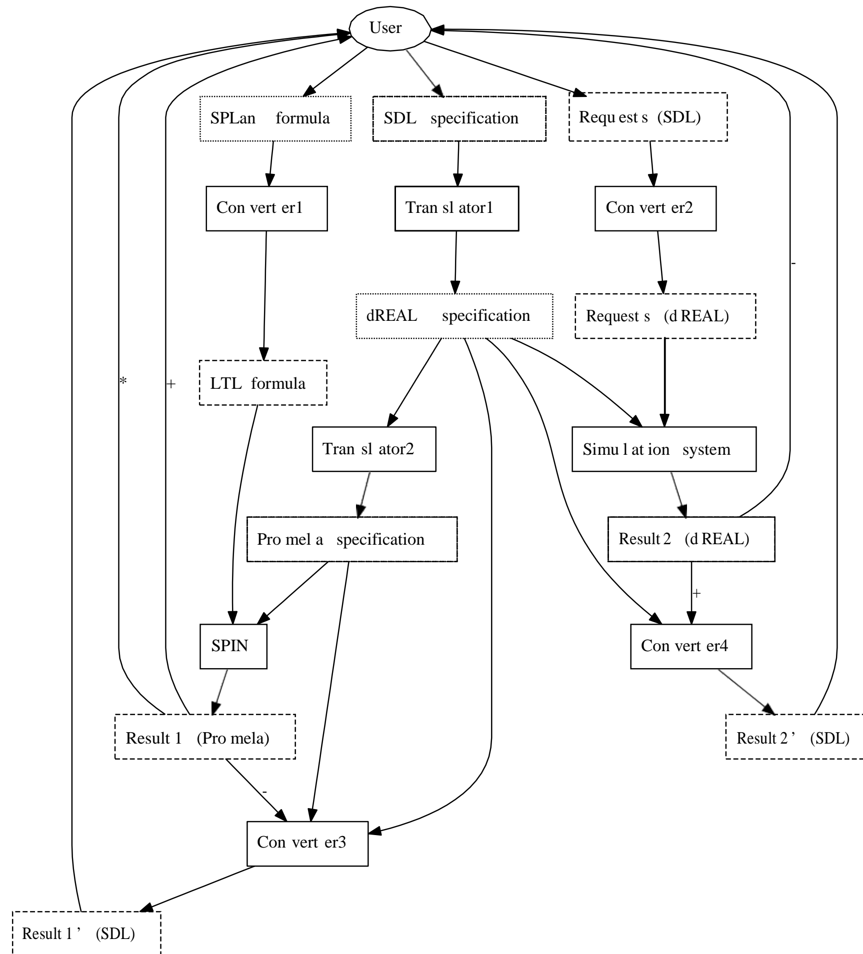

Figure 1. Software suite SRDSVer3

1. Current state of the SRDSVer3 system

The software suite SRDSVer3 is intended for the modeling, analysis, and verification of the SDL specifications using the dREAL intermediate language.

The SRDSVer3 suite (Figure 1) consists of the following components:

- Translator1 compiles an SDL specification into a dREAL specification
- Converter2 transforms the requests for the SDL specification into requests for the dREAL specification

- Simulation system analyzes the dREAL specifications according to the user's requests
- Converter4 transforms the Result2 (dREAL-related) of the Simulation system into Result2´(SDL-related)
- Translator2 compiles the dREAL specification into a Promela specification
- Converter1 transforms an SPLan-formula (SDL-related) into a corresponding LTL formula
- SPIN verification system checks the Promela specification received from Translator2 with the LTL formula received from Converter1
- Converter3 converts Result1 (from the SPIN verification) into Result1´(SDL-related).

2. Example (a case study)

We have recently used the following example to demonstrate the SRDSVer3 system.

It is an ATM (automated teller machines) network with the dynamic creation of client processes. The ATM network discussed in [4] and, with some changes, in [3] consists of several processes (Figure 2):

- one server with the data on clients and their accounts, and
- a fixed number of terminals (instances of the terminal process) to communicate with the clients.

To imitate the clients coming and going, the following additional processes were added to the SDL specification:

- one queue process that receives signals from the external environment containing information about the clients' intentions, creates an instance of the client process and sends to it the information received, and
- clients (instances of the client process).

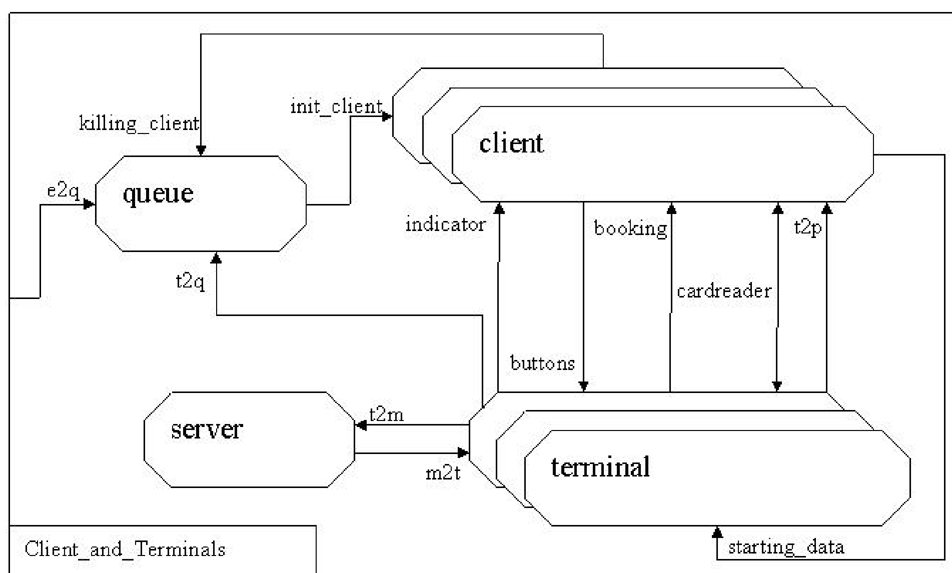

Figure 2. SDL diagram, overview

The complete SDL specification can be found in the GitLab repository [12].

3. Using Golang

Since the concept of channels and signal passing is inherent to Golang, it looks promising to try to use a Go program to simulate the case study. In a way, it was a manual translation from the SDL to Golang.

Since the author is not an expert in Golang, some design choices may be far from perfect, so any feedback would be appreciated. The most noticeable drawback for *Gofers* (those who use Golang) is that the identifiers do not follow the *camelCase* convention (all words but the first in a complex idenitifier have an initial uppercase letter), and use the *snake_case* instead (all words are lower-cased and separated with underscore). The preference of the snake case convention is attributed to the idea to make the program look more similar to the original SDL specification.

Below we consider how the components of the SDL specification are implemented in Golang. The complete Golang program can be found in the GitLab repository [13].

3.1 Overall structure

The program is implemented as a Go module, where the processes or entities of the same kind (such as channel definitions and external environments) are located in separate .go-files.

3.2 Signals

The signals without parameters described in the SDL as follows

```
SIGNAL
    ...
    terminal_no_money,
    no_money,
    correct_code,
    wrong_code,
    ready_for_job,
    press_start,
    ...
```

are represented in Go as the following structure with the signal name and sender field storing the PId of the process that sends the signal (in the SDL, this PId automatically goes to a special variable SENDER once the signal is read):

```
type signal_info struct {
    sig_name string
    sender int
}.
```

Signals with up to two parameters are translated into

```
type two_ints_info struct {
sig_name string
first     int
second  int
sender int
}.
```

The SDL channels (or, strictly speaking, SIGNALROUTEs) allow passing signals of different types; therefore, in Go we had to declare a channel so as to fit the signal with a maximal number of parameters, the unused parameters being zeroes.

3.3. Channels

The SIGNALROUTEs from SDL for signals with up to two parameters (translated into two_ints_info Go structures)

```
SIGNALROUTE t2m
    FROM terminal TO server WITH check_code, check_summ, balance,
insert_summ;
    SIGNALROUTE m2t
    FROM server TO terminal WITH summ, no_money, correct_code, wrong_code;
```

become two different kinds of channels, depending on whether the channel goes to a ‘simple’ process (without instances) or to a process with instances (such as a terminal or a client). In the latter case, the SIGNALROUTE is represented by an associative array (a map in the Go world), where the channels are indexed by the PIds of the instances.

```
// terminal -> server (the Machine)
var t2m chan two_ints_info
// server -> terminal
var m2t map[int]chan two_ints_info
//...
    t2m = make(chan two_ints_info, CHAN_SIZE)
    m2t = make(map[int]chan two_ints_info, TERM_COUNT)
    // Create all terminal-indexed channels here
    for i := 1; i <= TERM_COUNT; i++ {
            // ...
        m2t[i] = make(chan two_ints_info, CHAN_SIZE)
    }
```

Here the constants CHAN_SIZE and TERM_COUNT denote the maximum size of a channel (the number of signals it can contain) and the number of terminals (the instances of the terminal process), respectively.

3.4. Server

The server process receives requests from the instances of the terminal process and responds to the corresponding channel chosen by the sender field of the received signal. Below is the fragment designed to check if the PIN code of the card is correct:

```
for {  // Infinite loop
   //SDL: STATE main;
   sig := <-t2m
   term_num := sig.sender
   switch sig.sig_name {
   //SDL: INPUT check_code(cl_card_num, cl_code);
   case "check_code":
      cl_card_num  :=  sig.first
      cl_card_pin := sig.second
      result := "wrong_code"
      if code_table[cl_card_num] == cl_card_pin {
      result = "correct_code"
   }
   m2t[term_num] <- two_ints_info{sig_name: result}.
```

3.5. Terminals

An instance of the terminal process receives requests from the instances of the client process, and then it either sends a corresponding request for information to the server process or decides by itself (for example, if the terminal is out of cash, it cannot give money to the client). After that, the result is sent to the client process. Here is the fragment related to the checking of the PIN code:

```
   card_sig := <-card_reader[pid] // Read the card number from the card
   card_num  =   card_sig.first
   indicator[pid] <- signal_info{sig_name: "correct_card"} code_sig
   := <-buttons[pid] // Read the PIN code from the keyboard
   card_pin := code_sig.first
   t2m <- two_ints_info{
   sig_name: "check_code",
   first:            card_num,
   second:           card_pin,
   sender:           pid,
}
code_res_sig := <-m2t[pid] sig_name :=
code_res_sig.sig_name
indicator[pid] <- signal_info{sig_name: sig_name}.
```

3.6. Clients

An instance of the client process imitates the behavior of a client. The appropriate information is received from the queue process in the init signal.

```
   //SDL: INPUT init(cl_summ, cl_card_num, cl_code, terminal_pid,
      cl_operation);
   init := <-q2c[pid]
   term_num := init.term_num
   // ...
   card_reader[term_num] <- two_ints_info{sig_name: "card", first:
      init.cl_card_num}
sig_correct_card := <-indicator[term_num]
```

```
buttons[term_num] <- two_ints_info{
   sig_name:  "code",
   first:        init.cl_card_pin,
   sender:     pid,
}
sig_correct_code := <-indicator[term_num]
if sig_correct_code.sig_name != "correct_code" {
// ...
```

3.7. Handling the creation of instances (queue)

The instances of the terminal process are created at the start, whereas the instances of the client process are created only when a new client starts working with a terminal. Instances are modeled by signals sent from the external environment. In SDL, ENV is a special pseudo-process, while in Go, it must be explicitly specified as a separate process.

3.8. External environment (env)

The env process consists of a statement sending special client_info signals to the queue process. The signals look as follows:

```
CLIENT_1_PIN := code_table[CLIENT_1_CARD]
e2q <- client_info{
    cl_id:            CLIENT_1,
    term_num:      TERM_1,
    cl_operation: BALANCE,
    cl_card_num:
    CLIENT_1_CARD,
    cl_card_pin:
    CLIENT_1_PIN,
}.
```

3.9. Comparison with dREAL and Promela

Let us look at whether specifying a distributed system in Go is better because itis more concise.

- The original SDL specification [12] contains 428 lines.
- The result of translating it into dREAL is 2251 lines long.
- The Promela version generated from dREAL is even bigger: 2951 lines.

The Go files are 634 lines in total, which is greater than the original SDL specification but is much less than the other alternatives.

As for readability, the SDL specification looks better; in addition, it is a more standard way to describe distributed systems.

4. Conclusion

The approach presented in the paper has turned out to be not as satisfying as it originally looked. Moreover, it does not seem probable that software engineers using

Golang would use it as a specification language. They would rather write a prototype program of the system in question, and then run and debug it using some conventional methods. Thus, it may be better to try to verify the existing Golang programs by transforming them into formal models and then applying formal methods to them.
Further plans include learning more about the GOMELA tool ([6] and [8]) and either contributing to it or developing an alternative. Another possible direction of future work is to join the investigations at the Cyber-Physical Systems Laboratory of the Institute of Automation and Electrometry [14].

References

[1] Specification and Description Language. CCITT, Recommendation Z.100, 1988.

[2] ITU-T Specification and description language (SDL). ITU-T Recommendation, Z.100, 2015.

[3] Nepomniaschy V.A., Bodin E.V., Veretnov S.O. The language Dynamic-REAL and its application for verification of SDL-specified distributed system Programming and Computer Software // Programmirovanie.— 2015. — Vol. 41, No.1. — P. 41–48.

[4] Nepomniaschy V.A., Bodin E.V., Veretnov S.O. The analysis and verification of sdl-specifications of distributed systems using dynamic-real language // Vestnik TGU. Ser. Upravlenie, vychislitelnaja tehnika i informatika. — Tomsk, 2020. — No.53. — P. 118–126. DOI: 10.17223/19988605/53/12 (In Russian).

[5] Holzmann, G.J. The SPIN Model Checker. Primer and Reference Manual.— Addison-Wesley, 2004.

[6] Dilley N., Lange J. Bounded verification of message-passing concurrency in Go using Promela and Spin. — https://arxiv.org/pdf/2004.01323.pdf.

[7] The Gomela project - GitHub repository. —https://github.com/nicolasdilley/Gomela.

[8] Dilley N., Lange J. Automated verification of Go programs via bounded model checking // 36th IEEE/ACM International Conference on Automated Software Engineering. Proc. ASE 2021. — Melbourne, Australia, November 15–19. — 2021. — P. 1016–1027.

[9] The tool-chain implemented as part of paper “Automated Verification of Go Programs via Bounded Model Checking” GitHub repository. — https://github.com/nicolasdilley/gomela-ase21/ .

[10] Shi H., Ma W., Yang M., Zhang X.A. Case study of model checking retail banking system with SPIN // Journal of Computers. — 2012. — Vol. 7. — P. 2503–2510.

[11] Iqbal I.M., Adzkiya D., Mukhlash I. Formal verification of automated teller machine systems using SPIN // Proc. AIP Conf. 1867 (1): 020045. — 2017. DOI:10.1063/1.4994448.

[12] SRDSVer3 examples - GitLab repository. — https://gitlab.com/iis10/srdsv3.

[13] Golang programs - GitLab repository.— https://gitlab.com/iis10/atmnetwork.

[14] Garanina N.O., Staroletov S.M., Zyubin V.E., Anureev I.S. Model checking process-oriented IEC 61131-3 structured text programs // System Informatics. — 2023. — No.22. — P. 21-30.