applied sciences

Article

Discrete Time Model for Process Meta Language with Fictitious-Clock

Boštjan Vlaovič * and Aleksander Vreže

Faculty of Electrical Engineering and Computer Science, University of Maribor, Koroška Cesta 46, 2000 Maribor, Slovenia; aleksander.vreze@4s.si
* Correspondence: bostjan.vlaovic@um.si; Tel.: +386-41-769-386

Abstract: Industries like telecommunications, medical, automotive, military, avionics, and aerospace use complex real-time systems. Specification and Description Language (SDL) is one of the leading domain specific languages that is formally defined by international standards and well established in describing such systems. To check system properties abstracted model of the system is prepared in selected modeling language. We use Spin (Simple Promela Interpreter) model checker that is one of the leading tools for verification of complex concurrent and reactive systems. This paper focuses on modeling the SDL timer construct. It is one of the SDL constructs that is not easily modeled with Promela, but is present in many SDL systems. After an overview of the related work we propose a new Discrete Time Model for Promela (DTMP) that is seamlessly integrated in our framework for modeling SDL systems and can be used with the mainstream version of the Spin tool. To the best of our knowledge, this is not possible with the existing solutions. We describe how DTMP can be used to model SDL systems that use timers. Experimental results demonstrate its applicability to non-SDL systems with Fischer's mutual exclusion protocol and the Parallel Acknowledgment with Retransmission that were used in prior studies. We compare state-space requirements with one of the existing solutions DT Promela and DT Spin. With that, virtues and shortcomings of this high-level solution are exposed. We have shown that DTMP is effective when an extensive range of timer expiration values are used, which is usually the case in real-life SDL systems.

Keywords: formal specifications; formal languages; discrete time; model checking; automated extraction; SDL; Promela; Spin; Sdl2pml; SpinRCP

check for updates

Citation: Vlaovič, B.; Vreže, A. Discrete Time Model for Process Meta Language with Fictitious-Clock. *Appl. Sci.* **2022**, *12*, 2990. https://doi.org/10.3390/app12062990

Academic Editor: Flavio Cannavò

Received: 20 January 2022
Accepted: 10 March 2022
Published: 15 March 2022

Publisher's Note: MDPI stays neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Copyright: © 2022 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (https://creativecommons.org/licenses/by/4.0/).

1. Introduction

Telecommunication systems consist of heterogeneous nodes. They are controlled by different microcomputer systems running programs that were developed in various programming languages. Due to many concurrent activities, they usually describe complex behaviors. Checking of the correctness properties of such systems is not a trivial task. Therefore, use of formally-defined languages is recommended for specification of such systems. Specification and Description Language (SDL) is one of the leading domain specific languages [1] that is defined formally by international Standards [2–6] and well-established in industries like telecommunications, medical, automotive, military, avionics, aerospace, internet of things, system engineering [7], and others [8,9]. Versions SDL'88 and SDL'92 are used by our industry partner Iskratel d.o.o. for formal specification of concurrent, reactive, and distributed systems, e.g., MG6114AX (Media Gateway), IC1020AX (IMS Core), CS6116AX (Call Server, Unified Communications), IE1020AX (IMS Edge), IA1020AX (IMS AS), and CE6111AX (Compact Call Server).

The SDL system specifications describe the implementation details and are used to build the production systems. Such specifications contain SDL constructs and additional extensions that enable developers to include operators that are implemented in other programming languages. The C programming language is used for low-level or processor intensive operations. We work mostly on semi-formal specifications of telecommunication systems, since parts of the system consist of Abstract Data Type (ADT) operators written in C. Formal verification with the model checking technique can help improve the reliability of such systems. It can check automatically if the model of the system is in accordance with the system's requirements.

Formal verification can be performed by many software tools. We selected a Spin (Simple Promela Interpreter) tool [10]. It is a software package for verification of complex concurrent and reactive systems. The model of the system is described in a high-level language, named Promela (Process Meta Language). Like SDL, Promela adopts the strong formal basis established in ECFSM (Extended Communication Finite State Machine). Such systems consist of a finite number of separate components, which act independently one from another, and interact through the exchange of messages over message channels, or by memory sharing, via global variables. Message passing can be buffered or unbuffered (rendezvous channel). Concurrency is asynchronous and modeled by interleaving. There are no assumptions made on the relative speed of the process executions [11,12]. Properties can be checked with assertions, accept label, progress label, or LTL (Linear Temporal Logic) formula that is translated automatically to never-claim. The never-claim process is Büchi Automata, and can express ω-regular properties. If a system violates a property, Spin provides a counterexample, i.e., an execution path that violates the property.

Model checking of a complex SDL system is comprised of many steps. First, a model of the system must be prepared. It could be constructed manually or extracted with some (semi)automated process. The latter requires less-knowledgeable verification engineers and is much faster, especially for comprehensive SDL specifications. Research is motivated by the need of the industry for the formal verification of the SDL systems that are in production but are still actively developed or fine-tuned to the various local requirements of the telecommunication markets. The goal is to automate model extraction of real-life complex SDL specifications that include many timers with a great range of values without the need of manual abstraction of the timers. Our research focuses on a semi-automated approach. Before the automated extraction of the model is used, environment of the SDL system should be modeled to provide the behavior that is required to check the requirement specification successfully [13]. Automated extraction is based on the formal description and is methodologically sound [14]. For that, we have prepared a Promela framework for the abstract SDL machine. It supports predefined and user-defined sorts, a communication subsystem, multiple process instances, many of the SDL constructs, including `SAVE` construct, priority input, and others. Additionally, it includes unique variables—probes. During the simulation and formal verification, probes monitor if rules of the abstract SDL machine are broken. To ease editing, examination, syntax and redundancy check, simulation, verification, and to transform a Spin simulation trail into standard Message Sequence Chart (MSC), our research group is developing the Eclipse Rich Client Platform Integrated Development Environment for the Spin Model Checker (SpinRCP) [15–17].

In Promela, we do not know the exact time interval that elapses between two events. Therefore, we can not model systems where correct execution depends on the timing parameters. This motivated us to check available solutions and try to apply them to our framework for modeling SDL Systems in Promela, e.g., DT Promela and DT Spin [11,18]. Based on previous experiences with Spin model checker extensions that were not able to keep up with the mainstream version of Spin, we have formed the following research question: Can a high-level Discrete Time Model be used with standard Promela to model a real-life SDL system with timers? Due to various shortcomings of existing solutions, we have decided to propose a new Discrete Time Model for Promela (DTMP) that could be added seamlessly to our framework. We have demonstrated use of the DTMP with the 6.4.6 version of the Spin tool. The research results that are presented in this paper are implemented in the *sdl2pml* tool and were verified with an industry-size SDL description of the complex system [13]. An additional overview of our framework is available in [14,19].

This paper is organized as follows. Section 2 presents an overview of related work which includes various extensions of the Spin tool to support modeling of discrete and dense time. Section 3 gives a detailed description of the proposed DTMP that can be used with the mainstream version of the Spin tool. Experimental results are presented and discussed in Section 4. We use models of Fischer's Mutual Exclusion Protocol and Parallel Acknowledgment with Retransmission protocol to demonstrate the use of DTMP and compare the verification results with the DT Promela models from [20]. We conclude with discussion and directions for further research work.

2. Related Work

G. J. Holzmann, author of the Spin tool, presented an experiment of validating SDL specifications with Supertrace, the predecessor of the Spin tool [21]. It provided an automatically generated model of an SDL specification. Timers were modeled with Promela processes that took only subclass of timer behaviors into account. Timeout was considered as a possibility that is either present or absent without relative timings in the system. Later, Supertrace was integrated into the tool Sdlvalid [22].

In [11] S. Tripakis and C. Courcoubetis provide formal semantics for untimed Spin and introduce formal semantics for Real-time Promela (RT Promela). An example of a quantitative property, that can be checked independently of a specific time unit (dense time), is: "between event A and event B at least 5 time units should pass" [23]. Their verification method consists of considering the emptiness of a Büchi automaton that is extended with a finite number of clocks. RT Promela introduces globally declared clock variables that could be scalars or arrays. Each statement can be expanded with an optional time part that acts as a guard. A detailed description is out of the scope of this article, but it should be noted that this method was implemented as an extension to the Spin tool, with additional object files which implement the symbolic operations on difference bound matrices representing real-time information. Fischer's mutual exclusion protocol was used as one of the examples to demonstrate the use of RT Promela [11]. As was pointed out in [24], RT-Spin is not compatible with the partial order reduction algorithm. The tool is not available for Spin versions later than 2.0. Therefore, we did not investigate its applicability to the models of the SDL systems.

B. Knaack presented his ongoing research on Real-time model checking based on timed graphs that would support verification of quantitative aspects of real-time systems at Spin Workshop [25], but we are not aware of any available Spin extensions that would implement this research.

H. Tuominen presented modeling of Nokia Telecommunications SDL (NTSDL), which is a dialect of the SDL [26]. Timers are modeled by the process that receives all requests for the `SET` statements. Upon the reception of the request, it sends a timer expiration message back immediately to the activating process. `RESET` statement is a procedure that removes the message associated with the timer expiration from the buffer of the timer process and the activating process.

D. Bošnački and D. Dams integrated support for Discrete Time Promela into prototype version of Spin [18,24]. The DT Spin tool is intended for verification of concurrent systems that depend on timing parameters. It allows quantification of the time elapse between events, by specifying the time slice in which they occur. In DT Promela, statements of a process are divided into time slices by putting `set` and `expire` at the beginning and the end of the time slice. The application of DT Promela and DT Spin to model checking SDL systems is presented in [12]. It is a two-step process. First, the SDL system is translated to Intermediate Format (IF) with the tool sdl2if that was part of the commercial ObjectGEODE tool which became unavailable [27,28]. Some SDL constructs, e.g., enabling condition and continuous signals, are transformed to more primitive constructs. Furthermore, it does not support dynamically created processes and ADT. All mentioned shortcomings are overcome in our Promela framework for SDL systems. In the second step the if2pml tool is used to perform the translation to DT Promela.

In DT Spin, time is divided into slices of equal length and indexed by natural numbers. The elapsed time is measured in ticks of a global digital clock that is increased only when all processes have finished execution of their actions for the current time slice [24]. That is accomplished with the use of predefined Promela statement `timeout` that becomes true when no other statement within the system is executable. Each timer is represented with a variable of the `short` data type. Value of the variable defines the number of ticks before the timer expires at value zero. Expired timer becomes inactive with the variable set to −1. Extension introduces six macros, that can be used to manipulate the timers (Listing 1):

1. `set(tmr,value)` is used to activate a timer with the number of ticks set by the value,
2. `expire(tmr)` evaluates if the timer has expired,
3. `reset(tmr)` deactivates the selected timer,
4. `delay(tmr,value)` activates the timer and blocks execution of the process until its expiration,
5. `udelay(tmr)` non-deterministically delays execution for an unbounded number of ticks or continues with execution without delay,
6. `bdelay(tmr,val,auxtmr)` implements bounded delay with the use of two timer variables.

Listing 1. DT Spin macros.

```
#define OFF (-1)
#define TOVAL (0)

typedef timer
{
short val=OFF;
}

#define set(tmr,value) tmr.val=value
#define expire(tmr) (tmr.val==TOVAL)
#define reset(tmr) tmr.val=OFF

#define delay(tmr,value) set(tmr,value); expire(tmr)
#define udelay(tmr) do :: delay(tmr,1); ::break; od

#define bdelay(tmr,val,auxtmr) \
set(tmr,val);\
do \
:: expire(tmr) -> break \
:: else -> if :: break; :: delay(auxtmr,1); fi\
od

active proctype __Timers()
{
end__Timers:    do :: timeout od;
}
```

Timers of a simple SDL system could be modeled in DT Promela only with the macros `set`, `reset`, and `expire`. In SDL, expired timer appears as a signal in the input port of the process instance. In [12] the authors describe how to use DT Promela timers (variables) to model timers in SDL (signals). For each timer, expiration is modeled with an additional guard for the timer expiration at choice statements that correspond to the states with no outgoing transitions. If the timer should be discarded, transition is made to the same state, otherwise, transition is taken upon timer expiration. In SDL, time advances only when there is no possible transition, therefore this presents an equivalent model. However, to implement support for timer parameters, save construct, asterisk state, asterisk input, implicit transition, and enabling condition fully, our framework models timers as signals. A timer can be associated with parameters that are defined with expressions at the Set-node. The value of evaluated expression is received as a signal parameter at the expiration of the timer.

The latest update of DT Spin at our disposal is an extension of the Spin tool version 4.1.1 that was kindly provided by the authors during the initial development of our Promela framework for SDL systems. DT Promela is supported by the *sdl2pml* tool, but the expiration of a timer is modeled with signals, e.g., `::expire(tmr) -> process_input_queue!tmr(_pid, parameter1, parameter2, ...)`.

To the best of our abilities, we did not find research papers that would include real-life SDL systems. Most of the research is based on simplified SDL models.

3. Discrete Time Model for Promela

An SDL specification is comprised of different types of constructs which should all be supported by the selected formal verification environment. This article focuses on modeling the SDL timer construct. It is one of the SDL constructs that is not modeled easily with Promela, but is present in many SDL systems [29–32]. A timer is an item owned by an process that causes a timer expiration signal to occur at a specified time at the input queue of the process [3]. In SDL `TIME` and `DURATION` are predefined data sorts with real values. In SDL real values can be presented by one integer divided by another, i.e., rational numbers. The origin of time is system dependent. `TIME` values are used to set the expiry time of timers. `DURATION` is used for the values to be added to timers and as the result of the difference between `TIME` values [2,3].

Real numbers are abstracted with integers in the Promela model of the SDL system. Therefore, only the integer readings of the actual times with respect to a digital clock are recorded in the trace. DTMP implements a fictitious-clock where multiple ticks can be inserted between the events. It records the same order of the real-time events, but there are no silent events inserted between the events in the trace as they are in DT Promela [18,33].

DTMP was inspired with the SDL notion of time, definition of timers, and its communication model. A timer in SDL is owned by a process instance and can be active or inactive. An active timer returns a timer expiration message to the owning process instance at a specified time. Each process can have multiple timers and process instances. In DTMP, each timer is assigned a unique Timer Identification Number (TID). Our framework supports dynamic creation of process instances. If a maximum number of instances is not defined in an SDL specification, we set it to 1 [34]. We acknowledge that this is not in accordance with [2], where the maximum number, if not defined, should be unbounded. Therefore, we encourage that the expected number of allowed instances are defined explicitly in the SDL description of the system.

Figure 1 illustrates the basic idea of the DTMP. We have introduced a static Monitor process that manages all timers. A timer is declared as a signal with the unique TID that is defined globally with `#define` directive. Each process can include multiple definitions of timers. When a process activates a timer, it sends a message with TID and the value of the timer to the predefined rendezvous channel `chan__timeout`. This message can be extended with the optional timer parameters. The timer can be reset in a similar manner.

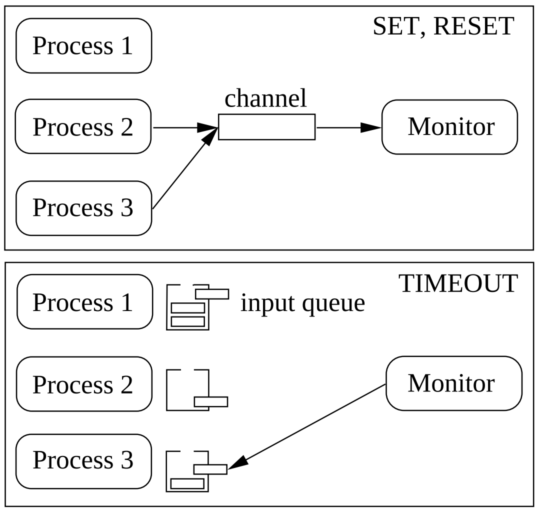


Figure 1. Basic idea of Discrete Time Model for Promela.

Time advances when no other statement within the system is executable. When the predefined Promela `timeout` event occurs the Monitor process sends messages to the input queues of all processes with expired timers. The input queue of the process instance is modeled with an associated Promela message channel. At each `timeout` event more than one timer can expire. This is in accordance with the SDL interpretation of timers and input queue management. DTMP extends an untimed Promela model with:

1. some temporary variables,
2. the synchronous communication channel `chan__timeout`,
3. timer manipulation inline functions `set` and `reset`,
4. the `Monitor` process,
5. timer expiration inline function `send_to_channel`.

Simple models can be prepared manually. Most of the DTMP functionality can be included with the dtmp.pml right before the proctype definitions. Only some macros and inline function `send_to_channel` has to be modified exclusively for each model of the SDL system. Contents of the dtmp.pml are shown in Listings 2 and 3.

The `TIMER_NUM__DATA_TYPE` limits the number of available timers in the model to the range of the `byte` data type. This can be increased, if needed, but is best kept to a minimum range. It defines the size of the array of timers that is stored by the `Monitor` process.

A timer is active if its value is greater than zero. The `TIMER_VAL__DATA_TYPE` is synonym for the `short` data type and defines the domain of timer values. If necessary, this can be changed to `int` which is the largest data type supported by Promela within the channel initializer [10]. Next, globally defined temporary variables are defined that are used by the `Monitor` process. They are excluded from the state descriptor during verification with the keyword `hidden` (lines 3–8).

When a timer is set or reset, processes communicate with the `Monitor` over the predefined synchronous channel `CHAN_TIMEOUT_PROC`, which is a synonym for the channel definition `chan__timeout` (Listing 2, lines 10–11). The number of the timers supported by the channel is limited by the `TIMER_NUM__DATA_TYPE`. Processes activate the selected timer with an inline function `set`. The body of the inline function is inserted directly into the body of the process as a replacement text for a symbolic name with parameters. It sends the TID (`tmr`) and its value (`value`) to the `CHAN_TIMEOUT_PROC` channel. Next, the Monitor returns the execution to the sending process with the return of the same signal. This prevents interleaving of processes during this statement. The timer can be reset with the `reset` inline function that sets timer value to 0. Additionally, the model of the SDL framework has to ensure that all pending timer expiration messages are removed from the input queue of the owning process instance.

Listing 2. dtmp.pml, part 1: Macros, variables, channel, inline `send`, and inline `reset`.

```
1  #define TIMER_NUM__DATA_TYPE byte
2  #define TIMER_VAL__DATA_TYPE short
3  hidden TIMER_VAL__DATA_TYPE unimportant_val;
4  hidden TIMER_NUM__DATA_TYPE tmp_timer_id;
5  hidden TIMER_NUM__DATA_TYPE tmp_counter;
6  hidden TIMER_NUM__DATA_TYPE min_val_index;
7  hidden TIMER_VAL__DATA_TYPE tmp_timer_id_val;
8  hidden TIMER_NUM__DATA_TYPE tmp_channel_offset;
9
10 chan chan__timeout=[0] of {TIMER_NUM__DATA_TYPE,TIMER_VAL__DATA_TYPE};
11 #define CHAN_TIMEOUT_PROC chan__timeout
12
13 inline set(tmr,value) { CHAN_TIMEOUT_PROC!tmr(value);
14 CHAN_TIMEOUT_PROC?tmr(unimportant_val); }
15 inline reset(tmr) { CHAN_TIMEOUT_PROC!tmr(0);
16 CHAN_TIMEOUT_PROC?tmr(unimportant_val); }
17
18 show bool pv__proc_full_queue_error = false;
19 show bool pv__runtime_error = false;
20 show bool pv__timeout_send_error = false;
```

Listing 3. dtmp.pml, part 2: Definition of process Monitor.

```
1  active proctype Monitor(){
2  TIMER_VAL__DATA_TYPE timers_val[NUMBER_OF_TIMERS];
3  do
4  :: atomic { CHAN_TIMEOUT_PROC?tmp_timer_id(tmp_timer_id_val);
5  timers_val[tmp_timer_id] = tmp_timer_id_val;
6  CHAN_TIMEOUT_PROC!tmp_timer_id(tmp_timer_id_val);}
7  }
8  :: atomic { timeout -> printf("MSC:␣TO\n");
9  atomic {
10 tmp_counter = 0;
11 min_val_index = 0;
12 do
13 :: tmp_counter < NUMBER_OF_TIMERS ->
14 if
15 :: timers_val[min_val_index] == 0 -> min_val_index++; tmp_counter++
16 :: (timers_val[tmp_counter] < timers_val[min_val_index] && \
17 timers_val[tmp_counter]!= 0) ->
18 min_val_index = tmp_counter;
19 tmp_counter++;
20 :: else -> tmp_counter++;
21 fi;
22 :: else -> break;
23 od;
24 }
25 tmp_counter = 0;
26 if
27 :: min_val_index == NUMBER_OF_TIMERS -> skip;
28 :: else ->
29 do
30 :: ((tmp_counter < NUMBER_OF_TIMERS) && (timers_val[tmp_counter] == \
31 timers_val[min_val_index])) ->
32 if
33 :: (timers_val[tmp_counter] != 0) ->
34 send_to_channel(tmp_counter);
35 if
36 :: tmp_counter == min_val_index -> skip; tmp_counter++;
37 :: else -> timers_val[tmp_counter] = 0; tmp_counter++;
38 fi;
39 :: else -> tmp_counter++;
40 fi;
41 :: tmp_counter == NUMBER_OF_TIMERS -> break;
42 :: else -> tmp_counter++;
43 od;
44 tmp_counter = 0;
45 do
46 :: tmp_counter < NUMBER_OF_TIMERS ->
47 if
48 :: ((timers_val[tmp_counter] != 0) && (tmp_counter != min_val_index))  ->
49 timers_val[tmp_counter] = timers_val[tmp_counter] - timers_val[min_val_index];
50 :: else -> skip;
51 fi; tmp_counter++;
52 :: tmp_counter >= NUMBER_OF_TIMERS -> break;
53 od;
54 timers_val[min_val_index] = 0;
55 fi;
56 tmp_counter = 0;
57 min_val_index = 0
58 }
59 :: pv__proc_full_queue_error == true -> assert(false);
60 :: pv__runtime_error == true -> assert(false);
61 :: pv__timeout_send_error == true -> assert(false);
62 od;
63 }
```

The `Monitor` process executes in a loop (Listing 3). It includes five option statements (guards). Executed statements are selected non-deterministically between the executable guards. The first option statement (lines 4–7) checks if a signal with two parameters can be received at the `CHAN_TIMEOUT_PROC` channel. This happens when some process changes the value of one of its timers. Timer identification number and value of the timer are stored in temporary variables `tmp_timer_id` and `tmp_timer_id_val`, respectively. Next, the value of the timer is stored in the `timers_val` array where TID is used as an index. Finally, the execution is transferred back to the sender. The whole option has to be executed as an atomic sequence.

The second option statement is executed when all processes are blocked and a predefined Promela `timeout` event occurs. Then, timers that should expire are found. Figure 2 presents the expiration of timers in DTMP graphically. In this example, four timers (t_1, t_2, t_3, t_4) are active. Their values are 4, 6, 10 and 6. A timer is inactive when its value is equal to 0. At each `timeout` event, timers that hold the minimum value are found. In the presented example, the timer t_1 expires at the first `timeout` event. To ensure correct expiration order, the value of t_1 is subtracted from the remaining active timers. Next, timers t_2 and t_4 should expire, but timers t_1 and t_4 are set to 8 and 5, respectively, before the next `timeout` event. Therefore, timer t_3 expires the next `timeout` event. The values of the remaining active timers (t_1, t_4) are decreased by 4.

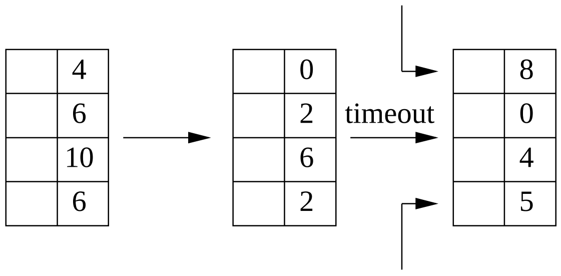


Figure 2. Expiration of timers.

Process Monitor finds the minimal expiration value of active timers by exploring the array of timers `timers_val` with the help of temporary index `tmp_counter` (Listing 3, lines 12–23). The first option statement of the if selection construct skips deactivated timers. The second option statement is executable if the active timer at the current temporary index has a new minimal expiration value. Then, variable `min_val_index` is updated with the TID. When `NUMBER_OF_TIMERS` is reached, the search is completed. Next, the inline `send_to_channel` is used to send a timer expiration messages (lines 29–43). The inline `send_to_channel` is adapted to the input queue of every process. Finally, values of all active timers are decreased (lines 45–53).

The maximum possible number of timers (`NUMBER_OF_TIMERS`) is acquired with the static analysis of the SDL specification. It is computed as follows:

$$\sum_{n=1}^{proc_t} num\_tmr\_proc_n * max\_inst_n, \tag{1}$$

where

- $proc_t$—is the number of process definitions in a model that include timers,
- $num\_tmr\_proc_n$—is the number of timer definitions in the n-th process, and
- max_inst_n—is the maximum number of instances of the n-th process.

The `Monitor` process also reports unexpected values of the automatically inserted probes that monitor execution of the Promela framework for the abstract SDL machine as defined in [14] (lines 59–61). If the input queue of the process is full, probe `pv__proc_full_queue` is activated. Probe `pv__runtime_error` detects errors in the Promela model of the SDL system. Probe `pv__timeout_send_error` detects addressing errors.

During the static analysis of the SDL specification all signals are enumerated. The timer expiration signals are identified in the Promela model with the use of TID. The range of TIDs is defined by the subset of the enumerated SDL signals:

- `TMR_BEGIN_INDEX`—TID of the first timer,
- `TMR_END_INDEX`—next free signal number in the model.

For each process we define similar macros where a unique prefix defines the location of the process within the SDL structure:

- `prefix__TMR_START_INDEX`—TID of the first timer used by the process,

- `prefix__TMR_END_INDEX`—value of the next free TID, and
- `prefix__TMR_NUMBER`—number of the timers in the process.

For example, the number of timers in the process `P` that is part of the block `B` would be defined in `B__P__TMR__NUMBER`. We have automated static analysis of the SDL specification and the definition of macros with the *sdl2pml* tool [19].

The Promela channel that models the SDL input queue of the specific SDL process should support reception of all possible SDL signals with parameters, including timer expiration signals. Therefore, the channel definition that models the input queue of a specific SDL process can be unique. If the process can have more than one instance, an array of channels is defined. Consequently, the definition of the inline `send_to_channel` has to be adapted for each system. It sends timer expiration messages to all process instances that include timers.

Listing 4 presents part of the inline that was generated by the *sdl2pml* tool for the real-world SDL system described in [13]. It includes only a small part of the inline definition that is dedicated to the process `L3netA` in the block `L3`. It demonstrates how macros are used to send a timer expiration message to the right process instance.

Let us assume that this particular system includes 49 SDL signals that are not associated with timers. Additionally, the specification permits five instances of the SDL process `L3netA` that utilizes four timers T1-T4. Imagine that process `L3netA` is the second process that is assigned TIDs and that the first analyzed process has only one instance that utilizes 10 timers. The following macros would be defined:

- `#define TMR_BEGIN_INDEX 50`—number of the signal that represents first TID,
- `#define L3__L3netA__TMR_START_INDEX 60`—number of the signal that represents the first timer used by the process `L3netA`,
- `#define L3__L3netA__TMR_END_INDEX 80`,
- `#define L3__L3netA__TMR_NUMBER 4`—number of utilized timers by each process instance,
- `chan__L3__L3netA 2`—the channel that models the input queue of the SDL process.

The first guard in the inline `send_to_channel` (Listing 4 checks if an expired timer is associated with the process `L3netA`. This is done by checking if the TID is within the expected range of values for all process instances (3–4). If this is not the case, probe `pv__timeout_send_error` is set to true, which triggers error at the Monitor process.

Listing 4. Definition of inline `send_to_channel`.

```
1  inline send_to_channel(tmr_id) {
2  if
3  ::(((tmr_id >= (L3__L3netA__TMR_START_INDEX - TMR_BEGIN_INDEX)) && \
4  (tmr_id < (L3__L3netA__TMR_END_INDEX - TMR_BEGIN_INDEX)))) ->
5  tmp_channel_offset = tmr_id - (L3__L3netA__TMR_START_INDEX - TMR_BEGIN_INDEX);
6  if
7  ::tmp_channel_offset != 0 ->
8  tmp_channel_offset = (tmp_channel_offset / L3__L3netA__TMR_NUMBER );
9  ::tmp_channel_offset == 0 -> skip;
10 fi;
11 tmp_timer_id =  TMR_BEGIN_INDEX + (tmr_id - tmp_channel_offset*L3__L3netA__TMR_NUMBER);
12 if
13 ::table_channum_ptr[chan__L3__L3netA[tmp_channel_offset]] == cv__buff ->\
14 pv__proc_full_queue = true;
15 ::else ->
16 chan__L3__L3netA[tmp_channel_offset]!tmp_timer_id(_pid,undefined__T_DChnlData,\
17 undefined__T_N_ISDNFrame,undefined__T_N_ISDNHead,pcv__null,pcv__null);
18 table_channum_nsp[chan__L3__L3netA[tmp_channel_offset]].\
19 data[table_channum_ptr[chan__L3__L3netA[tmp_channel_offset]]].name = tmp_timer_id;
20 table_channum_ptr[chan__L3__L3netA[tmp_channel_offset]]++;
21 fi;
22 ::else -> pv__timeout_send_error = true;   /* error - channel not found */
23 fi
24 }
```

Next, the channel offset has to be calculated for the process instance that owns the expired timer. Channel offset for the first instance of the process is equal to zero and increases by one for each new process instance. If the TID of the expired timer is 25, variable `tmp_channel_offset` is assigned the number of the timer within the model of the process, $25 - (60 - 50) = 15$. This represents timer T4 in the fourth instance of the process. Channel offset is obtained by dividing this number with the number of utilized timers by each process instance, $\lfloor 15/4 \rfloor = 3$ (line 8). In our example, calculated offset is associated with the fourth instance, as expected. Now, the number of the timer expiration message is calculated. Each timer is assigned only one unique timer expiration message number that is used by all process instances, $50 + (25 - 3 * 4) = 63$ (line 11). Message numbers 50–9 are assigned to the first analyzed process. For the timers utilized by process `L3netA` the following numbers are used: T1 $\leftrightarrow$ 60, T2 $\leftrightarrow$ 61, T3 $\leftrightarrow$ 62, T4 $\leftrightarrow$ 63. This confirms that timer T4 expired.

Constant value `cv_buff` defines the capacity of the message channel that models the SDL input queue of the process. To optimize the state-space of the model it should be kept to the minimum. By default the *sdl2pml* tool sets it to 4. However, if this limit is reached during the run of the system, execution of the model is not sound. If the input queue of the process is full, probe `pv__proc_full_queue` is activated and the process Monitor stops the evaluation of the model, due to the false assertion (line 14), and the buffer should be increased before the next run. If everything is in order, a timer expiration message is sent to the associated message channel `chan__L3__L3netA` with the calculated channel offset. The associated message channel should be able to receive data types from all signals used by the process, not just timers. During the analysis of the SDL system the *sdl2pml* tool obtains all SDL sorts used by the signals associated with the process and defines messages with a minimal number of fields, such that appropriate Promela data types for all used messages are included. Promela channel `chan__L3__L3netA` supports seven fields. Since the process does not use timers with parameters, most of the fields are not used and are just substituted with dummy values. After the message is delivered, the tables `table_channum_nsp` and `table_channum_ptr` are updated, which are part of the framework for modeling `INPUT` and `SAVE` SDL constructs.

We acknowledge that a manual construction of Promela model for larger SDL systems would be time consuming and error-prone. Therefore, the use of automated tools is recommended, such as *sdl2pml* [19]. During the static analysis of the SDL-specification all indexes for timers are calculated and analysis of process input queues is performed. It is followed by the extraction of the Promela model from the SDL system. DT Promela and DTMP are supported within our framework. Input to the *sdl2pml* consists of an SDL-specification file, command-line options, and a configuration file `sdl2pml.conf`. The latter can redefine values for some build-in parameters. Command-line options include support for predefined and user-defined probes as described in [34]. Additionally, a user can change many parameters of the automated generation of a model, e.g., default size of the channel buffer. All algorithms are implemented in C++ programming language and are based on their formal specifications in [34,35]. Now, simulation and verification can be run as described in [13].

This concludes the description of proposed DTMP. In the next section we demonstrate how DTMP can be simplified when the system under study is not complex.

4. Experimental Results

We have decided to check the time-space complexity of the formal verification with models of the Fischer's Mutual Exclusion Protocol (FMEP) and the Parallel Acknowledgment with Retransmission (PAR) that were used in prior studies [11,24,36]. Modification of existing models will demonstrate the applicability of DTMP for non-SDL systems and expose some of its virtues and shortcomings compared to the DT Promela.

4.1. Fischer'S Mutual Exclusion Protocol

Pseudo code for the well-known Fischer's Mutual Exclusion Protocol is shown in Listing 5 [37]. Atomic operations are written between "<" and ">". The critical section and the code that is not part of the mutual exclusion protocol must not modify any variable used by the algorithm. The **await** `<x == 0>` should be interpreted as **while** ¬ `<x == 0>` **do** `skip`. The algorithm is executed by a process with the unique identifier n. The delay operation must be long enough that process m, which evaluated its await statement before n executes `x = n`, can complete statement `x = m`. In this case, process n will not enter the critical section and will wait for the next opportunity.

Listing 5. Pseudo code of the Fischer's protocol.

```
repeat
<not critical section>
repeat
await <x == 0>
<x = n >
<delay>
until<x == n >
<critical section>
until false
```

Listing 6 shows the DT Promela model for FMEP that is based on the model available at [20]. Processes are created within initial process `init` (lines 21–26). The presented model has five process instances: `P(0)`, `P(1)`, `P(2)`, `P(3)`, and `P(4)` that execute concurrently. At the beginning, the value of the shared variable x is zero. After unbounded delay, the executing process instance checks if the value of the x variable is zero. If this is the case, it assigns x its identification value of $id + 1$. Next, the process waits for $deltaC$ ticks to enable competing process instance to enter the critical section (line 14). Before it enters into the critical section, it checks if it is still available—the value of the shared variable x still holds its identification value. If that is the case, the process enters the critical section. It stays in the critical section for an unbounded delay. Before it leaves the critical section, it sets the shared variable x to zero.

Listing 6. DT Promela model of the Fischer's Mutual Exclusion Protocol.

```
 1  #include "dtime.h"
 2
 3  #define N 5
 4  #define deltaB  1
 5  #define deltaC  2
 6
 7  byte x, in_crit;
 8
 9  proctype P(byte id){
10  timer y, y1;
11  do
12  :: udelay(y);
13  x == 0 -> atomic{delay(y1,deltaB) -> x = id+1;}
14  atomic{ delay(y, deltaC); } ->
15  atomic{ x == id+1; in_crit++; } ->
16  udelay(y1);
17  atomic{ x = 0; in_crit-- }
18  od
19  }
20
21  init
22  {
23  atomic {
24  run P(0); run P(1); run P(2); run P(3); run P(4);
25  }
26  }
27
28  never {
29  do
30  :: in_crit > 1 -> break;
31  :: skip
32  od
33  }
```

We have verified the model with the declaration of a temporal claim. The `never-claim` is used most commonly to specify behavior that should never happen. Only statements that can have a side effect on the system state are not allowed. If claim is present in the model of the system, each step along the execution path consists of a pair of transitions. First, the claim automaton is executed. The second transition is from one of the active processes. If the claim automaton does not have any executable transitions, the search along this path is stopped. The search then backtracks and explores other executions. The claim is defined as a series of propositions, or boolean expressions, on the system state that must become true for the behavior of interest to be matched. A never claim can be used to match either finite or infinite behaviors. Finite behavior is matched if the claim can reach its final state. In our case, this happens if the variable `in_crit` is greater than 1 which indicates that more than one process has entered the critical section (lines 28–33).

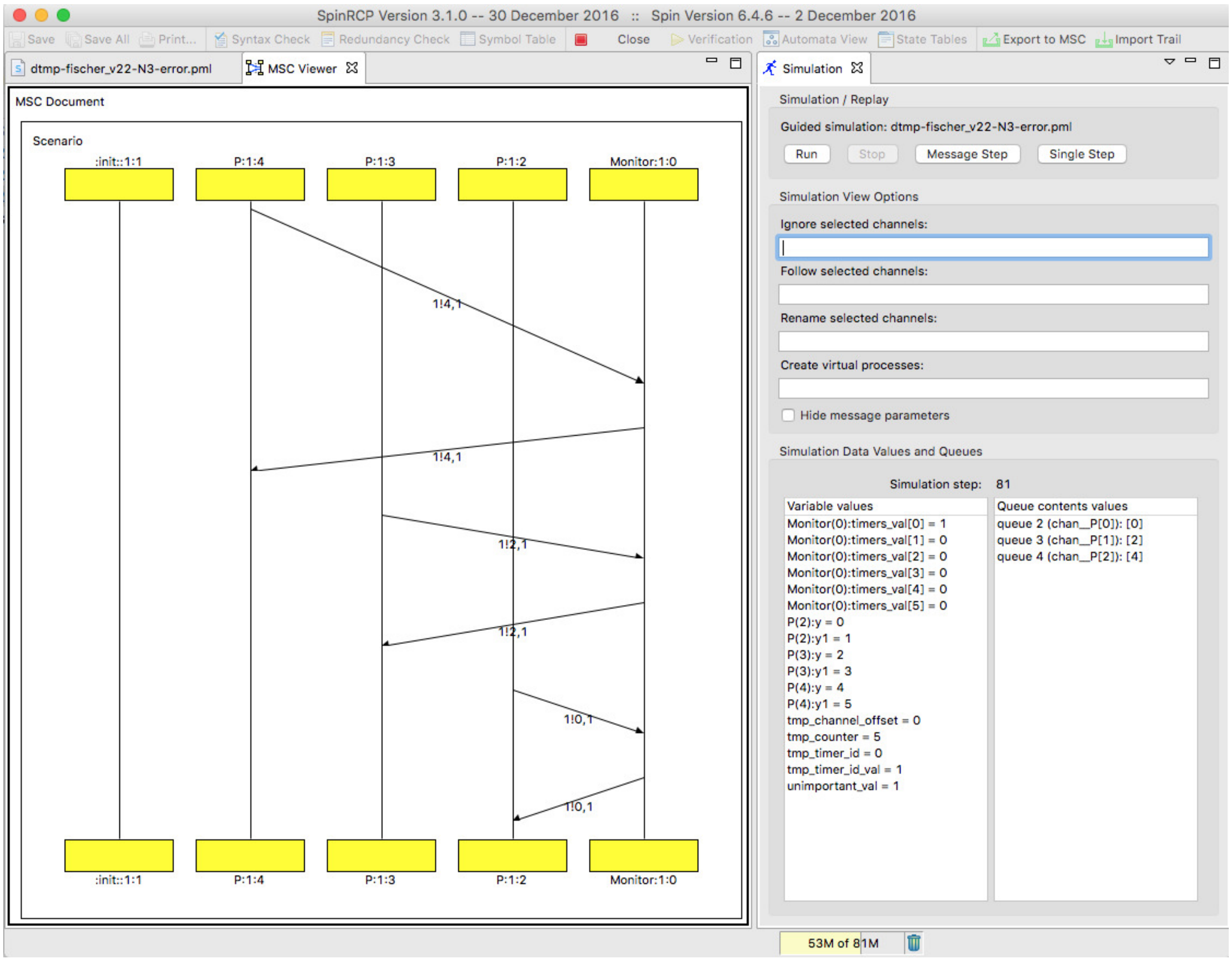
Listing 7 shows the DTMP of FMEP, without never claim and init process, which are the same as in the DT Promela model of the protocol. The expiration of a timer is modeled in DTMP with a message. Here we follow rules that are explained in Section 3. Because there are no other signals, `PROC_P__TIMER_START_IND` starts at 0. The `PROC_P__TIMER_END_IND` is set at the maximum number of timers. We have decided that each process will use two timers and `PROC_P__TIMER_NUM` is set accordingly. Input queues for the process instances are modeled with the array of channels `chan__P`. It suffices that associated channels `chan__P[id]` store only one message of the type `TIMER_NUM__DATA_TYPE`. The size of the array is the same as the number of process instances. In an attempt to prepare as similar a model to the DT Promela version as possible, we have modeled `expire`, `delay`, and `udelay` statements (lines 8–10).

Construction of the inline `send_to_channel` might have seemed complicated when the SDL framework was used (Listing 4), but here it is shown that its preparation can be straightforward for simpler models (lines 12–28). Timer identification number is used to find the process that owns the timer. First guard checks if TID is within defined limits (line 14). If this is the case, temporary variable `tmp_channel_offset` is assigned the index of the timer in the array of timers. This has to be divided by the number of timers used by each process. TID 0 and 1 are assigned to the process with ID 0, TID 2 and 3 are assigned to process with ID 1, etc. Next, the buffer of associated process is checked to see if it is within expected limits, and a timer expiration signal is sent to the selected channel. The model of the process P is very similar to the DT Promela model. Timers y and y1 are assigned TID at the start of the process. Now, a mainstream version of Spin can be used to verify the protocol formally.

To demonstrate the use of the Spin 6.4.6 and the SpinRCP 3.1.0 we have set `deltaB` to the value of 2 in the model with three active processes. Figure 3 shows the beginning of the MSC diagram for the guided simulation of the execution trail that violated the condition that only one process can enter the critical section. Process `P[2]` sets TID 4 to the value of 1 (`deltaB`). Process Monitor updates the `timers_val` table and returns the execution to the process. Next, timers with TID 2 and 0 are set to the same value. After the Promela timeout statement, process Monitor checks which timers are about to expire. Since all of the timers have expiration value 1, they all expire, and owning processes are sent the timer expiration message. The execution of the guided simulation was interrupted during the update of the array of timers, right before the update of the `timers_val[0]` which still holds the value of the expired timer. The timer expiration messages are not displayed in the MSC diagram because they have not been received by the processes thus far. They are still in the associated channels that hold the TID of the expired timers (Figure 3).

Listing 7. DTMP of the Fischer's Mutual Exclusion Protocol.

```
#define PROC_P__TIMER_START_IND 0
#define PROC_P__TIMER_END_IND   10
#define PROC_P__TIMER_NUM       2
#define cv_buff                 2
#define NUMBER_OF_TIMERS 10
chan chan__P[5] = [1] of {byte};

inline expire(tmr) { chan__P[id]?<eval(tmr)> -> chan__P[id]?tmr}
inline delay(tmr,value) { set(tmr,value); expire(tmr); }
inline udelay(tmr) { atomic{ do :: delay(tmr,1); ::break; od } }

inline send_to_channel(tmr_id){
if
:: ((tmr_id >= PROC_P__TIMER_START_IND) && (tmr_id <= PROC_P__TIMER_END_IND)) ->
tmp_channel_offset = (tmr_id - PROC_P__TIMER_START_IND);
if
:: tmp_channel_offset != 0 ->
tmp_channel_offset = (tmp_channel_offset / PROC_P__TIMER_NUM);
:: tmp_channel_offset == 0 -> skip;
fi;
if
:: len(chan__P[tmp_channel_offset]) == cv_buff ->  pv__proc_full_queue_error = true;
:: else -> chan__P[tmp_channel_offset]!tmr_id;
fi
:: else -> pv__runtime_error = true;
fi;
tmp_channel_offset = 0;
}

#include "dtmp.pml"

#define N 5
#define deltaB  1
#define deltaC  2
byte x, in_crit;

proctype P(byte id){
TIMER_NUM__DATA_TYPE y, y1;
atomic{
y = (id * PROC_P__TIMER_NUM) + 0;
y1 = (id * PROC_P__TIMER_NUM) + 1;
}
do
:: udelay(y); x == 0 ->  atomic{delay(y, deltaB); x = id + 1;}
atomic { delay(y, deltaC); } ->
atomic {x == id + 1 -> in_crit++;} ->
udelay(y);
atomic{ x = 0; in_crit--; }
od;
}

never{
do
:: in_crit > 1 -> break;
:: skip
od
}

init {
atomic {
run P(0);run P(1);run P(2);run P(3);run P(4);
}
}
```



Simulation / Replay

Guided simulation: dtmp-fischer_v22-N3-error.pml

Simulation View Options

Ignore selected channels:

Follow selected channels:

Rename selected channels:

Create virtual processes:

☐ Hide message parameters

Simulation Data Values and Queues

Simulation step: 81

```
Variable values                      Queue contents values
Monitor(0):timers_val[0] = 1         queue 2 (chan__P[0]): [0]
Monitor(0):timers_val[1] = 0         queue 3 (chan__P[1]): [2]
Monitor(0):timers_val[2] = 0         queue 4 (chan__P[2]): [4]
Monitor(0):timers_val[3] = 0
Monitor(0):timers_val[4] = 0
Monitor(0):timers_val[5] = 0
P(2):y = 0
P(2):y1 = 1
P(3):y = 2
P(3):y1 = 3
P(4):y = 4
P(4):y1 = 5
tmp_channel_offset = 0
tmp_counter = 5
tmp_timer_id = 0
tmp_timer_id_val = 1
unimportant_val = 1
```

Figure 3. Formal verification with Spin and SpinRCP.

A detailed analysis revealed that process P[1] and P[2] entered the critical section concurrently. As shown, SpinRCP can display global and local variables and the contents of channel queues. Additionally, the display of the MSC can be optimized with the virtual processes, selected channels can be ignored or followed explicitly [16].

We have performed exhaustive exploration with Partial Order Reduction (POR) of both models. We have used DT Spin 4.1.1, Spin 4.1.1, and Spin 6.4.6, all with hash-table size of 2^{18}, so that memory usage can be compared. We have performed eight verification runs with each tool. Valid execution of the protocol depends on the value of the timers. The value of `deltaC` must be greater than the value of `deltaB`. Verification confirmed than only one process can enter into the critical section concurrently. Verification results of the DTMP model with both versions of Spin were very similar. Table 1 summarizes results for the DT and mainstream version of Spin 4.1.1. It presents the number of states, number of transitions, state-vector size, memory usage, and time required for formal verification of models. The parameter `N` presents the number of concurrent processes. Additionally, to check the effect of higher timer values, we have increased the value of `deltaC` from 2 to 16. The `deltaB` variable was always equal to the default value of 1.

When `deltaC` was increased to 16, the fictitious clock of DTMP has generated a noticeably lower number of states. Additional communication and execution of the Monitor process generated more transitions. As expected, for this type of protocol, DT Spin's low-level implementation of discrete time outperforms the high-level approach of the DTMP.

However, DTMP is part of the SDL framework for the semi-automated model extraction from the SDL system.

Table 1. Results for Fischer's Mutual Exclusion Protocol for DP Promela and DTMP.

Tool	Options and Parameters			Formal Verification Results				
	deltaC	N	Option	States	Transitions	State-Vector [B]	Memory [MB]	Time [s]
DT Spin 4.1.1 (DT Promela)	2	2	POR	212	275	36	1.253	0.000
		3	POR	1880	2579	44	1.253	0.004
		4	POR	15,502	23,141	52	1.970	0.012
		5	POR	123,076	201,961	60	8.114	0.132
	16	2	POR	436	527	36	1.253	0.000
		3	POR	4316	5477	44	1.356	0.004
		4	POR	37,790	50,525	52	2.994	0.036
		5	POR	307,736	433,591	60	18.457	0.352
Spin 4.1.1 (DTMP)	2	2	POR	184	308	64	1.253	0.000
		3	POR	1751	3751	84	1.484	0.008
		4	POR	16,094	42,110	100	2.578	0.072
		5	POR	147,753	456,389	116	14.546	0.776
	16	2	POR	268	420	64	1.356	0.000
		3	POR	2885	5431	84	1.676	0.012
		4	POR	26,678	59,694	100	3.839	0.156
		5	POR	232,803	615,429	116	22.725	1.540

4.2. Positive Acknowledgment with Retransmission Protocol

The PAR protocol was described in [38]. It is a one-way data-link protocol used over unreliable transmission channels StoR and RtoS that connect the sender and the receiver. When a message is sent to the channel, three things can happen [36]:

1. the message is transfered correctly,
2. the message is corrupted in transit,
3. the message is lost.

Each time the message is sent to the StoR channel, the sender starts a timer. The sender waits for an acknowledgment before the new message is sent. Protocol uses only a positive type of acknowledgment. The message is re-transmitted at period of *To* until the receiving host acknowledges reception of the message. Premature expiration of the timer can disturb the functioning of the protocol, since duplicates of the message would be sent before positive acknowledgement managed to reach the sender. Consequently, there can be multiple acknowledgments in the channel RtoS. If the next message is lost in the channel StoR, the sender will mistakenly interpret the received acknowledgment as a confirmation of the successful transfer of the lost message. Therefore, the re-transmission period should be greater than the delays in both channels and the message processing time at the receiver.

The DT Promela model of the PAR protocol is shown in Listing 8 [20]. It consists of two process definitions, `Sender` and `Receiver`. Process instances are created during system initialization. Delay of the `StoR` channel is defined by the dK variable, while variable dL defines the delay of the `RtoS` channel. The value of the variable dR defines message processing time at the `Receiver`. The sum of both channel delays and the `Receiver`'s message processing time defines the minimum period of re-transmission, $To > (dK + dL + dR)$. Message values are calculated as modulo of number 8, which is defined by the `MAX`.

Listing 8. DT Promela model of PAR protocol.

```
 1  #include "dtime.h"                              /* timed features */
 2
 3  #define ACK 1
 4  #define dK  3
 5  #define dL  3
 6  #define dR  1
 7  #define To  8
 8  #define MAX 8                                   /* max number of different messages */
 9
10  chan StoR = [1] of {byte, bit}                  /* channels */
11  chan RtoS = [1] of {bit}
12
13  proctype Sender(chan in, out)
14  {
15  timer sc;
16  byte mt;                        /* message data */
17  bit  sn=0;                      /* sequence number*/
18
19  R_h:  udelay(sc);                     /* unbounded start delay */
20  mt = (mt+1)%MAX;                /* input from the upper layer */
21
22  S_f:  delay(sc,dK);
23  out!mt,sn;                      /* send and delay in channel StoR */
24  set(sc,To-dK);
25
26  W_s:  do
27  :: in?_ ->
28  if
29  :: atomic{skip; delay(sc, 1);
30  sn=1-sn; goto R_h;};      /* ack is OK */
31  :: atomic{printf("MSC:␣ACKerr\n");
32  goto S_f};
33  fi;
34  :: expire(sc) -> goto S_f;            /* timeout */
35  od;
36  }
37
38  proctype Receiver(chan in, out)
39  {
40  timer rc;
41  show byte mr;                    /* received message */
42  show byte me=1;                  /* expected message */
43  bit rsn, esn=0;                  /* received and expected sequence number */
44
45  W_f: in?mr,rsn;
46  if
47  :: rsn == esn -> goto S_h;               /* correct message and seq. num */
48  :: rsn == 1-esn -> goto S_a;             /* correct message, wrong  seq. num */
49  :: atomic{printf("MSC:␣MSGerr\n"); goto W_f;};
50  fi;
51
52  S_h:
53  assert(mr == me);                        /* expected message received */
54  delay(rc,dR);                            /* message processing delay */
55  atomic{esn = 1-esn; me = (me+1)%MAX};
56
57  S_a: atomic{delay(rc, dL); out!ACK};         /* send and delay in channel RtoS */
58  atomic{delay(rc, 1); goto W_f;};
59  }
60
61  init
62  {
63  atomic{
64  run Sender(RtoS, StoR);
65  run Receiver(StoR, RtoS);
66  }
67  }
```

Processes exchange messages using channels. The `Receiver` process accepts messages from the `Server` process through the `StoR` channel that can store one message with two fields, one for the value (`byte`) and one for the sequence number (`bit`). Sequence number is used as the positive acknowledgement that the `Server` process receives from the `Receiver` process over the `RtoS` channel.

After unbounded delay the `Sender` generates next message (line 19). Before message value `mt` and sequence number `sn` is sent to the `StoR` channel, the delay in the channel is modeled with delay of the *dK* ticks (line 22). Next, timer sc is set to expire at retransmission period *To*. Sender now waits for the reception of acknowledgment (line 27–33) or expiration of the retransmission period.

If any signal is received before timer sc expires, it is decided non-deterministically if positive acknowledgment is received or an error is reported and retransmission of the message and the sequence number is started at the label S_f. If acknowledge is accepted, execution is delayed for one tick and a new sequence number is calculated. Execution continues at the label R_h with the calculation of the new message value. If no signal has been received until retransmission period *To* (timer sc has expired), the execution continues at label S_f with the retransmission of the signal.

At the Receiver process, the received sequence number *rsn* is compared to the expected sequence number *esn* (lines 45–50). If they match, received message value *rm* is compared to the expected message value *em*. For this, an assert statement is used (line 53). Violation of this safety property would be reported during the verification and simulation run. Next, execution is delayed for message execution processing delay *dR*. It is followed with the calculation of the next expected sequence number and message value. After the delay of *dL* ticks in the channel RtoS, acknowledgment is sent to the Sender. Now, the Receiver process is ready to receive the next signal after the delay of one tick. Notice how statements in DT Promela are divided into time slices by putting set and expire at the beginning and the end of the time slice.

DTMP of the PAR protocol is shown in Listing 9. The system uses only two timers, one for each process. At initialization, each process is assigned identification number, defined as sender and receiver. These numbers are also used as timer identification numbers. Signal ACK is assigned the next free signal identification number. The StoR channel can hold one message with three fields reserved for the data (byte), sequence number (bit), and TID (TIMER_NUM__DATA_TYPE). Channel RtoS can hold one message with TIMER_NUM__DATA_TYPE field. Compared to the DT Promela model, which uses data type bit, in the DTMP model this field is shared by the ACK and the TID. Specification of the Sender and Receiver processes is the same as in the DT Promela model.

When Promela timeout occurs, the Monitor process finds timers with the lowest expiration numbers and sends timer expiration messages to the associated processes. If only one timer is used by the process specification, inline send_to_channel can be simplified greatly, as is shown in Listing 9 (lines 10–16). An expiration message is sent to the process which has the same ID as the TID. Inline expire has to take into account that processes use channels with different numbers of fields (lines 17–22). Guards evaluates which timer has expired and executes the correct receive statement. Inlines delay and udelay and are modeled in the same way as in the DT Promela model.

To check the influence of timer values on the time-space of the formal verification, we have increased the values of the delays tenfold for each additional verification run. We have tracked the number of verification states, transitions, size of the state-vector, required memory, and execution time. The results presented in Table 2 show that the required resources for formal verification with DT Spin increase when values of timers are increased, while the size of the timers does not influence the DTMP. This is especially noticeable in the last verification run where values that exceed the range of short data type were used. Before the verification, we have replaced the lines 4–7 in the dtime.h with typedef timer int val=OFF;. Similarly, we extended the range in DTMP with the changed definition #define TIMER_VAL__DATA_TYPE int.

Listing 9. DTMP of PAR protocol.

```
#define TIMER_NUM__DATA_TYPE byte
#define NUMBER_OF_TIMERS 2
#define pcv__null 0
#define sender   0
#define receiver 1
#define ACK 2

chan StoR = [1] of {byte, bit, TIMER_NUM__DATA_TYPE}
chan RtoS = [1] of {TIMER_NUM__DATA_TYPE}
inline send_to_channel(tmr_id){
if
:: tmr_id == sender -> RtoS!tmr_id;
:: tmr_id == receiver -> StoR!pcv__null(pcv__null,tmr_id);
:: else -> pv__timeout_send_error = true;
fi
}
inline expire(tmr_id) {
if
:: tmr_id == receiver -> in?eval(pcv__null),eval(pcv__null),eval(tmr_id);
:: tmr_id == sender -> in?eval(tmr_id);
fi
}
inline delay(tmr,value) {set(tmr,value); expire(tmr);}
inline udelay(tmr) {do ::atomic{set(tmr,1); expire(tmr);} ::break; od}

#include "dtmp.pml"
#define dK 3
#define dL 3
#define dR 1
#define To 8
#define MAX 8

proctype Sender(byte id; chan in, out) {
byte mt;
bit  sn=0;
byte tmp_sig;
R_h:  udelay(id);
mt = (mt+1)%MAX;       /* Input from the upper layer */
S_f:  delay(id,dK);
out!mt,sn,pcv__null; /*send and delay in channel K*/
set(id,To-dK);
W_s:  do
:: in?tmp_sig ->
if
:: tmp_sig == id -> goto S_f;
:: else ->
if
:: atomic{skip; delay(id, 1); sn=1-sn; goto R_h;};
:: atomic{printf("MSC:␣ACKerr\n");  goto S_f};
fi;
fi;
od;
}

proctype Receiver(byte id; chan in, out) {
show byte mr;
show byte me=1;
bit rsn, esn=0;
W_f: in?mr,rsn,pcv__null;
if
:: rsn == esn -> goto S_h;   /*correct message and seq. num */
:: rsn == 1-esn -> goto S_a; /*correct message, wrong  seq. num */
:: atomic{printf("MSC:␣MSGerr\n"); goto W_f;};
fi;

S_h: assert(mr == me);
delay(id,dR);
atomic{esn = 1-esn; me = (me+1)%MAX};

S_a: atomic{delay(id, dL); out!ACK};
atomic{delay(id, 1); goto W_f;};
}

init {
atomic{ run Sender(sender, RtoS, StoR); run Receiver(receiver, StoR, RtoS); }
}
```

Table 2. Results for the Parallel Acknowledgment with Retransmission for DT Promela and DTMP.

Tool	Options and Parameters		Formal Verification Results				
	Multiplier	**Option**	**States**	**Transitions**	**State-Vector [B]**	**Memory [MB]**	**Time [s]**
DT Spin 4.1.1 (DT Promela)	1	POR	632	690	48	1.253	0.000
	10	POR	2612	2670	48	1.356	0.004
	100	POR	22,412	22,470	48	2.911	0.016
	1000	POR	220,412	220,470	48	18.341	0.176
	10,000	POR, INT	2.2×10^6	2.2×10^6	52	183.602	2.692
Spin 4.1.1 (DTMP)	1	POR	1005	1282	60	1.356	0.000
	10	POR	1005	1282	60	1.356	0.000
	100	POR	1005	1282	60	1.356	0.000
	1000	POR	1005	1282	60	1.356	0.000
	10,000	POR	1005	1282	60	1.356	0.000
	100,000	POR, INT	1005	1282	68	1.356	0.004

5. Discussion and Conclusions

A new approach for modeling discrete time in Promela is presented. The previously known solution required a special version of Spin that do not include features from the recent versions of Spin, e.g., the inclusion of embedded C code for the ADT operators.

We have compared DTMP to the DT Promela and DT Spin. Modeling of discrete time with the DT Spin is clear and simple. A timer is declared locally in a process. It is shown that complexity of formal verification with DT Spin is influenced directly by number of timers and their values. For qualitative properties, where correct ordering of events suffices, this can be optimized with the proposed fictitious-clock model of discrete time without silent events [33].

During the development of DTMP we were focused on modeling real-life SDL systems. DTMP is high-level Promela model for the SDL timer construct that is seamlessly integrated in our framework for modeling SDL systems and can be used with the mainstream version of the Spin tool. Therefore, its implementation is influenced by the rest of the Promela framework for modeling the abstract SDL machine. To the best of our knowledge, this is the only framework that supports a full SDL system structure, predefined and user-defined data types, dynamic process creation and termination, multiple process instances, timers with parameters, save construct, asterisk state, asterisk input, priority input, implicit transition, `SAVE` construct, rational numbers, enabling condition, direct and indirect addressing, path limitations, procedures and some other less frequently used SDL concepts. We stress that all of the listed constructs must be supported to model SDL specifications with implementation details.

We have verified DTMP with complex SDL specification of an ISDN IUA protocol in [13], but further studies are necessary to check possible optimizations that would reduce the added complexity of this high-level solution. Our goal is to automate model extraction of real-life complex SDL specifications that include many timers with a great range of values without the need of manual abstraction of the timers. We have shown that DTMP is well suited for such systems, but we recognize that current research limitations include a low number of real-life specifications and lack of studies on large SDL systems (1 mio. lines of SDL code). Further research is necessary to explore the possible limitations of the method related to state space of the system and time needed for the formal verification of the model.

The new activities within SDL standardization bodies and industry is motivating. Algorithms that are implemented in *sdl2pml* do not support [3] specifics. We are planning to prepare new algorithms and extend the framework with support for new features, e.g., activation-delay for the output construct, priority order, and other. We see a potential to get fully Standards-compliant SDL descriptions of SDL systems extracted mechanically and verified formally by the mainstream version of the Spin tool. Additionally, possible support for the SDL-RT should be investigated [7,39].

Author Contributions: Conceptualization, B.V. and A.V.; methodology, B.V. and A.V.; software, A.V.; validation, B.V. and A.V.; formal analysis, B.V. and A.V.; investigation, B.V. and A.V.; resources, B.V. and A.V.; data curation, B.V. and A.V.; writing—original draft preparation, B.V. and A.V.; writing—review and editing, B.V.; visualization, B.V. and A.V.; supervision, B.V. and A.V.; project administration, B.V. and A.V.; funding acquisition, B.V. and A.V. All authors have read and agreed to the published version of the manuscript.

Funding: This research was funded partly by the European Regional Development Fund and Slovenian Ministry of Higher Education, Science and Technology under the R&D project Correctness Verification of Communication System Functioning in the framework of the Centre of Excellence for Information and Communication Technologies, Iskratel d.o.o, and 4S d.o.o. The authors acknowledge the financial support from the Slovenian Research Agency (research core funding No. P2-0069) by the research program Advanced methods of interaction in telecommunication.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Rosique, F.; Losilla, F.; Pastor, J.Á. A Domain Specific Language for Smart Cities. *Proceedings* **2018**, *2*, 148. doi: 10.3390/ecsa-4-04926. [CrossRef]
2. ITU-T Recommendation Z.100. CCITT Specification and Description Language (SDL), 1993. Series Z: Programming Languages. Available online: https://www.itu.int/rec/T-REC-Z.100-199303-S/en (accessed on 7 September 2021).
3. International Telecommunication Union. Z.100: Specification and Description Language—Overview of SDL-2010, 2021. Series Z: Programming Languages. Available online: https://www.itu.int/rec/T-REC-Z.100-202106-I/en (accessed on 7 September 2021).
4. Iternational Telecommunication Union. Z.101: Specification and Description Language—Basic SDL–2010, 2021. Series Z: Programming Languages. Available online: https://www.itu.int/rec/T-REC-Z.101-202106-I/en (accessed on 7 September 2021).
5. International Telecommunication Union. Z.102: Specification and Description Language—Comprehensive SDL-2010, 2021. Series Z: Programming Languages. Available online: https://www.itu.int/rec/T-REC-Z.102-202106-I/en (accessed on 7 September 2021).
6. International Telecommunication Union. Z.104: Specification and Description Language—Data and action language in SDL-2010, 2021. Series Z: Programming Languages. Available online: https://www.itu.int/rec/T-REC-Z.104-202106-I/en (accessed on 7 September 2021).
7. Pragmadev. Applications in Telecom, Military, Aeronautic, Space, Internet of Things, and System Engineering, 2022. Available online: https://www.pragmadev.com/telecom.html (accessed on 7 September 2021).
8. Fonseca i Casas, P.; Fonseca i Casas, A. Using Specification and Description Language for Life Cycle Assesment in Buildings. *Sustainability* **2017**, *9*, 1004. [CrossRef]
9. Zahid, S.; En-Nouaary, A.; Bah, S. Practical Model Checking of a Home Area Network System: Case Study. *J. Comput. Inf. Technol.* **2019**, *27*, 1–16.
10. Holzmann, G.J. *The SPIN Model Checker: Primer and Reference Manual*; Addison Wesley: Boston, MA, USA, 2003.
11. Tripakis, S.; Courcoubetis, C. Extending Promela and Spin for Real Time. Tool and algorithms for the construction and analysis of systems. In Proceedings of the 2nd International Workshop, TACAS'96, Lecture Notes in Computer Science, Passau, Germany, 27–29 March 1996; Springer: Berlin/Heidelberg, Germany, 1996; pp. 329–348.
12. Bošnački, D.; Dams, D.; Holenderski, L.; Sidorova, N. Model Checking SDL with Spin. In *Lecture Notes in Computer Science*; Springer: Berlin/Heidelberg, Germany, 2000; pp. 363–377.
13. Vlaovič, B.; Vreže, A.; Brezočnik, Z. Applying Automated Model Extraction for Simulation and Verification of Real-Life SDL Specification with Spin. *IEEE Access* **2017**, *5*, 5046–5058. [CrossRef]
14. Vlaovič, B.; Vreže, A.; Brezočnik, Z.; Kapus, T. Automated Generation of Promela Model from SDL Specification. *Comput. Stand. Interfaces* **2006**, *29*, 449–461. [CrossRef]
15. Brezočnik, Z.; Vlaovič, B.; Vreže, A. Model Checking using Spin and SpinRCP. *Inf. MIDEM J. Microelectron. Electron. Compon. Mater.* **2013**, *43*, 235–250.

16. Brezočnik, Z.; Vlaovič, B.; Vreže, A. SpinRCP: The Eclipse Rich Client Platform Integrated Development Environment for the Spin Model Checker. In Proceedings of the 2014 International SPIN Symposium on Model Checking of Software, SPIN 2014, San Jose, CA, USA, 21–23 July 2014; ACM: New York, NY, USA, 2014; pp. 125–128. [CrossRef]
17. Brezočnik, Z.; Kovše, T. SpinRCP. Faculty of Electrical Engineering and Computer Science, University of Maribor, Slovenia. 2020. Available online: http://lms.uni-mb.si/spinrcp/ (accessed on 12 December 2021).
18. Bošnački, D.; Dams, D. Discrete Time Promela and Spin. In *Lecture Notes in Computer Science*; Springer: Berlin/Heidelberg, Germany, 1998; pp. 307–310.
19. Vreže, A.; Vlaovič, B.; Brezočnik, Z. Sdl2pml-Tool for Automated Generation of Promela Model from SDL Specification. *Comput. Stand. Interfaces* **2009**, *31*, 779–786. [CrossRef]
20. Bošnački, D. DTSpin. Available online: http://www.win.tue.nl/dragan/DTSpin.html (accessed on 19 November 2021).
21. Holzmann, G.; Patti, J. Validating SDL Specifications: An Experiment. In Proceedings of the IFIP WG6.1 Ninth International Symposium on Protocol Specification, Testing and Verification, Enschede, The Netherlands, 6–9 June 1989; Vissers, C., Brinksma, E., Eds.; Elsevier Science Publishers: Amsterdam, The Netherlands, 1989 ; pp. 317–326.
22. Holzmann, G. Practical methods for the formal validation of SDL specifications. *Comput. Commun.* **1992**, *15*, 129–134. [CrossRef]
23. Tripakis, S. *Real-Time Spin (RT-Spin)*; Verimag: Saint-Martin-d'Hères, France, 1996.
24. Bošnački, D.; Dams, D. Integrating Real Time into Spin: A Prototype Implementation. In Proceedings of the FORTE XI/PSTV XVIII'98 IFIP TC6 WG6.1 Joint International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols (FORTE XI) and Protocol Specification, Testing and Verification (PSTV XVIII), Paris, France, 3–6 November 1998; Springer: Boston, MA, USA, 1998; pp. 423–439.
25. Knaack, B. Towards Real-time modelchecking using SPIN. In Proceedings of the 1997 International SPIN Symposium on Model Checking of Software, SPIN 1997, Enschede, The Netherlands, 5 April 1997.
26. Tuominen, H. Embedding a Dialect of SDL in PROMELA. In *Lecture Notes in Computer Science, Proceedings of the 6th International SPIN Workshop on Model Checking of Software, Trento, Italy, 5 July 1999*; Springer: Berlin/Heidelberg, Germany, 1999; Volume 1680, pp. 245–260.
27. Baeten, J. Esprit Project 23498—VIRES (Verifying Industrial Reactive Systems). Available online: https://cordis.europa.eu/project/id/23498 (accessed on 18 January 2022).
28. Verilog. *ObjectGEODE—Method Guidelines, Version 1.0*; Verilog SA: Toulouse, France, 1997.
29. Mejdi, H.; Jedli, B.; Hasnaoui, S. Designing a FlexRay controller—From SDL to StateFlow and Simulink blocks: Generation and verification. In Proceedings of the 2017 8th International Conference on Information and Communication Systems (ICICS), Irbid, Jordan, 4–6 April 2017; pp. 29–33. [CrossRef]
30. Soufiane, Z.; Abdeslam, E.N.; Slimane, B. An SDL to Discrete-Time PROMELA Transformation of Home Area Network Model. In Proceedings of the 12th International Conference on Intelligent Systems: Theories and Applications, SITA'18, Rabat, Morocco, 24–25 October 2018; Association for Computing Machinery: New York, NY, USA, 2018. [CrossRef]
31. Mazzanti, F.; Ferrari, A. Ten Diverse Formal Models for a CBTC Automatic Train Supervision System. *Electron. Proc. Theor. Comput. Sci.* **2018**, *268*, 104–149. doi: [CrossRef]
32. Asma, E.H.; Bensaid, H.; En-nouaary, A. Model Checking of WebRTC Peer to Peer System. *Comput. Inf. Sci.* **2019**, *12*, 56–71. [CrossRef]
33. Alur, R.; Dill, D.L. A Theory of Timed Automata. *Theor. Comput. Sci.* **1994**, *126*, 183–235. [CrossRef]
34. Vlaovič, B. Automatic Generation of Models with Probes from the SDL System Specification. Ph.D. Thesis, Faculty of Electrical Engineering and Computer Science, University of Maribor, Maribor, Slovenia, 2004.
35. Vreže, A. Extending Automatic Modeling of SDL Specifications in Promela with Embedded C Code and a New Model of Discrete Time. Ph.D. Thesis, Faculty of Electrical Engineering and Computer Science, University of Maribor, Maribor, Slovenia, 2006.
36. Vaandrager, F.W. Two Simple Protocols. *Appl. Process Algebra* **1990**, *17*, 23–44.
37. Lamport, L. A fast mutual exclusion algorithm. *ACM Trans. Comput. Syst.* **1987**, *5*, 1–11. [CrossRef]
38. Tanenbaum, A. *Computer Networks*; Prentice-Hall Software Series; Prentice-Hall: Upper Saddle River, NJ, USA, 1981.
39. SDL-RT. 2021. Available online: http://www.sdl-rt.org/ (accessed on 10 January 2022).