A Language Support for Exhaustive Fault-Injection in Message-Passing System Models

Masaya Suzuki Takuo Watanabe
Department of Computer Science, Tokyo Institute of Technology, Tokyo, Japan
draftcode@psg.cs.titech.ac.jp takuo@acm.org

This paper presents an approach towards specifying and verifying adaptive distributed systems. We here take fault-handling as an example of adaptive behavior and propose a modeling language Sandal for describing fault-prone message-passing systems. One of the unique mechanisms of the language is a linguistic support for abstracting typical faults such as unexpected termination of processes and random loss of messages. The Sandal compiler translates a model into a set of NuSMV modules. During the compilation process, faults specified in the model will be woven into the output. One can thus enjoy full-automatic exhaustive fault-injection without writing faulty behaviors explicitly. We demonstrate the advantage of the language by verifying a model of the two-phase commit protocol under faulty environment.

1 Introduction

As large-scale computing is more prevalent than ever, adaptability in software systems gains increasing importance. In particular, context-awareness and dynamic self adaptability in changing environment is crucial for developing sustainable systems. In this paper, we take fault-handling as an example of dynamic adaptive behavior in faulty circumstances and propose a method for describing formal models of fault-prone message-passing systems. The proposal is a linguistic approach; we designed and developed a modeling language Sandal that provides abstractions of typical faults such as unexpected termination of processes and random loss of messages.

The Sandal compiler translates a model into a set of NuSMV[4] modules. During the translation process, faults specified in the model will be woven into the output; in other words, the compiler performs a sort of software fault-injection (SFI). SFI and model checking are both fundamental techniques for developing reliable software. The combination of the both is a promising formal approach for verifying software systems in unreliable environments.

To model check a system, we usually describe the model of the system with a modeling language. According to the levels of abstraction, we categorize modeling languages into low-level and high-level languages. The former provides primitives for describing a model as a system of automata while the latter provides abstraction mechanisms for representing processes, channels, etc. NuSMV[4] and LOTOS[1] are examples of low-level languages, and Promela[6] and Rebeca[8] are categorized as high-level languages.

Both types of languages have their advantages and are used for modeling various systems. However, they too have difficulties in describing fault-prone systems due to the lack of proper mechanisms for expressing faulty behaviors. For example, injecting a fault to a message reception part in a model may affect not only the part itself and its nearby parts but also other modules of the model. These unwanted effects result in low modularity and maintainability of the model.

Automating fault-injection solves the above problem to some extent. For this purpose, some automation tools such as MODIFI[10] and FSAP/NuSMV-SA[2] have been developed. As these tools are

M. M. Bersani, D. Bresolin, L. Ferrucci and M. Mazzara (Eds.):
First Workshop on Logics and MODel-checking for self-* systems (MOD*)
EPTCS 168, 2014, pp. 45–58, doi:10.4204/EPTCS.168.4

designed to treat hardware faults, they are not suitable for our purpose. In addition, the above-mentioned modularity problem cannot be solved by these automation tools.

We adopt a linguistic approach to this problem and propose a high-level modeling language Sandal targeted to message-passing systems. Sandal provides language constructs for specifying typical faults including timeout, unexpected termination of processes, random loss of messages. The main advantage of this approach is the increased modularity of model descriptions owing to the language constructs. Another advantage is that the injection of faults does not affect the original fault-free behaviors of the model in an unexpected way.

We give the semantics of Sandal as a translation to automata. According to the semantics, we implemented a Sandal compiler that generates a set of NuSMV modules. Every fault specified in the model is injected automatically by the compiler in a non-deterministic way so that all possible fault scenarios are generated on the fly by the method referred as *exhaustive fault-injection*[9].

To demonstrate the advantage of our approach, we present a case study of two-phase commit protocol. In the case study, we compare the models written in Sandal and Promela from the viewpoint of modularity and verification performance. From a modularity viewpoint, Sandal achieves better performance with both the simplicity of specifying faults and the maintainability of models than Promela. The verification result shows that Sandal can properly inject faults into the model and can verify the properties regarding the faults as expected. The overhead of the verification speed of Sandal is in an acceptable range.

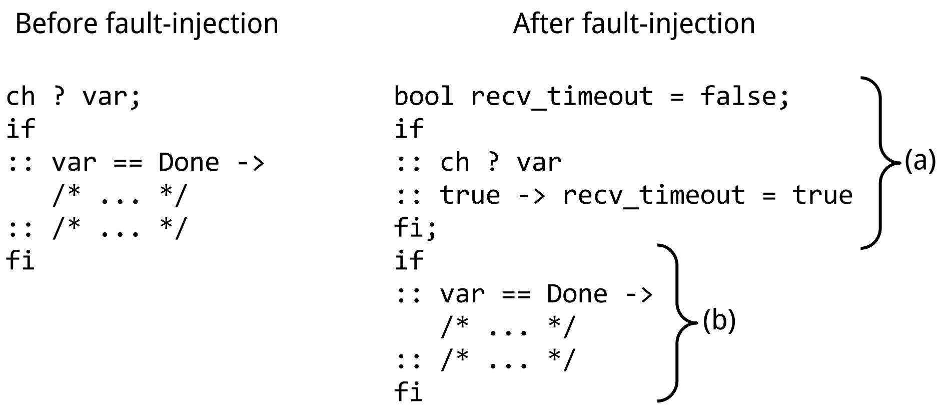
The rest of this paper is organized as follows. The next section gives a detailed discussion of the problem. Section 3 presents the overview of our modeling language Sandal, and Section 4 demonstrates its application to the two-phase commit protocol as a case study. Section 5 mentions some future work and Section 6 concludes with a summary.

2 Fault-Injection for Software Models

Injecting faults into a complex software model is sometimes a complicated process and requires manual instrumentation of the model. Part of the reason is that software models are not the target of current automatic fault-injection tools. They usually support the faults in a single variable like bit-flip or value-stuck, which are suitable for hardware models. In contrast, typical fault in software systems involves multiple actions and/or variables; hence it cannot be expressed in an error of a single variable.

The cost of the manual instrumentation depends on the language used to describe the model. If the model is constructed in a low-level modeling language that only provides the basic constructs representing state transitions in automata, users should describe the *foundation layer* of a system as well as the system behaviors. Here, the foundation layer means the part of the model that the *high-level* system model is built onto it. For example, to describe a system that uses message-passing communication, we need to model the messaging mechanism and the scheduler using the basic language primitives. On the other hand, if the model is constructed in a high-level modeling language that has the constructs corresponding to the foundation layer, we can easily build a system model on top of them.

Unfortunately, the both approaches incur some problems in modeling faults. In the low-level language approach, the distinction between the foundation layer and the system model is vague. This decreases the modularity of the model. To make things worse, the foundation and the injected faults sometimes affect the system model in an unexpected way. It is also the case that the primitives provided by the language is so inexpressive that faults cannot be implemented within the foundation layer. In this case, the fault descriptions ooze into the system model. The same problem may happen even if we use



Before fault-injection

```
ch ? var;
if
:: var == Done ->
   /* ... */
:: /* ... */
fi
```

After fault-injection

```
bool recv_timeout = false;
if
:: ch ? var
:: true -> recv_timeout = true
fi;
if
:: var == Done ->
   /* ... */
:: /* ... */
fi
```

Figure 1: Implementing Fault-Injection in a Promela Model

high-level languages. This kind of languages provide constructs regarding a foundation layer, but we cannot change its behavior. This means that it is hard to implement the fault happened in the functionality provided by the foundation layer. It might be possible to overcome this restriction by adding an abstraction between a system model and the foundation, but it leads the same problem as the approach of the low-level languages.

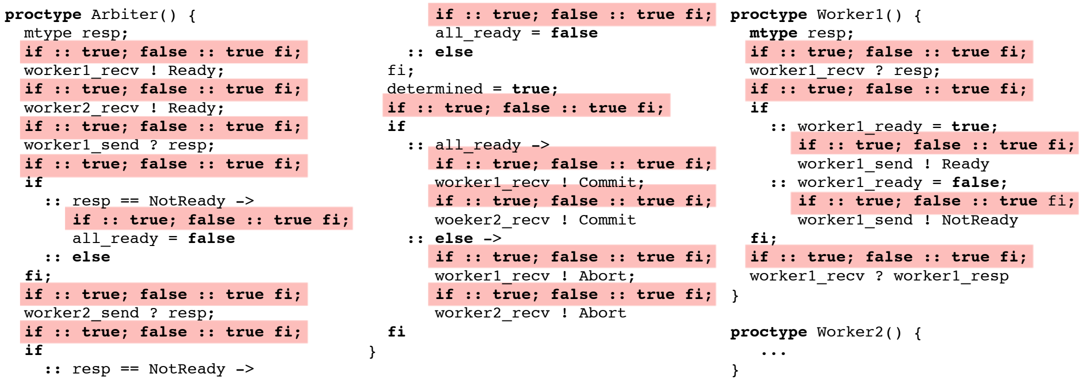
To illustrate the problems regarding fault-injection, we show two simple example models written in Promela. The first example is a timeout action in receiving messages. Due to the overload of network traffic, operations over the network sometimes take longer time than expected, and the system should give up the operation. Figure 1 shows an example of a timeout action implemented as fault-injection. The original fault-free process receives a message from the channel `ch` and does some computation based on the received value. In the fault-injected version of the model, the receive operation has a chance to timeout. The fault is implemented in the part (a). In this part, the timeout action is implemented as skipping the message reception.

Unlike other kinds of faults dealt with in this paper (random loss of messages and unexpected termination of processes), timeout faults are often explicitly handled in the models. This means that the part (b) of the model may treats the case in which the process fails to receive a message within an expected time.

Another example is the injection of unexpected termination faults. This fault emulates unexpected shutdown or failure of machines. It can be described in the way that processes have a chance to stop its execution. For example a simple Promela sentence

```
if :: true; false :: true fi
```

can model such behavior. The problem is that the above sentence should be inserted into the place where each observable action happen. Figure 2 shows the model after the fault-injection. The highlighted parts are the implementation of this fault. They are scattered in the model, and this may doubles the size of the resulting model and decrease the maintainability.



```
proctype Arbiter() {
  mtype resp;
  if :: true; false :: true fi;
  worker1_recv ! Ready;
  if :: true; false :: true fi;
  worker2_recv ! Ready;
  if :: true; false :: true fi;
  worker1_send ? resp;
  if :: true; false :: true fi;
  if
    :: resp == NotReady ->
       if :: true; false :: true fi;
       all_ready = false
    :: else
  fi;
  if :: true; false :: true fi;
  worker2_send ? resp;
  if :: true; false :: true fi;
  if
    :: resp == NotReady ->
       if :: true; false :: true fi;
       all_ready = false
    :: else
  fi;
  determined = true;
  if :: true; false :: true fi;
  if
    :: all_ready ->
       if :: true; false :: true fi;
       worker1_recv ! Commit;
       if :: true; false :: true fi;
       woeker2_recv ! Commit
    :: else ->
       if :: true; false :: true fi;
       worker1_recv ! Abort;
       if :: true; false :: true fi;
       worker2_recv ! Abort
  fi
}
```

```
proctype Worker1() {
  mtype resp;
  if :: true; false :: true fi;
  worker1_recv ? resp;
  if :: true; false :: true fi;
  if
    :: worker1_ready = true;
       if :: true; false :: true fi;
       worker1_send ! Ready
    :: worker1_ready = false;
       if :: true; false :: true fi;
       worker1_send ! NotReady
  fi;
  if :: true; false :: true fi;
  worker1_recv ? worker1_resp
}

proctype Worker2() {
   ...
}
```

Figure 2: A Promela model with unexpected termination of processes

3 The Modeling Language Sandal

3.1 The Overview of the Language

Sandal is designed to describe message passing systems. Message passing systems have processes communicating with each other only by message passing. In Sandal, a system consists of processes and channels. A process consists of a thread of execution and variables. Unlike the counterpart of real operating systems, it does not contain many threads. It has only one thread of execution. Sandal employs the shared-nothing model so processes cannot communicate via shared variables.

There are two types of channels in Sandal: rendezvous channels and buffered channels. With rendezvous channels, two processes can communicate in a synchronous way; both a sender and a receiver should be ready on the same channel to communicate. With buffered channels, two processes can communicate in an asynchronous way. The values sent are saved in the buffer of a channel. If a receiver wants to receive a value, it tries to pop out from the buffer.

To achieve fully automatic exhaustive fault-injection, Sandal is aware of the faults. Although there are many faults that can be considered, Sandal treats three types of faults: unexpected termination of processes, random loss of messages, and timeout in receiving a message.

Unexpected termination of processes is a fault that processes are unintentionally shut down in arbitrary timing. This is intended to emulate the real situation that one of the machines in a distributed system is crashed. Hardwares will, sooner or later, be broken. Even if one single computer has a low failure rate, the accumulated failure rate of the machines will be unignorable. If such faults are recovered and abstracted away by hardware, it is all right with software. Unfortunately, there is the case that cannot be recovered by hardware; therefore, a system's state should be recovered by software. Thus, the fault tolerance property for unexpected termination of processes is one of the basic requirement for distributed systems.

Random loss of messages is a fault that some messages are randomly dropped when sending them. This occurs when a message is sent by an unreliable way like UDP. Timeout in receiving a message is a fault that a transmission is not completed in some time window. They occur for various reasons: network overload, sending messages to the machine that is turned off, misconfiguration or maintenance

```
proc Starter(recv_ch channel { bool }, send_ch channel { bool }) {
  var v bool
  send(send_ch, true); recv(recv_ch, v)
}
proc Receiver(recv_ch channel { bool }, send_ch channel { bool }) {
  var v bool
  recv(recv_ch, v); send(send_ch, true)
}
init {
  P0: Starter(receiver_to_starter, starter_to_receiver),
  P1: Receiver(starter_to_receiver, receiver_to_starter),
  receiver_to_starter: channel { bool },
  starter_to_receiver: channel { bool },
}
```

Figure 3: An Example Model in Sandal

of a router, and so on. Such network unresponsiveness can be a temporary matter, so these faults may repeatedly appear. A possible solution is to send multiple copies of a message until the sender process receives the ACK for it. However this protocol can not guarantee the reception of the message.

We implemented an experimental compiler that generates a set of NuSMV modules from a Sandal model. The reason for employing NuSMV as the compilation target is that the semantics of a Sandal model can be expressed as a set of NuSVM modules in a straightforward manner. The source code of the compiler (including some sample models) is available at the first author's GitHub repository[1].

3.2 Syntax

The syntax of Sandal is similar to that of the programming language Go[5], which loosely follows the tradition of the programming language C. Figure 3 shows a simple model in Sandal that describes a system in which two processes exchange messages.

3.2.1 Process Templates

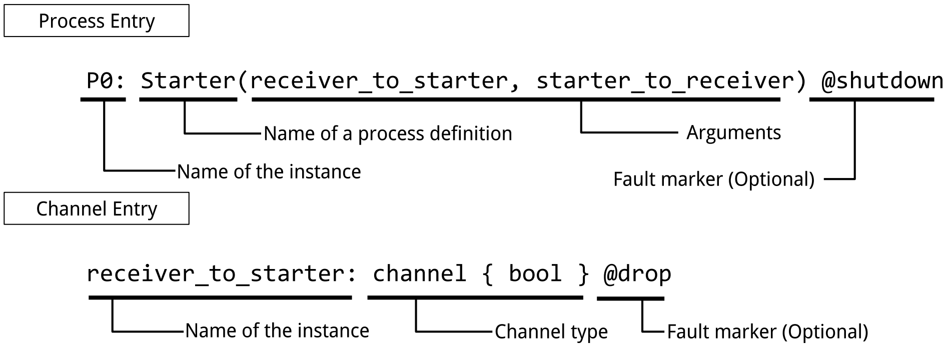
A process definition, a construct starting with the keyword **`proc`**, defines a template of processes. The identifier after **`proc`** is the name of the definition. In Figure 3, two templates named `Starter` and `Receiver` are defined. A list of parameters follows after the template name. The both template in the example have two parameters `recv_ch` and `send_ch` of type **`channel`** `{` **`bool`** `}`, a channel type whose message contents are boolean values. The last part of the template definition is a block (one or more statements wrapped in braces).

3.2.2 Init-Blocks

A block preceded by the keyword **`init`** is called an *init-block*. It describes the configuration of processes and channels in the system to be defined.

An init-block contains entries (called init-block entries) separated by commas. Each entry must be an instantiation of either a process or a channel. It starts with the name of the instance followed by a

[1] https://github.com/draftcode/sandal



```
P0: Starter(receiver_to_starter, starter_to_receiver) @shutdown
```

```
receiver_to_starter: channel { bool } @drop
```

Figure 4: Two Types of Init-Block Entries

colon and the rest part depends on what it represents (Figure 4). If it is an instantiation of a process, the name of a process definition and its arguments follow. If it is an instantiation of a channel, a channel type follows.

Fault markers can be attached to init-block entries. A fault marker attached to a process (or a channel) states that the specified fault may occur in the process (or channel). They are added to the last part of the entries in init-blocks. The current version of Sandal provides two fault markers **`@shutdown`** and **`@drop`**.

3.2.3 Messaging Statements

Because the statement and expression syntax of Sandal closely matches that of traditional programming languages, we only mention messaging statements in this subsection for brevity. Statements **`recv`** and **`peek`** are used for receiving values from a channel. The difference is that **`recv`** statements pop the values out from a channel while **`peek`** statements just copy them. These operations have non-blocking and timeout variants.

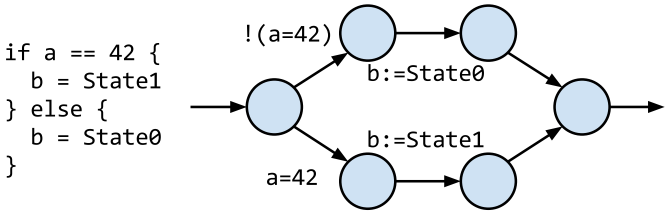
The arguments to these messaging statements are treated specially. The first argument should be a channel. This is a channel that is used to communicate. The rest of the arguments should be variable names. After the statement is executed, the received values are stored to these variables.

3.3 Semantics

3.3.1 Processes

A process in a Sandal model can be seen as a state machine. Each transition in the machine may have a condition (called *guard*) and side-effects (called actions). For example, a guard may be "*a value is ready to be received in the channel named c*" and an action may be "*receive a value in the channel and store it in the variable named v.*"

A process is a graph of statements that are executed in order. For example, Figure 5 shows a composite (**`if`**) statement and its semantics. There are two branches based on the **`if`** statement. They are merged into one branch after executing assignment statements. Every statement has semantics like this. The whole process can be expressed in an automaton that is a concatenation of the automata of its statements.



```
if a == 42 {
  b = State1
} else {
  b = State0
}
```

Figure 5: A Simple Statement and its Semantics

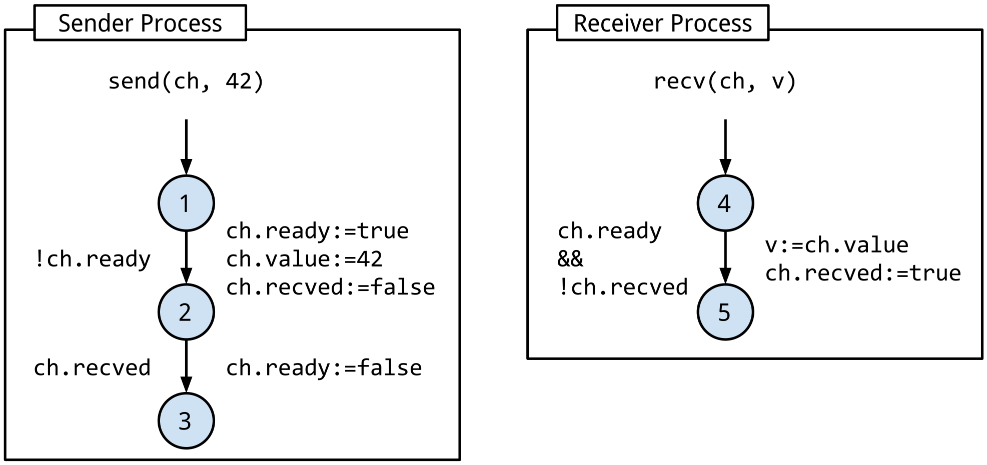


Figure 6: Two Processes Exchanging a Value

3.3.2 **send** and **recv**

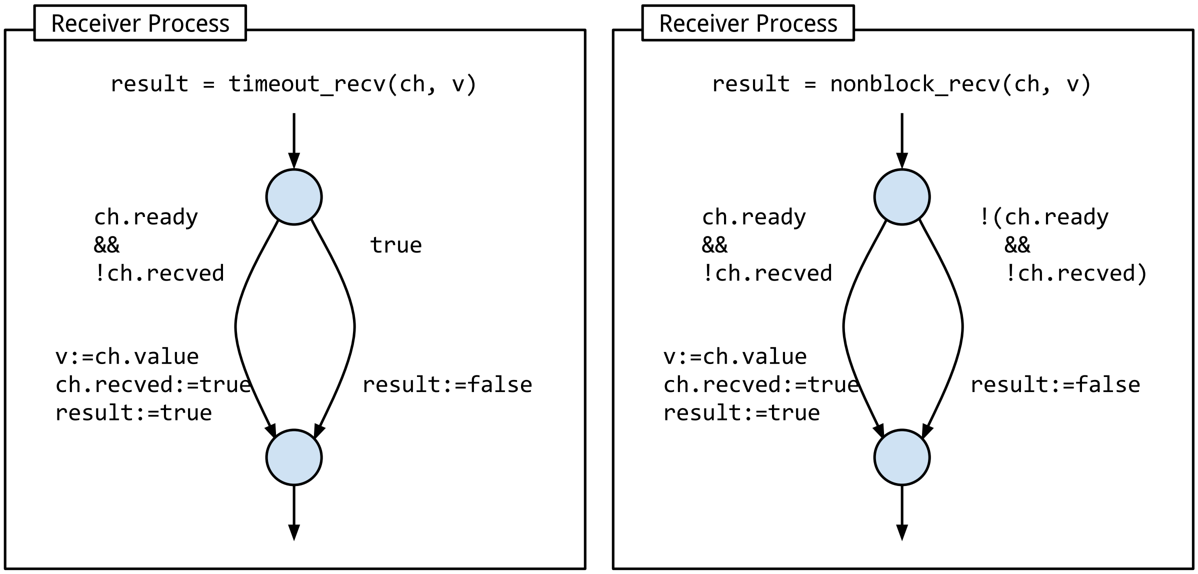
A pair of **send** and **recv** statements cooperate to exchange messages. Figure 6 shows the semantics of them. To show the process of this exchange, consider two processes trying to send and receive a value via a rendezvous channel. Sending a value via a rendezvous channel has been done by using three internal variables in a channel: a ready flag, a received flag and a value buffer. The initial states of flags are false. The procedure follows.

1. In the initial setting, two processes are at the state 1 and the state 4 (in Figure 6). The initial value of the ready flag is false. Thus the sender process can proceed to the state 2 while the receiver process cannot proceed to the state 5.

2. After the sender process steps to the state 2, the ready flag is true and the value to be sent is set to the buffer. At this point, the sender process is blocked because the received flag is false and the receiver process can make a step to the state 5. The receiver process receives a value from the buffer and set the received flag.

3. The sender process can proceed to state3, and the whole exchange process has been completed.

With this process, only one sender and receiver can communicate in a channel at once, and, even if one process tries to communicate, it blocks until the other process comes to communicate with it.



(a) timeout recv expression

(b) nonblock recv expression

Figure 7: `timeout_recv` and `nonblock_recv`

3.3.3 `timeout_recv` and `nonblock_recv`

Sandal provides two variants of `recv` statement: `timeout_recv` and `nonblock_recv`. Unlike `recv` statement, they are provided as functions because they should return boolean values that express the status of timeout.

A `timeout_recv` expression receives a value from a specified channel as `recv` statement. In addition, it may perform a timeout action modeled as the right branch shown in Figure 7 (a). If a value is successfully received (in Figure 7), the expression evaluates to true. Otherwise, it evaluates to false.

The expression may perform timeout action even if the corresponding sender process is ready to send a value. This is an intended behavior. Since network delay is unbound, the communication always has a chance to unable to complete a transmission in a specific time window. The behavior of `timeout_recv` expressions reflects these cases.

A `nonblock_recv` expression is another variant of `recv` statement. It receives a value only if it is ready. The behavior is modeled with two branches as shown in Figure 7 (b). Only one branch can be chosen since the guard of one branch is the negation of the other. If a value is ready and is successfully received, the expression itself evaluates to true. Otherwise, the expression evaluates to false.

The difference between `timeout_recv` and `nonblock_recv` is that the former is categorized as a fault. As Sandal injects faults in a non-deterministic manner, the timeout fault will be injected non-deterministically. On the other hand, `nonblock_recv` is not a fault and the additional behavior is not added in a non-deterministic manner.

3.3.4 Unexpected Termination of Processes

Unexpected termination of processes is a fault that a process is unintentionally shutdown. Using this fault, we can express machine crashes or process crashes. This type of faults will be injected to processes

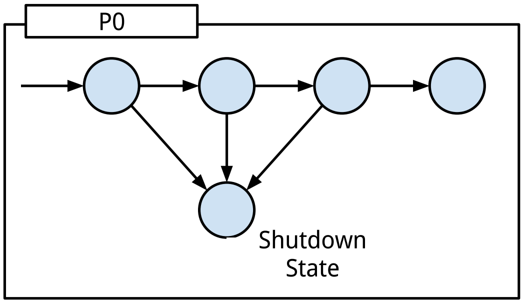


Figure 8: Unexpected Terminations of Processes

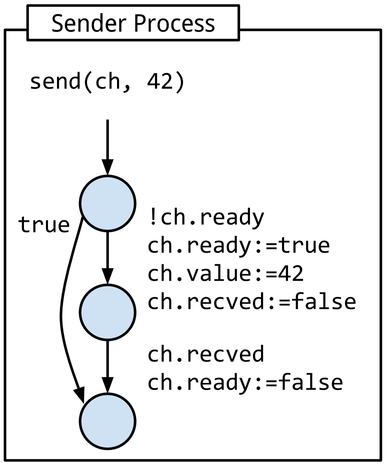


Figure 9: Random Loss of Messages

that have **`@shutdown`** fault markers.

To implement this fault, a shutdown state is introduced and transitions to the state are added in the target process (Figure 8). These newly added transitions have no guard conditions nor actions. The transitions are injected before and after the execution of statements. This means that each statement is an atomic action, and the termination fault does not interfere with their execution.

This type of fault are also implemented in non-deterministic way; the chance to execute a statement normally and the chance to go to the shutdown state are even. Model checker tries both choices and tries all combination of these choices. By harnessing non-determinism, model checker can simulate arbitrary shutdown scenarios.

3.3.5 Random Loss of Messages

Random loss of messages is a fault that some messages are dropped. Thus no receiver will be able to receive them. This type of faults will be injected to channels that have **`@drop`** fault markers, and all **`send`** statements over those channels start to drop a message occasionally.

The implementation of this fault is done by modifying the semantics of **`send`** statements of those faulty channels. The modified **`send`** statements may skip their normal behavior occasionally (Figure 9).

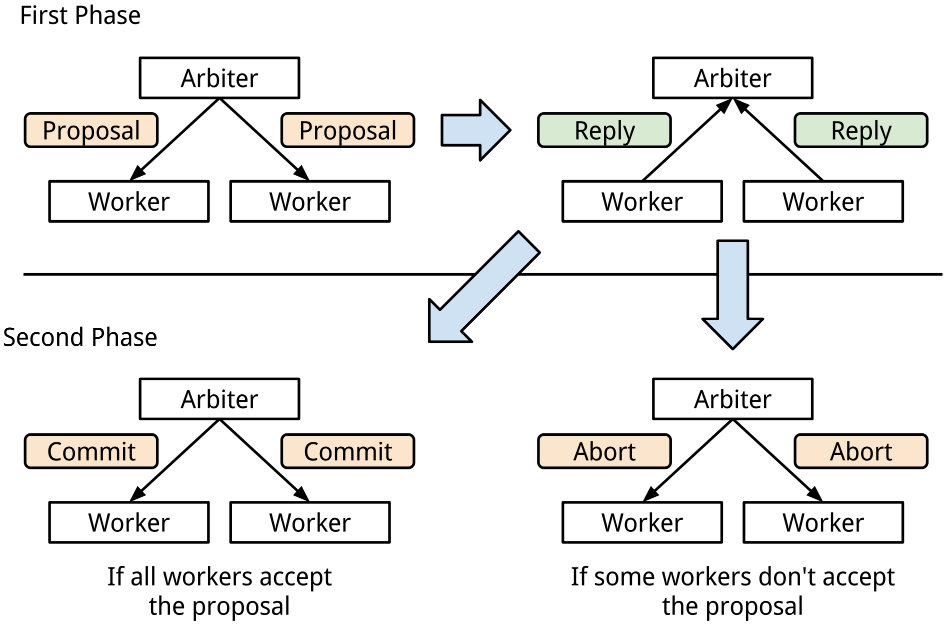


Figure 10: Two-Phase Commit Protocol

4 Case study

As a case study of this work, this section show the modeling and verification of two-phase commit protocol that is an algorithm to solve a consensus problem. It provides a way to determine a value which is acknowledged by all of the machine participated. It is used in major database systems such as MySQL to realize a transaction over multiple nodes.

The algorithm is performed by a single process called an arbiter and two or more processes called workers (Figure 10). The arbiter initiates the protocol and proposes a value. The workers receive requests from the arbiter and send replies to it. In the first phase of the protocol, the arbiter sends a proposal to the workers. Each worker checks the proposed value and replies whether it is acceptable or not. In the second phase, the arbiter aggregates the replies from the workers and see if all of the workers can accept the proposed value. If the value is acceptable, the arbiter sends a commit message to the workers. The workers received a commit message should accept the proposed value. If one worker replies the value is not acceptable in the first phase, the proposal fails, and the arbiter sends abort messages to the other workers.

In this case study, several models are written in Sandal and Promela. One is the model of two-phase commit protocol without any fault, and the rest is ones with faults. The injected faults are random loss of messages, unexpected termination of processes, and timeout in receiving messages. In each model, the safety property of two-phase commit is verified. A Sandal model with these faults is shown in Figure 11. Random loss of messages and unexpected termination of processes are injected by adding fault markers and timeout in receiving messages is injected by replacing recv statements of the arbiter with timeout recv statements.

The verification results show both the Sandal models and the Promela models that produce the valid results; the safety property holds for the model without faults and the model with timeout faults. The reason that the property holds with timeout faults is because the models fallback to the abort behavior if

```
data Response { Ready, NotReady, Commit, Abort }
proc Arbiter(chRecvs []channel { Response },
             chSends []channel { Response }) {
  var determined bool = false
  for ch in chSends {
    send(ch, Ready)
  }
  var all_ready bool = true
  for ch in chRecvs {
    var resp Response
    var recved bool = timeout_recv(ch, resp)
    if !recved || (recved && resp != Ready) {
      all_ready = false
    }
  }
  determined = true
  if all_ready {
    for ch in chSends {
      send(ch, Commit)
    }
  } else {
    for ch in chSends {
      send(ch, Abort)
    }
  }
}
proc Worker(chRecv channel { Response }, chSend channel { Response }) {
  var resp Response
  recv(chRecv, resp)
  choice { send(chSend, NotReady) }, { send(chSend, Ready) }
  recv(chRecv, resp)
}
init {
  chWorker1Send : channel { Response } @drop,
  chWorker1Recv : channel { Response } @drop,
  chWorker2Send : channel { Response } @drop,
  chWorker2Recv : channel { Response } @drop,
  arbiter : Arbiter([chWorker1Send, chWorker2Send],
                    [chWorker1Recv, chWorker2Recv]) @shutdown,
  worker1 : Worker(chWorker1Recv, chWorker1Send) @shutdown,
  worker2 : Worker(chWorker2Recv, chWorker2Send) @shutdown,
}
ltl {
  F (G (arbiter.determined &&
     ((!arbiter.all_ready) ->
        (!(worker1.resp == Commit) && !(worker2.resp == Commit)))))
}
```

Figure 11: A Two-Phase Commit Model in Sandal

Table 1: Sizes of 2PC Models

	Sandal	Promela
No fault	51 lines	66 lines
With Timeout	51 lines	74 lines
With Message loss	51 lines	70 lines
With Termination	51 lines	98 lines

Table 2: Verification Speeds of 2PC Models

	Sandal	Spin
No fault	0.96 sec	1.01 sec
With Timeout	2.88 sec	1.02 sec
With Message loss	2.11 sec	1.06 sec
With Termination	0.51 sec	1.17 sec

Table 3: Allocated BDD Nodes

No fault	925483
With Timeout	261547
With Message loss	369272
With Termination	588751

Table 4: States Stored

No fault	110 states
With Timeout	305 states
With Message loss	11 states
With Termination	7 states

the arbiter cannot receive workers' replies. The safety property does not hold for the model with random loss of messages and the model with unexpected termination of processes.

The sizes of the models are measured by lines. Table 1 shows lines of the eight models. Since Sandal has a built-in fault support, the sizes of the models do not grow even if some faults are injected, and the models do not lose their maintainability. The injected faults are well controlled by the language so that unwanted side-effects do not occur. The Promela models increase their sizes as some faults are injected. To overcome this issue, an automatic fault-injection tool is needed, but avoiding unwanted side-effects is still hard to accomplish.

Aside from the validity of the verification, the verification speed is also a matter of concern. The benchmark is taken using Linux 3.12.9 running on top of a PC with Intel Core i7-3770K 3.50GHz and 16GB memory. For model checking, we use NuSMV 2.5.4 (as the backend of Sandal) and Spin 6.2.5. The execution times needed to verify the models are shown in Table 2. It shows the verification speed of Sandal is still acceptable even if some faults are injected. It is interesting that the speed is increased or decreased when injecting faults in Sandal while there are no differences among the speeds of the Spin models. The reason for this is considered to be the difference of the model checking algorithms. NuSMV, the backend of Sandal, does symbolic model checking while Spin does explicit model checking.

The resources consumed in the verifications are shown in Table 3 and Table 4. They show the number of the BDD nodes allocated by NuSMV and the number of the states stored by Spin respectively.

No significant relationships between the BDD node sizes and the verification speeds can be observed. The number of states in some Promela models are very small. This is because Spin stops verification when it find the first counter-example.

5 Future Work

The result reported in this paper is a part of our ongoing work towards the verifiable framework for self-adaptable distributed systems based on a reflective architecture proposed by the second author[11]. To achieve the goal, we need to establish a modeling framework for general adaptable (or reflective) behaviors The current version of Sandal only provides limited features for modeling adaptable behaviors (a fixed set of fault-handling actions) as its language constructs. Based on this work, future work is discussed as follows.

Fault as a Cross-Cutting Concern A cross-cutting concern is a feature that affects multiple parts of the program. These concerns include authentication and logging. Because they are difficult to compose in a modular way in many cases, their implementation is scattered in the source code of the program. Therefore, separation of concerns principle is often violated.

Aspect-oriented programming (AOP) [7] is one of the approaches for this problem. AOP is motivated to increase modularity by enforcing separation of concern principle.

In the fault-injection approach proposed in this paper, faults can be captured as a cross-cutting concern. It affects multiple parts of a system, and the realization of a fault is scattered in a model. If the modeling language employ AOP, faults as well as other adaptable behaviors can be organized into aspects and incorporated into the model by a similar mechanism to aspect weaving. Actually aspect-oriented approach is proven to be effective not only in programming languages but also in modeling languages[12].

Feedback of Failure Detectors Failure detector is a mechanism that enables a machine to estimate failures in a system [3]. The failure it can treat varies. Most simple one is estimating other machine's crashes. The timeout feature of Sandal is also one of failure detection. It can detect a message cannot be sent in some time window and can feedback to the system. Failure detectors do only estimation due to the limitation of the reliability of themselves. Besides this limitation, they contribute constructing fault-tolerant distributed systems.

Giving feedback from failure-detection mechanisms to system models is one of the future work. The mechanism itself is sometimes unrelated to a system. It can be abstracted away from the system model specification, and, thus, modeling languages can provide a way to describe those mechanisms with modularity.

6 Concluding Remark

We propose a linguistic approach to reducing the cost of modeling fault-prone distributed systems. The key technology is a variation of software fault injection (SFI) applied to process models used for model checking. We designed and implemented a modeling language Sandal that supports the specification of typical faults in message-passing systems. Using the Sandal compiler, all possible faults specified in a model is automatically injected into the result that can be model checked by NuSMV. The advantage of the method is demonstrated by specifying and verifying models of the two-phase commit protocol.

Acknowledgments

This work is partly supported by JSPS KAKENHI Grant No. 24500033.

References

[1] Tommaso Bolognesi & Ed Brinksma (1987): *Introduction to the ISO specification language LOTOS*. *Computer Networks and ISDN Systems* 14(1), pp. 25–59, DOI: 10.1016/0169-7552(87)90085-7.

[2] Marco Bozzano & Adolfo Villafiorita (2003): *Improving System Reliability via Model Checking: The FSAP/NuSMV-SA Safety Analysis Platform*. In: *Computer Safety, Reliability, and Security (SAFECOMP 2003)*, *Lecture Notes in Computer Science* 2788, Springer-Verlag, pp. 49–62, DOI: 10.1007/978-3-540-39878-3_5.

[3] Tushar Deepak Chandra & Sam Toueg (1996): *Unreliable Failure Detectors for Reliable Distributed Systems*. *Journal of the ACM* 43(2), pp. 225–267, DOI: `10.1145/226643.226647`.

[4] Alessandro Cimatti, Edmund Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani & Armando Tacchella (2002): *NuSMV 2: An OpenSource Tool for Symbolic Model Checking*. In: *Computer Aided Verification*, *Lecture Notes in Computer Science* 2404, Springer-Verlag, pp. 359–364, DOI: `10.1007/3-540-45657-0_29`.

[5] *The Go Programming Language*. Available at `http://golang.org`.

[6] Gerard J. Holzmann (1997): *The Model Checker Spin*. *IEEE Transactions on Software Engineering* 23(5), pp. 279–295, DOI: `10.1109/32.588521`.

[7] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier & John Irwin (1997): *Aspect-Oriented Programming*. In: *ECOOP '97 – Object-Oriented Programming*, *Lecture Notes in Computer Science* 1241, Springer-Verlag, pp. 220–242, DOI: `10.1007/BFb0053381`.

[8] Marjan Sirjani, Ali Movaghar, Amin Shali & Frank S. de Boer (2004): *Modeling and Verification of Reactive Systems using Rebeca*. *Fundamenta Informaticae* 63(4), pp. 385–410. Available at `http://iospress.metapress.com/content/wg947keu129prhbd/`.

[9] Wilfried Steiner, John Rushby, Maria Sorea & Holger Pfeifer (2004): *Model Checking a Fault-Tolerant Startup Algorithm: From Design Exploration To Exhaustive Fault Simulation*. In: *International Conference on Dependable Systems and Networks (DSN '04)*, pp. 189–198, DOI: `10.1109/DSN.2004.1311889`.

[10] Rickard Svenningsson, Jonny Vinter, Henrik Eriksson & Martin Törngren (2010): *MODIFI: A MODel-Implemented Fault Injection Tool*. In: *Computer Safety, Reliability, and Security*, *Lecture Notes in Computer Science* 6351, Springer-Verlag, pp. 210–222, DOI: `10.1007/978-3-642-15651-9_16`.

[11] Takuo Watanabe (2013): *Towards a Compositional Reflective Architecture for Actor-Based Systems*. In: *Workshop on Programming based on Actors, Agents, and Decentralized Control (AGERE!@SPLASH 2013)*, ACM, pp. 19–24, DOI: `10.1145/2541329.2541341`.

[12] Kiyoshi Yamada & Takuo Watanabe (2006): *An Aspect-Oriented Approach to Modular Behavioral Specification*. In: *Proceedings of 1st Workshop on Aspect-Based and Model-Based Separation of Concerns in Software Systems (ABMB 2005)*, *Electronic Notes in Theoretical Computer Science* 163(1), Elsevier, pp. 45–56, DOI: `10.1016/j.entcs.2006.07.002`.