JOURNAL OF SOFTWARE: EVOLUTION AND PROCESS
J. Softw. Evol. and Proc. 2012; **00**:1–38
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/smr

Comparative Modeling and Verification of Pthreads and Dthreads*

Yuan Fei[1] Huibiao Zhu[1†] Xi Wu[1,4] Huixing Fang[5] Shengchao Qin[2,3]

[1]*Shanghai Key Laboratory of Trustworthy Computing*
MOE International Joint Laboratory of Trustworthy Software
International Research Center of Trustworthy Software
East China Normal University, Shanghai, China
[2]*School of Computing, University of Teesside, Middlesbrough, UK*
[3]*School of Computer Science and Software Engineering, Shenzhen University, Shenzhen, China*
[4]*School of Information Technology and Electrical Engineering, University of Queensland, Brisbane, Australia*
[5]*School of Information Engineering, Yangzhou University, Jiangsu, China*

SUMMARY

The POSIX threads (Pthreads) library is a thread API for C/C++ to control parallel threads and spawn concurrent process flows. Programming in Pthreads usually suffers from undesirable deadlock, data race and race condition problems due to the potential non-deterministic execution behaviors between parallel threads. Dthreads, as another multithreading model that re-implements Pthreads, was proposed by Liu *et al.* [1] for efficient deterministic multithreading. They found out that, under specific test cases, Dthreads can effectively prevent data races. However, no comparison test has been made with Pthreads.

To carry out a formal comparison between Pthreads and Dthreads over deadlocks, data races and race conditions, in this paper, we adopt CSP (Communicating Sequential Processes) as a formal model for specifying part of API functions in Pthreads and Dthreads, and illustrate the model construction using four classical example programs. By feeding the models into the model checker PAT (Process Analysis Toolkit), we have verified that deadlocks and data races exist in Pthreads, but do not exist in Dthreads, for the considered programs. We have also found that neither of them can prevent race conditions. Our comparative modelling and verification of Pthreads and Dthreads show that though Dthreads can not prevent all the deadlock situations, shown by verification results of another two example programs, Dthreads is better than Pthreads on eliminating data races and preventing deadlocks. Considering limited scalability of Dthreads, we have introduced a new programming model to support coarse granularity in bank transfer. Our modelling is also extended by covering the synchronization operations in Liu *et al.*'s work. Copyright © 2012 John Wiley & Sons, Ltd.

Received ...

KEY WORDS: Pthreads; Dthreads; Modeling; Verification; CSP

1. INTRODUCTION

The demands for multi-threaded programs increase rapidly with the development of multi-processors. However, the consequent concurrency issues, for example deadlocks [3, 4], data races [5, 6], and race conditions [7, 8], make concurrent programming more challenging. Deadlocks are often due to improper request/release of resources between multiple threads. Data races occur when

*A short version of this paper appeared in HASE 2016: 17 International Symposium on High Assurance Systems Engineering [2].
†Corresponding author.
E-mail address: hbzhu@sei.ecnu.edu.cn (H. Zhu).

two or more threads read and write the same memory unit concurrently. A race condition appears because of unsuitable timing or ordering of events.

Deadlock detections can be conducted via static or dynamic techniques. Static techniques [3, 9] use static analysis tools to analyse the codes to find out the locations where deadlocks may occur. However, they usually require high computation cost and may not be suitable for large-scale code. Dynamic techniques [6, 10] detect deadlocks along with the execution of the codes. Due to their dynamic nature, it would be difficult for them to detect all deadlock scenarios. Data race detections could also be divided into static analysis techniques and dynamic analysis techniques. Static analysis techniques [5, 11] check all execution paths to determine if there is a data race. However, incorrect alerts on data races may occur with the adoption of certain hypotheses, which can lead to a high rate of false positives. Dynamic analysis techniques [6, 12], conversely, monitor memory and synchronizing information at runtime to detect data races. They have higher accuracy, but need to improve code coverage, e.g. by considering multiple executions of programs. Finally, race condition detections are categorized into static analysis techniques and dynamic analysis techniques as well. Static analysis techniques [13] can only report the possible occurrences of race conditions on special programming environments. Therefore, manual analysis is needed when it comes to runtime environments. Dynamic analysis techniques [14] are of two kinds: modifying or interrupting system calls and checking log files. They both have their drawbacks. The former causes the loss of accuracy. The latter relies on the completeness of log files.

The uncertainty in the execution of threads is the fundamental cause for concurrency issues. The methods to eliminate uncertainty can be categorized into external methods and internal methods. The most common external methods are testing all the possible interleavings at runtime and embedding control logic at compiling time. For example, in [15], Petri nets-based control logic is added into the source codes to eliminate concurrency problems. However, such methods may cause high cost. Internal methods try to avoid the uncertainty. For example, Edwards *et al.* introduced a concurrent programming language called the *SHIM* for C-plus-Pthreads compiler [16], but it has a sharp learning curve and may be impractical for every programmer to learn it. Grace [17] can eliminate concurrency problems, but only for certain program idioms (e.g. fork-join).

The IEEE POSIX 1003.1c standard [18] specifies a thread programming interface that has been widely used to achieve portability. As an implementation to this standard, Pthreads [19] may cause concurrency problems [20]. Dthreads, proposed by Liu *et al.* [1], is a lightweight implementation of Pthreads, aiming to avoid such problems by guaranteeing deterministic execution. Some researches have been done on the comparison of Pthreads and Dthreads. In [21], iThreads, a threading library for parallel incremental computation, is compared with Pthreads and Dthreads on performance gains, scalability and performance overheads. A Framework for multi-threaded program including Pthreads and Dthreads is presented by Webber *et al.* [22].

There are several testing approaches [23, 24, 25] to verify multi-threaded programs by model checking. Some of them are devoted to Java and some of them are used for C program using the Pthreads library. Another important work on reasoning about locking in multi-threaded code is done at an abstract model level [26]. As there is no methods for Dthreads, we choose the model checker PAT [27, 28] as the same approach for Pthreads and Dthreads.

In [1], Dthreads is tested with cases and the result shows that Dthreads can effectively prevent data races. But they have not made comparison test with Pthreads over deadlocks, data races and race conditions. In this paper, we comparatively verify Pthreads and Dthreads in the scenario of four classical example programs based on several API functions. First, we employ CSP [29, 30] to model example programs with the API functions of Pthreads and Dthreads. As automatic verifications are useful in many fields [31, 32], we add the conversion from C code to PAT code. By using the model checker PAT, for our considered models converted from CSP models automatically, we verify the presence of deadlocks, data races and race conditions in Pthreads, and their absence in Dthreads. And we add two more example programs to rich our comparison using more synchronization operations. Our verification results of comparison for Pthreads and Dthreads show that Dthreads is better than Pthreads on eliminating data races and preventing deadlocks. In addition, we find out that neither of them can avoid race conditions.

DOI: 10.1002/smr

The rest of the paper is organized as follows. Section 2 gives a brief introduction of Dthreads, as well as the introduction of CSP. Section 3 introduces thread API functions and example programs. Section 4 is devoted to the modelling of Pthreads and Section 5 is about the modelling of Dthreads. Section 6 describes the conversion method from C code to PAT code, which is implemented by Java. In Section 7, we apply model checker PAT to verify deadlocks, data races and race condition. Section 8 extends our modelling by covering the synchronization operations in Liu *et al.*'s work. In Section 9, we also verify some new models applying the new API functions. Section 10 introduces a new programming model for bank transfer. Finally, Section 11 describes the conclusion and future work.

2. BACKGROUND

In this section, we briefly introduce the special characteristics of Dthreads compared with Pthreads and show the main mechanism of it. We also give a brief introduction to CSP.

2.1. Dthreads vs Pthreads

Dthreads and Pthreads have different design decisions on the following three aspects:

The way of implementing threads: In Pthreads, threads have shared address space. It means that when there is an update operation, the thread updates shared data directly. But in Dthreads, threads are implemented in the way of processes, which means they have separate addresses. Hence, the thread can choose to update shared data or its private copy under certain conditions.

The writing order to shared memory: As Pthreads has various interleavings of threads at runtime, it may result in different writing order to shared memory. Dthreads contains a special rule that only the thread which owns the token can write to shared memory. Hence, the expected writing order to shared memory can be guaranteed.

The sequence of synchronization operations: The sequence of synchronization operations in Pthreads may be different at runtime. However, as Dthreads adds a special token processing mechanism, the sequence in Dthreads is the same.

We now introduce an overview of Dthreads execution, as illustrated in Fig.1. The vertical lines represent the fences and the lines with arrows between them stand for threads. The states of fences switch between *Arrival Phase* and *Departure Phase*. Similarly, the states of threads also change between *Parallel Phase* and *Serial Phase*.

Here we use Thread 1 as an example to describe the whole process. First, Thread 1 is in *Parallel Phase* and the fence is in *Arrival Phase*. When it performs a synchronization operation, it blocks. That is to say, Thread 1 is stopped by the fence. As Thread 1 is the first one arriving at the fence, it gets the token. After Thread 2 and Thread 3 arrive at the fence, the fence's state changes to *Departure Phase* and Thread 1 will be woken up and its state is set to *Serial Phase*. Then, it finishes the synchronization operation and passes the token to Thread 2, which means it leaves the fence. Once Thread 2 and Thread 3 also leave the fence, Thread 1 and the fence are reset to *Parallel Phase* and *Arrival Phase* respectively. This cycle repeats.

2.2. A Brief Introduction of CSP

In this subsection, we give a short introduction to CSP (Communicating Sequential Processes) [29, 30]. It is a process algebra proposed by Hoare in 1978. As one of the most mature formal methods, it is tailored for describing the interaction between concurrency systems by mathematical theories. Because of its well-known expressive ability, CSP has been widely used in many fields [33, 34, 35, 36].

CSP processes are constituted by primitive processes and actions. We use the following syntax to define the processes in this paper, whereby P and Q represent processes, the alphabets $\alpha(P)$ and $\alpha(Q)$ mean the set of actions that the processes P and Q can take respectively, and a and b denote

DOI: 10.1002/smr

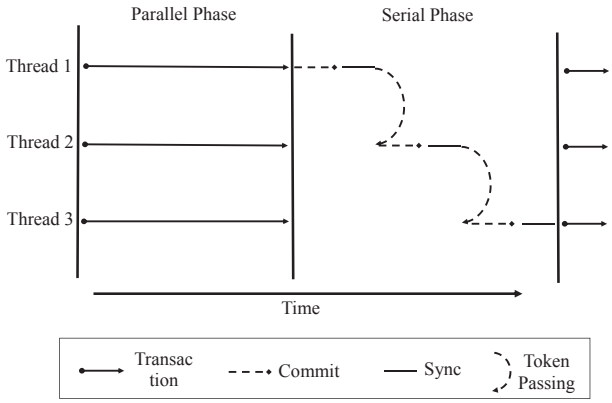


Figure 1. An overview of Dthreads execution (adapted from[1])

the atomic actions and c stands for the name of a channel.

$$P,Q = Skip \mid Stop \mid a \rightarrow P \mid \ c?x \rightarrow P \mid c!e \rightarrow P \mid P \Box Q \mid P \| Q \mid P|||Q \mid P \lhd b \rhd Q \mid P;Q$$

where:

- $Skip$ stands for a process which only terminates successfully.
- $Stop$ represents that the process does nothing and its state is deadlock.
- $a \rightarrow P$ first performs action a, then behaves like P.
- $c?x \rightarrow P$ receives a message by the channel c and assigns it to a variable x, then does the subsequent behavior like P.
- $c!\ e \rightarrow P$ sends a message e through the channel c, then performs P.
- $P \Box Q$ acts like either P or Q and the environment decides the selection.
- $P \parallel Q$ shows the parallel composition between P and Q.
- $P|||Q$ indicates the process chooses to perform actions in P and Q randomly.
- $P \lhd b \rhd Q$ denotes if the condition b is true, the process behaves like P, otherwise, like Q.
- $P;Q$ executes P and Q sequentially.

3. THREAD API AND EXAMPLE PROGRAMS

In this section, we first list the relevant interfaces of both Pthreads and Dthreads and then present four classical multi-threaded programs on which our comparative modelling and verification will be carried out.

3.1. API (Application Programming Interface)

We list five API functions that will be modelled. These API functions will be invoked by our illustrative programs.

- pthread_create()
- pthread_exit()
- pthread_join()
- pthread_mutex_lock ()

DOI: 10.1002/smr

- pthread_mutex_unlock ()

Their functions are for creating, terminating and joining threads, as well as locking and unlocking mutexes, respectively. Because Dthreads is a re-implementation of the multi-threading library Pthreads, it has the same API functions.

3.2. Example Programs

We choose one classical example program to reason about deadlocks and three example programs about the bank transformation process with respect to data races and race conditions, to be the scenarios of Pthreads and Dthreads. Here we show the main parts of the example programs. The main part of the example program for deadlocks is listed as below.

```
void *ThreadFunc1(void *arg){
  pthread_mutex_lock(&m0);
  pthread_mutex_lock(&m1);
  pthread_mutex_unlock(&m1);
  pthread_mutex_unlock(&m0);
  return NULL;
}

void *ThreadFunc2(void *arg){
  pthread_mutex_lock(&m1);
  pthread_mutex_lock(&m0);
  pthread_mutex_unlock(&m0);
  pthread_mutex_unlock(&m1);
  return NULL;
}
```

These two functions describe the tasks of two sub-threads. Thread 1 first locks mutex $m0$ and mutex $m1$, then unlocks mutex $m1$ and mutex $m0$. And thread 2 changes the order of locking and unlocking mutexes.

Now we describe how the threads cooperate in Dthreads briefly. First, we suppose that thread 1 and thread 2 have been created and the order of the threads arriving at the fence is thread 1, thread 2 and the main thread. It is also supposed that the current time is for the threads to leave the fence. Thread 1 first gets the token and does its first lock operation. According to the principle of Dthreads, the thread can only release the token to other threads until it owns no mutex. So thread 1 keeps on doing the following lock and unlock operations and then passes the token to thread 2. Thread 2 also does all the manipulation of mutex $m1$ and $m0$ and gives the token to the main thread. The main thread can not do the join operation because thread 1 and thread 2 are still running. As the three threads have no other non-synchronization operation, it can be considered that they have arrived at the fence. Again, thread 1 owns the token and does the exit operation and transmits the token to thread 2. Thread 2 also ends itself and delivers the token to the main thread. The main thread now can do the join operation of two subthreads and end the whole program.

The main part of the three example programs considering data races and race conditions are illustrated below. Initially, A and B are assigned to 100 and 0, respectively. We use some example cases in [37] with modification. The sub-threads of the three programs almost do the same things, expect for the statements about locking the mutex. If the balance of account A is not less than 100, the sub-thread moves 100 from account A to account B. Due to the manipulation of the mutex, the properties of data races and race conditions need to be considered, which we will discuss later.

```
void *ThreadFunc1(void *arg){
   if (A < 100){
     return NULL;
   }
   B = B + 100;
   A = A - 100;
   return NULL;
}

void *ThreadFunc2(void *arg){
   if (A < 100){
```


DOI: 10.1002/smr

```
      return NULL;
    }
    B = B + 100;
    A = A - 100;
    return NULL;
}
```

```
void *ThreadFunc1(void *arg){
  if (A < 100){
     return NULL;
  }
  pthread_mutex_lock(&th_lock);
  B = B + 100;
  pthread_mutex_unlock(&th_lock);
  pthread_mutex_lock(&th_lock);
  A = A - 100;
  pthread_mutex_unlock(&th_lock);
  return NULL;
}

void *ThreadFunc2(void *arg){
  if (A < 100){
     return NULL;
  }
  pthread_mutex_lock(&th_lock);
  B = B + 100;
  pthread_mutex_unlock(&th_lock);
  pthread_mutex_lock(&th_lock);
  A = A - 100;
  pthread_mutex_unlock(&th_lock);
  return NULL;
}
```

```
void *ThreadFunc1(void *arg){
  pthread_mutex_lock(&th_lock);
  if (A < 100){
     return NULL;
  }
  B = B + 100;
  A = A - 100;
  pthread_mutex_unlock(&th_lock);
  return NULL;
}

void *ThreadFunc2(void *arg){
  pthread_mutex_lock(&th_lock);
  if (A < 100){
     return NULL;
  }
  B = B + 100;
  A = A - 100;
  pthread_mutex_unlock(&th_lock);
  return NULL;
}
```

4. MODELING PTHREADS

We use CSP to model the illustrative programs and the thread API functions given in Section 3. For each illustrative program, the whole system is modelled as the parallel composition of four CSP processes, *Mutex*, *Controller*, *Program* and *Buffer*, which will be defined in what follows. The buffer component *Buffer* is used to store and manage data used in the system, for other components to access. For convenience, we only show three components in Fig. 2: *Program*, *Controller* and *Mutex*, omitting *Buffer*. Note that the process *Program* is used to model an illustrative program, *Controller*

DOI: 10.1002/smr

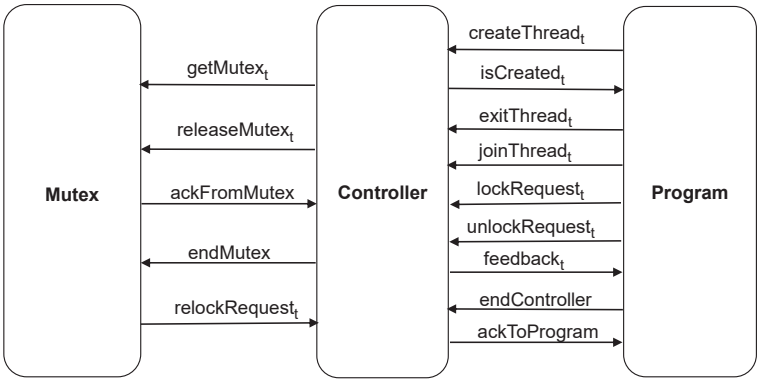


Figure 2. Interprocess Communication of the Model of Pthreads

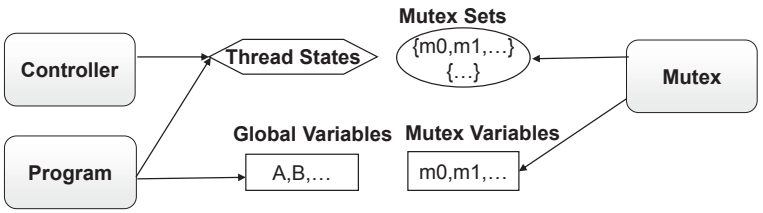


Figure 3. Data Communication of the Model of Pthreads

processes the requests from the synchronous operations of the threads, such as creating a new thread, locking a mutex, etc., and *Mutex* is especially responsible for handling mutexes according to the lock and unlock requests from *Controller*.

To better understand the process *Buffer*, we give the data communication graph of Pthreads model in Fig. 3. The thread states, mutex sets, mutex variables and global variables are all modelled by *Buffer*. They are used in *Program*, *Controller* and *Mutex* respectively.

$$SystemPthreads(scenario) =_{df} Mutex() \| Controller() \| Program(scenario) \| Buffer()$$

Note that the parameter *scenario* can be *deadlock*, *datarace* or *racecondition*, corresponding to the three illustrative programs respectively.

Table I. The Explanations of Channels in Pthreads Model

Channels	Functionalities
$(ack*)_t$	Receiving the acknowledgements
$(end*)_t$	Terminating the processes
$(*Mutex)_t$	Managing the mutexes between *Mutex* and *Controller*
$(*Thread)_t$	Manipulating the threads between *Program* and *Controller*
$(*Request)_t$	Managing the mutexes between *Controller* and *Program*
$feedback_t$	Receiving the feedback from *Controller* to *Program*
$isCreated_t$	Checking if the thread t is created or not from *Controller* to *Program*

Table I gives the explanations of channels in the Pthreads Model. Note that we use a uniform form to describe the channels with the same functionality. For example, $(ack*)_t$ represents the channels beginning with the word *"ack"* and their effects, including the channel *ackFromMutex* and the channel *ackToProgram*. Both of them are used to receive the acknowledgements from the senders to the receivers.

DOI: 10.1002/smr

4.1. The Program

In this subsection, we will model the three example programs described in Section 3. The process *Program* changes in different scenarios, and it is modelled as below.

$$
\begin{aligned}
Program(scenario) =_{df}\ & ProgramDeadLock(\ \triangleleft(scenario == deadlock)\triangleright \\
& (\ ProgramTransfer1 \triangleleft (scenario == transfer1)\triangleright \\
& (\ ProgramTransfer2 \triangleleft (scenario == transfer2) \triangleright ProgramTranfer3\ )\ ) \\
ProgramDeadLock =_{df}\ & (\ ||_{t:\{1..(T_n-1)\}} ThreadDeadLock_t\ ) \| MainThread; endController!yes \rightarrow \\
& ackToProgram?v \rightarrow Skip \\
ProgramTransfer1 =_{df}\ & (\ ||_{t:\{1..(T_n-1)\}} ThreadTransfer1_t\ ) \| MainThread; endController!yes \rightarrow \\
& ackToProgram?v \rightarrow Skip \\
ProgramTransfer2 =_{df}\ & (\ ||_{t:\{1..(T_n-1)\}} ThreadTransfer2_t\ ) \| MainThread; endController!yes \rightarrow \\
& ackToProgram?v \rightarrow Skip \\
ProgramTransfer3 =_{df}\ & (\ ||_{t:\{1..(T_n-1)\}} ThreadTransfer3_t\ ) \| MainThread; endController!yes \rightarrow \\
& ackToProgram?v \rightarrow Skip
\end{aligned}
$$

Note here the *ID* of main thread is assumed to be 0 and v denotes a variable. T_n denotes the maximum number of the threads at runtime and t represents a sub-thread *ID*.

We focus on *ProgramDeadlock*, *ProgramTransfer1*, *ProgramTransfer2* and *ProgramTransfer3*. All the subthreads interleave with each other and run in parallel with the main thread. At the end, each process notifies the process *Controller* that it has been completed and then ends itself.

Process *ThreadDeadLock*, process *ThreadTransfer1*, process *ThreadTransfer2* and process *ThreadTransfer3* correspond to the main parts of the illustrative programs in Section 3. In what follows we first present the core model for the program with respect to deadlocks (expressed in *ThreadDeadLock*). The two threads try to obtain the mutexes they need, and wait until they succeed. Note m is the *ID* of a mutex.

$$
\begin{aligned}
ThreadDeadLock_t =_{df}\ & isCreated_t?v \rightarrow (Lock_{t,t-1}; Lock_{t,2-t}; unLock_{t,2-t}; unLock_{t,t-1}; Exit_t) \\
Lock_{t,m} =_{df}\ & ThreadState_t?state \rightarrow (lockRequest_t!m \rightarrow LockSub_t \triangleleft (state == run) \triangleright Lock_{t,m}) \\
LockSub_t =_{df}\ & feedback_t?v \rightarrow (Skip \triangleleft (v == yes) \triangleright LockSub_t) \\
UnLock_{t,m} =_{df}\ & ThreadState_t?state \rightarrow (unlockRequest_t!m \rightarrow feedback_t?v \rightarrow Skip \\
& \triangleleft (state == run) \triangleright UnLock_{t,m})
\end{aligned}
$$

The process *ThreadTransfer1*, *ThreadTransfer2* and *ThreadTransfer3* given below represent the core model of the illustrative program with respect to the account transformation process. It modifies the value of two variables, according to different conditions. More specifically, *ThreadTransfer1* means the thread determines if the value of the global variable A is more than or equal to 100, the global variable A will be set to A-100 and B will be assigned to B+100 if so. *ThreadTransfer2* also does the same things but each assignment is protected by locking and unlocking the mutex M. *ThreadTransfer3* is similar with *ThreadTransfer1* as well. It adds the lock operation and the unlock operation at the front and rear of the assignments. Note that *val1* and *val2* denote variables, *cond* and *val* represent values in the program.

$$
\begin{aligned}
ThreadTransfer1_t =_{df}\ & isCreated_t?v \rightarrow WriteME_{100,A,A,100}; WriteME_{100,A,B,-100}; Exit_t \\
ThreadTransfer2_t =_{df}\ & isCreated_t?v \rightarrow Lock_{t,M}; WriteME_{100,A,A,100}; UnLock_{t,M}; \\
& Lock_{t,M}; WriteME_{100,A,B,-100}; UnLock_{t,M}; Exit_t \\
ThreadTransfer3_t =_{df}\ & isCreated_t?v \rightarrow Lock_{t,M}; WriteME_{100,A,A,100}; WriteME_{100,A,B,-100}; UnLock_{t,M}; Exit_t \\
WriteME_{cond,var1,var2,val} =_{df}\ & getValue_{var1}?data1 \rightarrow getValue_{var2}?data2 \rightarrow \\
& (setValue_{var2}!(data2+val) \triangleleft (data1 >= cond) \triangleright Skip)
\end{aligned}
$$

DOI: 10.1002/smr

4.2. The Mutex

Process *Mutex* describes the manipulation of mutexes, i.e., handling lock and unlock requests from process *Controller*. In addition, it also modifies information about the mutexes. The mutexes are abstracted using process *Mutex*, The module of process *Mutex* is shown as below:

$$\begin{aligned} Mutex =_{df}\ & (\Box_{t\in 1..(T_n-1)} MLock_t) \Box\ (\Box_{t\in 1..(T_n-1)} MUnLock_t) \\ & \Box\ (endMutex?v \rightarrow ackFromMutex!yes \rightarrow Skip) \end{aligned}$$

Here, t and T_n are defined to be the thread *ID* and the maximal number of threads. Process $MLock_t$ and process $MUnLock_t$ handle the lock requests and unlock requests from thread t respectively. The last part is to end this module and synchronize with the whole system.

For the process *MLock*, it first gets the message m from the channel $getMutex_t$, which means $thread_t$ requests for mutex m. In addition, competitions between threads awaiting the same mutex lead to the impossibility to predicate which one will get the mutex. As a result, we use set instead of queue to store threads' *ID*s waiting for the mutex. The thread *ID t* will be put into the set of the mutex m and the size of the set will be increased.

$$MLock_t =_{df}\ getMutex_t?m \rightarrow Mutex_m?state \rightarrow \left(\begin{array}{l} (Mutex_m!busy \rightarrow ackFromMutex!yes \rightarrow Skip) \\ \lhd (state == idle) \rhd \\ (MutexSetIn_m!t \rightarrow MutexSetSize_m!Increase \rightarrow \\ ackFromMutex!no \rightarrow Skip) \end{array} \right); Mutex$$

For the process $MUnLock_t$, it first receives the message m from the channel $releaseMutex_t$, which means thread t releases the mutex m. Therefore, thread *ID t* needs to be removed from the set and the size of the set should be decreased. Furthermore, it must check if there are still any threads waiting for the mutex m. If so, one element of the set is taken out and the thread is allowed to request for the mutex m again. The process is illustrated as below.

$$\begin{aligned} MUnLock_t =_{df}\ & releaseMutex_t?m \rightarrow Mutex_m!idle \rightarrow MutexSetSize_m?size \rightarrow \\ & \left(\begin{array}{l} (\ MutexSetOut_m?t \rightarrow MutexSetSize_m!Decrease \\ \rightarrow RelockRequest_t!m \rightarrow Skip\ ) \lhd (size > 0) \rhd Skip \end{array} \right); \\ & ackFromMutex!yes \rightarrow Skip; Mutex \end{aligned}$$

4.3. The Controller

We use the process *Controller* to handle the messages from the process *Program* and the process *Mutex*. Its purpose is to create threads, terminate threads, join threads, lock and unlock mutexes.

$$\begin{aligned} Controller =_{df}\ & (\Box_{t\in 1..(Tn-1)} CLock_t) \Box\ (\Box_{t\in 1..(Tn-1)} CUnLock_t) \\ & \Box\ (\Box_{t\in 1..(Tn-1)} CReLock_t) \Box\ (\Box_{t\in 1..(Tn-1)} CCreateThread_t) \\ & \Box\ (\Box_{t\in 1..(Tn-1)} CExitThread_t) \Box\ (\Box_{t\in 1..(Tn-1)} CJoinThread_t) \\ & \Box\ EndController \end{aligned}$$

We illustrate the first two sub-processes below. By mapping to the model, the action that a thread tries to lock a mutex signifies that the process *Program* sends a lock or unlock request to the process *Controller*. Hence, the sub-processes $CLock_t$ and $CUnLock_t$ in *Controller* depict how to deal with the requests.

First, *Controller* informs *Mutex* that thread t needs to lock or unlock mutex m. When *Mutex* replies with a failure or success feedback, *Controller* just sends it directly to *Program* without any change.

$$CLock_t =_{df}\ lockRequest_t?m \rightarrow getMutex_t!m \rightarrow ackFromMutex?v \rightarrow feedback_t!v \rightarrow Skip; Controller$$

$$CUnLock_t =_{df}\ unlockRequest_t?m \rightarrow releaseRequest_t!m \rightarrow ackFromMutex?v \rightarrow feedback_t!v \rightarrow Skip; Controller$$

DOI: 10.1002/smr

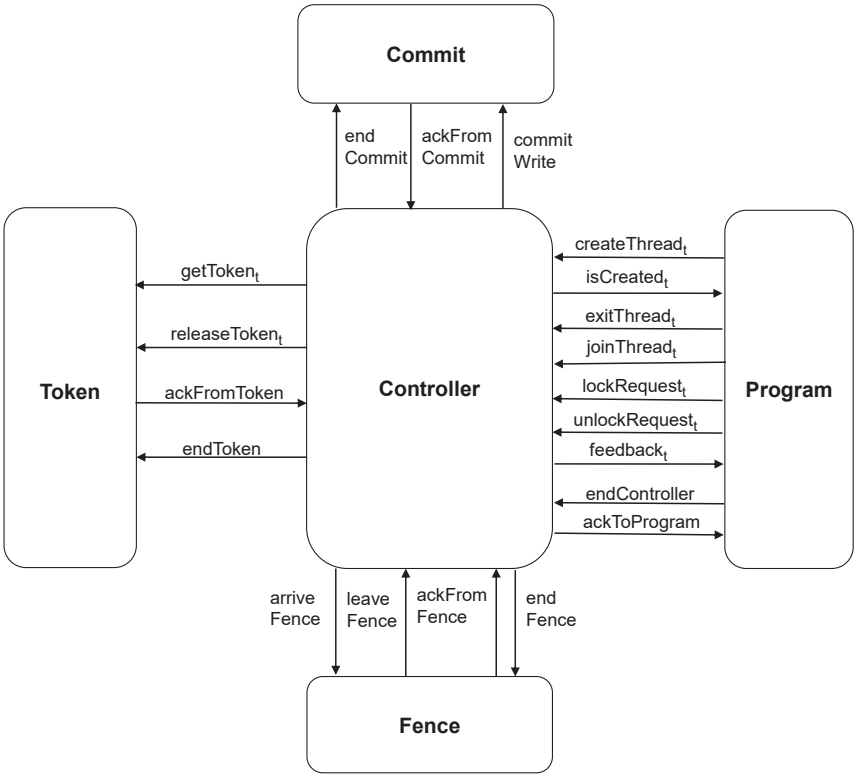


Figure 4. Interprocess Communication of the Model of Dthreads

5. MODELING DTHREADS

As mentioned in Section 2, Dthreads has some specific mechanisms (for fence and commit). For the formal modelling, we introduce two more modules *Fence* and *Commit* for the fence and the special commit mechanism. We also list the five components in Fig. 4, omitting *Buffer*. They are *Program*, *Controller*, *Token*, *Commit* and *Fence*. In Dthreads, the token replaces all the mutexes for lock and unlock requests. Hence, here *Token* replaces *Mutex* compared with the Pthreads model. Due to the fence and special commit steps, the process *Commit* modifies memory contents and decides when to commit the update to (shared) memory. The process *Fence* simulates the fence mechanism in Dthreads, which controls the running of threads. As a result, the whole system is modelled as follows.

$$SystemDthreads(scenario) =_{df} \ Token \,||\, Controller \,||\, Commit \,||\, Fence \,||\, Program(scenario) \,||\, Buffer$$

Fig. 5 provides the data used by the components of the Dthreads model except for *Buffer*. We relate the processes to the data by the lines with arrow. The correlated data include waiting threads number, living threads number, thread states, fence phase, token queue, lock count, token state and global variables. They are modelled by *Buffer*.

Table II provides the explanation for channels in Fig. 4. Note that there are some channels in both of our Dthreads Model and Pthreads Model, i.e., having the same name (e.g., $feedback_t$). Their functionalities are the same, but the messages in them are in a different form. We do not list them in Table II.

5.1. The Program

The components in the process *Program*, such as *Lock* and *UnLock*, are fundamentally the same as those in Pthreads, except for *WriteME*. As mentioned earlier, Dthreads re-implements the Pthreads. An important difference between Pthreads and Dthreads is the way they treat updates. In Pthreads, the write operation makes direct modification to the (shared) memory. However, in Dthreads, once a thread tries to do the write operation, if it is the owner of the token, the thread is allowed to update

DOI: 10.1002/smr

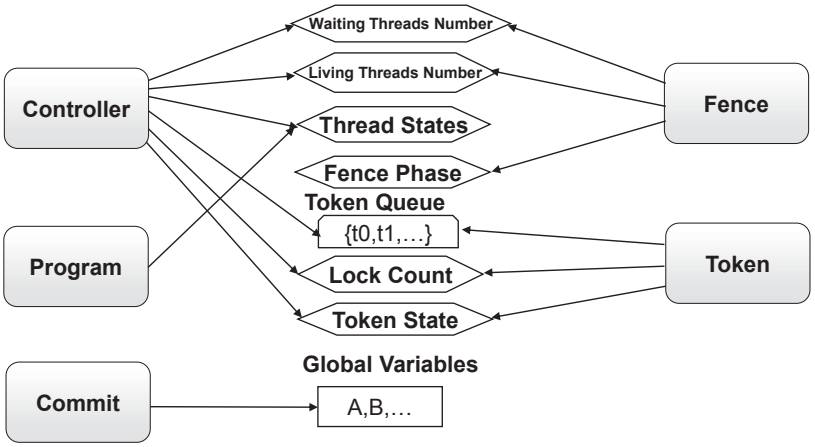


Figure 5. Data Communication of the Model of Dthreads

Table II. The Explanations of Channels in Dthreads Model

Channels	Functionalities
$(*Token)_t$	Managing the token between *Token* and *Controller*
$(*Fence)_t$	Changing the states of the fences between *Fence* and *Controller*
commitWrite	Committing the update to memory from *Controller* to *Commit*

the memory by using *SetValue* channel. Otherwise, it just updates its own private copy by sending a message to *SetPrivateValue* channel. The write operation in Dthreads can be formally defined as below. Note that t is for thread ID, var, $val1$ and $var2$ denote the variables, while $cond$, val, $val2$ and val represent the data used in the program.

$$\begin{aligned} WriteME_{t,cond,var1,var2,val} =_{df}\ & getValue_{var1}?data1 \rightarrow getValue_{var2}?data2 \rightarrow \\ & (Modify_{t,var2,val} \lhd (data1 >= cond) \rhd Skip) \end{aligned}$$

$$\begin{aligned} Modify_{t,var,val} =_{df}\ & TokenOwner?t' \rightarrow (SetValue_{var}!val \rightarrow Skip \lhd (t' == t) \rhd \\ & SetPrivateValue_{t,var}!val \rightarrow isWrite_t!yes \rightarrow Skip) \end{aligned}$$

5.2. The Token

When dealing with the mutex in Pthreads, we use a set to store the thread *ID* for each mutex. However, in Dthreads, we need to modify the state of the token, maintain the token queue, and record the number of locks each thread owns to achieve Dthreads' special mechanism. Our modelling is shown as below.

$$\begin{aligned} Token =_{df}\ & (\Box_{t\in 1..(T_n-1)} GetToken_t) \Box (\Box_{t\in 1..(T_n-1)} ReleaseToken_t) \\ & \Box\ (\ endToken?v \rightarrow ackFromToken!yes \rightarrow Skip\ ) \end{aligned}$$

Here, *t* is the *ID* of a sub-thread. If *Token* receives the acquire and release requests from *Controller*, $GetToken_t$ and $ReleaseToken_t$ will deal with them respectively. When the whole system terminates,

DOI: 10.1002/smr

the last part will end this module. We define the $GetToken_t$ and $ReleaseToken_t$ as below.

$$\begin{aligned} GetToken_t =_{df}\ & GetToken_t?m \rightarrow Token?state \rightarrow (\,(\,Token!busy \rightarrow TokenOwner!t \rightarrow \\ & ackFromToken!yes \rightarrow Token\,) \lhd (state == idle) \rhd (\,TokenQue_t!Enqueue \rightarrow \\ & TokenQueLength!Increase \rightarrow ackFromToken!no \rightarrow Token\,)\,) \end{aligned}$$

$$\begin{aligned} ReleaseToken_t =_{df}\ & releaseToken_t?v \rightarrow LockCount_t?num \rightarrow \\ & \left(\begin{array}{l} (\,Token!idle \rightarrow TokenOwner!none \rightarrow ackFromToken!yes \rightarrow Token\,) \\ \lhd (num==0) \rhd (\,LockCount_t!Decrease \rightarrow LockCount_t?num \rightarrow \\ (\,(\,Token!idle \rightarrow TokenOwner!none \rightarrow ackFromToken!yes \rightarrow Token\,) \\ \lhd (num==0) \rhd (ackFromToken!no \rightarrow Token\,)\,) \end{array}\right) \end{aligned}$$

When a token acquiring request comes, $GetToken_t$ checks the state of the token. If the state is *idle*, it sets the state to *busy*, declares the token owner to be the thread t and replies *Controller* with a positive feedback. Otherwise, the thread is inserted into the token queue and $GetToken_t$ feeds *Controller* with a negative feedback.

$ReleaseToken_t$ is a little different from $GetToken_t$. First of all, $ReleaseToken_t$ judges the number of locks that the thread owns. If the number is zero, it sets the state of the token to *idle*, clears the token owner, and sends a success feedback to *Controller*. Otherwise, it decreases the lock count and checks if thread *t* holds any lock. If so, a failure feedback is sent to *Controller* or it just does the same things as when the number is zero.

5.3. The Commit

The process *Commit* takes charge of managing updates to the memory. When it receives the request of committing memory, *Commit* first determines if the private copy has changed. If so, it updates the copy to shared memory. Finally, *Commit* gives a success feedback to *Controller*. Note that t denotes the sub-thread *ID* and *VariableSet* denotes the variables used in the program.

$$\begin{aligned} Commit =_{df}\ & \Box_{t\in 1..(T_n-1), v\in VariableSet}\ commitWrite_t?v \rightarrow commitWrite_{t,v} \\ CommitWrite_{t,v} =_{df}\ & isWrite_t?state \rightarrow (\,(\,GetPrivateValue_{t,v}?d \rightarrow SetValue_v!d \rightarrow isWrite_t!no \rightarrow \\ & Skip\,) \lhd (state == yes) \rhd Skip\,); ackFromCommit!yes \rightarrow Commit \end{aligned}$$

5.4. The Controller

As Dthreads has a particular mechanism of committing updates and controlling fences, *Controller* here is more complex than the one in the Pthreads model. *Controller* is composed of *ParallelPhase* and *SerialPhase*.

$$Controller =_{df}\ fencePhase?phase \rightarrow (ParallelPhase \lhd (phase == Arrival) \rhd SerialPhase)$$

As mentioned in Section 2, when a thread tries to do a synchronization action, it is blocked by the fence. During this period, all the threads are in the *Parallel Phase*. We need to record these actions in the process *ParallelPhase*, as they will be executed when the threads leave the fence. *ParallelPhase* describes the situation when the fence is in the *Arrival Phase*, and all the threads are in the *Parallel Phase*, which is modelled as below.

$$\begin{aligned} ParallelPhase =_{df}\ & (\Box_{t\in 1..(T_n-1)} PPLock_t) \Box\ (\Box_{t\in 1..(T_n-1)} PPUnLock_t) \\ & \Box\ (\Box_{t\in 1..(T_n-1)} PPCreate_t) \Box\ (\Box_{t\in 1..(T_n-1)} PPJoin_t) \\ & \Box\ EndController \end{aligned}$$

The first four parts in *ParallelPhase* record different synchronization actions of threads and tell the fence that the thread has arrived. The last part in *ParallelPhase* is the end part. That is, when the system receives the message of ending from the outside, it automatically ends itself. $PPLock_t$ is listed below as an example to illustrate the main function of *ParallelPhase*.

DOI: 10.1002/smr

$$PPLock_t =_{df} \; lockRequest_t?m \rightarrow SetMutexVar_t!m \rightarrow ThreadState_t!lock \rightarrow arriveFence!t \rightarrow Controller$$

The process *SerialPhase* is more complicated. It handles the situation when the fence is in the *Departure Phase*. It takes the first *ID* of threads from the token queue, which means releasing the blocked threads, and executing the pending synchronized actions. According to the records of synchronized actions, the blocked thread continues its unfinished action. If the action is to lock the mutex, $SPLock_t$ will handle it. If the action is to unlock the mutex, $SPUnLock_t$ will tackle it. If the action tries to create a thread, $SPCreate_t$ will deal with it. If the action wants to join two threads, $SPJoin_t$ will cope with it. When the action tries to kill the process, $SPExit_t$ will manage it.

$$\begin{aligned} SerialPhase =_{df} \; & leaveFence?v \rightarrow TokenQueDequeue?t \rightarrow TokenQueLength!Decrease \rightarrow \\ & ThreadState_t?state \rightarrow (SPLock_t \lhd (state == lock) \rhd \\ & (\, SPUnLock_t \lhd (state == unlock) \rhd (\, SPCreate_t \lhd (state == create) \rhd \\ & (\, SPJoin_t \lhd (state == join) \rhd (\, SPExit_t \lhd (state == exit) \rhd Contoller)\,)\,)\,) \end{aligned}$$

Let us take $SPLock_t$ as an example to make the explanation. $SPLock_t$ illustrates the way how *Controller* deals with the requests for the mutex from *Program*. It first checks if the lock count is zero. If it is zero, thread *t* is able to ask for the token. $AcquireToken_t$ will keep on asking for the token until it gets one. And thread *t* updates its private copy to the shared memory after owning the token. Besides, the lock count is increased and the subsequent management is handled by $SPLockSub_t$. As explained earlier, Dthreads uses the lock count to guarantee that a thread keeps on holding the token until it gives up all the locks it owns. Consequently, the job of $SPLockSub_t$ is to manage the following lock and unlock requests from *Program*. They are modelled as below.

$$\begin{aligned} SPLock_t =_{df} \; & LockCount_t?num \rightarrow (\, AcquireToken_t; commitWrite!t \rightarrow \\ & ackFromCommit?v \rightarrow Skip \lhd (num == 0) \rhd Skip\,); \\ & LockCount_t!Increase \rightarrow feedback_t!yes \rightarrow SPLockSub_t; \\ AcquireToken_t =_{df} \; & getToken!t \rightarrow ackFromToken?v \rightarrow (\, Skip \lhd (v == yes) \rhd AcquireToken_t); \\ SPLockSub_t =_{df} \; & (\, lockRequest?t \rightarrow SPLock_t\,) \\ & \Box \left(\begin{array}{l} unlockRequest?t \rightarrow releaseToken!t \rightarrow ackFromToken?v \rightarrow \\ \left(\begin{array}{l} \left(\begin{array}{l} commitWrite!t \rightarrow ackFromCommit?v \rightarrow \\ TokenQueEnqueue!t \rightarrow TokenQueLength!Increase \rightarrow \end{array} \right) \\ feedback_t!yes \rightarrow Controller \end{array} \right) \\ \lhd (v == yes) \rhd \\ (feedback_t!no \rightarrow SPLockSub_t) \end{array} \right) \end{aligned}$$

5.5. The Fence

The process *Fence* simulates the fence mechanism in Dthreads. Two processes *ArrivalPhase* and *DeparturePhase* constitute *Fence* as shown below.

$$\begin{aligned} Fence =_{df} \; & (\, fencePhase?state \rightarrow (\, ArrivalPhase \lhd (state == Arrival) \rhd DeparturePhase\,) \\ & \Box \; endFence?v \rightarrow ackFromToken?v \rightarrow Skip\,) \end{aligned}$$

The fence first checks which phase it is in. If it is in the *Arrival Phase*, it increases the waiting thread number when a thread arrives. Once all the living threads have arrived, the fence is set to the *Departure Phase*, and releases all the threads blocked by itself. After that, the fence is reset to the *Arrival Phase*. These are defined below.

$$\begin{aligned} ArrivalPhase =_{df} \; & arriveFence?t \rightarrow waitingThread!Increase \rightarrow waitingThreadNum?n \rightarrow \\ & livingThreadNum?n' \rightarrow (\, fencePhase!Departure \rightarrow Skip \lhd (n == n') \rhd Skip\,); Fence \\ DeparturePhase =_{df} \; & waitingThread!Decrease \rightarrow leaveFence!yes \rightarrow waitingThreadNum?n \rightarrow \\ & (\, fencePhase!Arrival \rightarrow Skip \lhd (n == 0) \rhd Skip\,); Fence \end{aligned}$$

DOI: 10.1002/smr

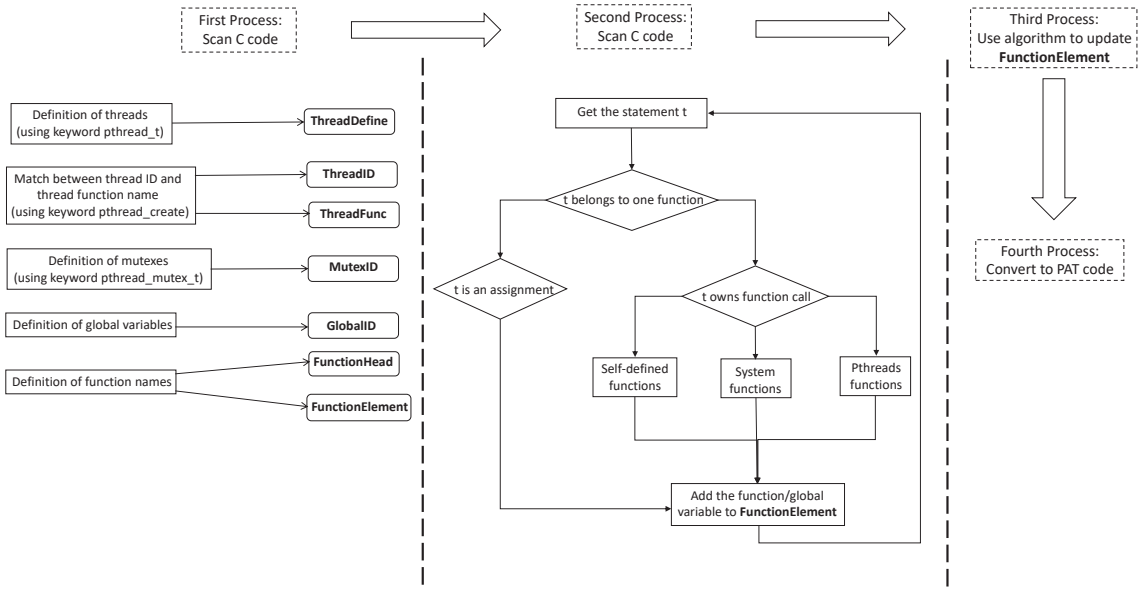


Figure 6. Process of conversion from C code to PAT code

6. CONVERSION

In order to complete our verification, a conversion from C code to PAT code is needed. In [2], we just do this manually. Now, we introduce a new method which can transfer C code to PAT code automatically. It is coded in Java. The key point of conversion is how to analyse the context of C code. Here we give the process of conversion in Fig. 6. In the first scan, we find all the definitions of threads and put them into string list *ThreadDefine*. The statements containing function *pthread_create* are analysed to record the matching between the thread functions and the thread IDs for future use. The calling of *pthread_mutex_lock* and *pthread_mutex_unlock* need parameters of mutex IDs, so we use the key word *pthread_mutex_t* to extract mutex IDs to *MutexID*. Global variables are also recorded in *GlobalID*. Moreover, we detect all the function names and add them to *FunctionHead* and *FunctionElement*.

In the second scan, we analyse every statement belonging to one function, to check if it is an assignment and if it owns a function call. Each global variable, appearing in left hand side of the assignment operator, is recorded in the responding element in *FunctionElement*. The functions can be divided into self-defined functions, Pthreads functions and system functions. As system functions, such as *printf()*, often not affect the global variables, we do not discuss these here. Self-defined functions and Pthreads functions are treated specially. All function calls are also noted in *FunctionElement*.

We introduce a new method to deal with a more complex situation. As we translate all the elements in one function to the element in *FunctionElement*, it is possible that the element is a self-defined function call, which means the function calls another function. We can not tell if it modifies any global variable or not. If we can get the subelements of this function, we can figure out if it has affected the global variables. Consequently, a method is needed to handle the expansion of functions appearing in each element in *FunctionElement*, illustrated in Fig. 7. Expansion here means to make every subelement of the element in *FunctionElement* to be a global variable or a Pthreads function without self-defined functions.

We will traverse all the elements in *FunctionElement*. As the subelements (global variables or function calls) are separated by the symbol |, they are retrieved and inserted to a temp stack called *TmpStack*. Then the following situation is treated by function *Expand()* and its subfunction *ExpandSub()*.

DOI: 10.1002/smr

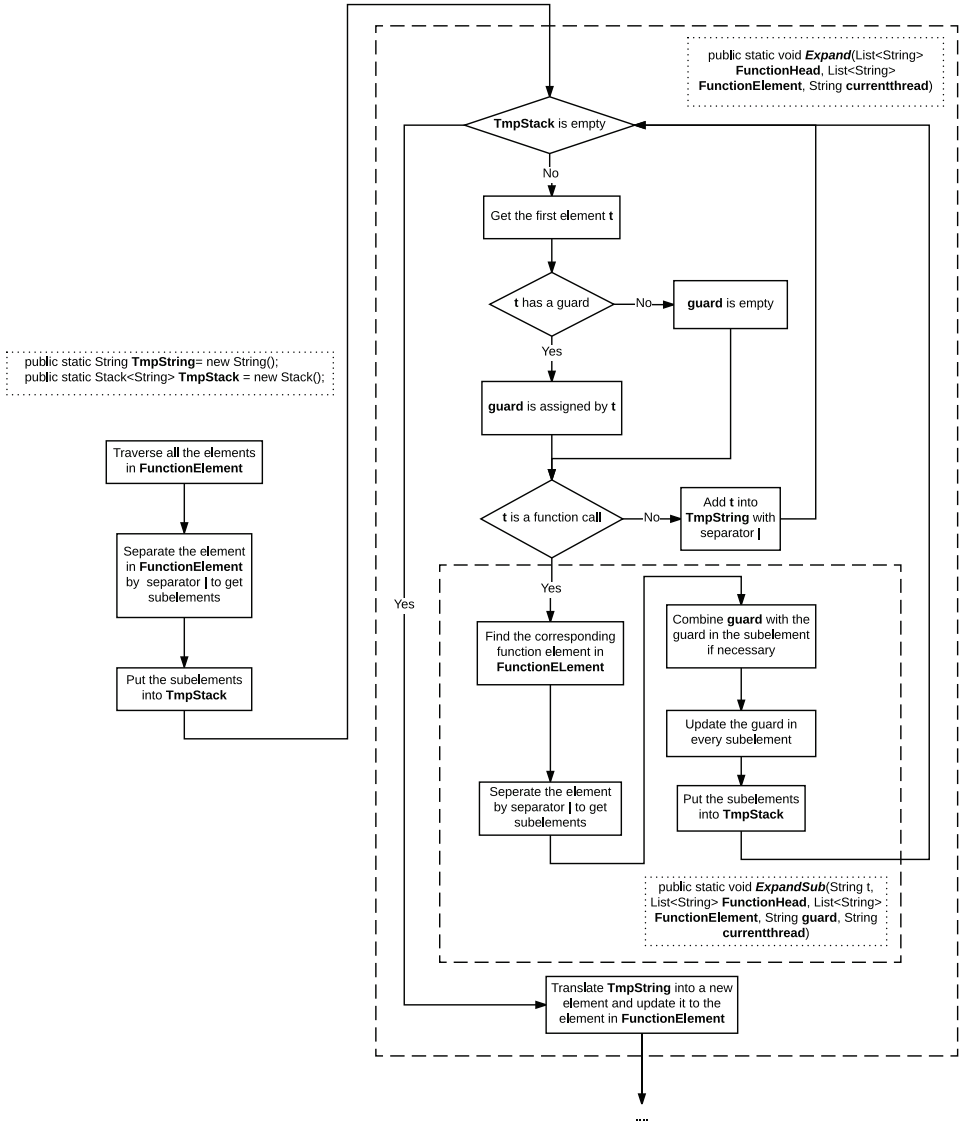


Figure 7. Method to expand the functions appearing in each element in *FunctionElement*

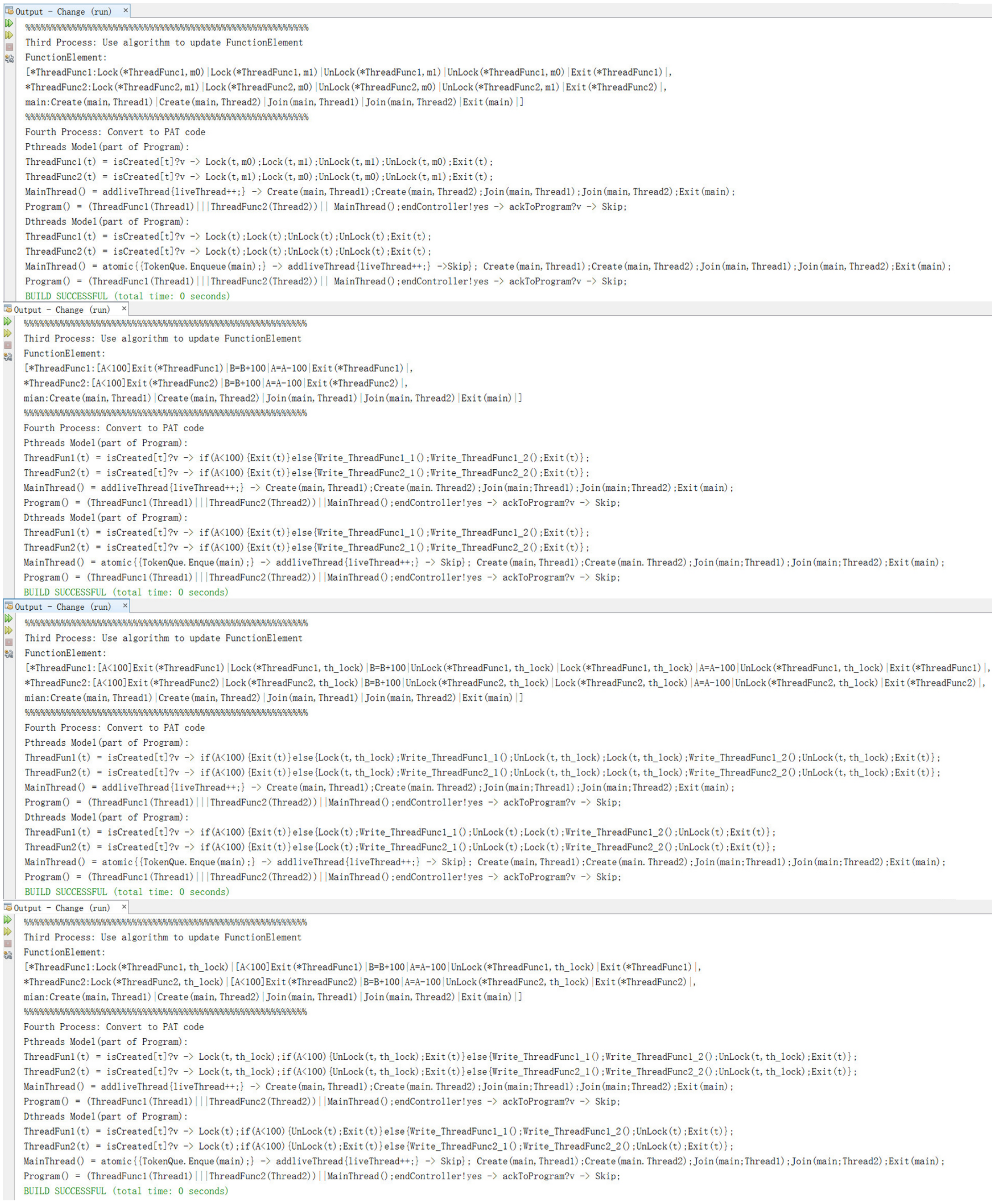
We get the first element *t* out of *TmpStack* and use *guard* to record the guard in *t*. Then *t* is checked if it is a function call. If so, we get the responding function element in *FunctionElement*, and take all the subelements by the separator |. Then the combination between the variable *guard* and the guard in subelement will be done if necessary. Finally, we put them into *TmpStack* for future checking. If *t* is not a function call, it is inserted directly to *TmpString*. When there is no element in *TmpStack*, *TmpString* will be translated into a new element and updated it to *FunctionElement*.

The last process is to build PAT code. We use the elements in *FunctionElement* to build the part of *Program* in our CSP model. Fig. 8 shows the processing procedure of changing the C code of the four example programs in Section 3.2.

7. VERIFICATION

Liu *et al.* [1] raised the point that Dthreads can almost eliminate data races and prevent deadlocks, whereas Pthreads can not. That is to say, for majority programs which cause data races and

Prepared using smrauth.cls
DOI: 10.1002/smr



```
Output - Change (run)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
Third Process: Use algorithm to update FunctionElement
FunctionElement:
[*ThreadFunc1:Lock(*ThreadFunc1,m0)|Lock(*ThreadFunc1,m1)|UnLock(*ThreadFunc1,m1)|UnLock(*ThreadFunc1,m0)|Exit(*ThreadFunc1)|,
*ThreadFunc2:Lock(*ThreadFunc2,m1)|Lock(*ThreadFunc2,m0)|UnLock(*ThreadFunc2,m0)|UnLock(*ThreadFunc2,m1)|Exit(*ThreadFunc2)|,
main:Create(main,Thread1)|Create(main,Thread2)|Join(main,Thread1)|Join(main,Thread2)|Exit(main)|]
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
Fourth Process: Convert to PAT code
Pthreads Model(part of Program):
ThreadFunc1(t) = isCreated[t]?v -> Lock(t,m0);Lock(t,m1);UnLock(t,m1);UnLock(t,m0);Exit(t);
ThreadFunc2(t) = isCreated[t]?v -> Lock(t,m1);Lock(t,m0);UnLock(t,m0);UnLock(t,m1);Exit(t);
MainThread() = addliveThread{liveThread++;} -> Create(main,Thread1);Create(main,Thread2);Join(main,Thread1);Join(main,Thread2);Exit(main);
Program() = (ThreadFunc1(Thread1)|||ThreadFunc2(Thread2))|| MainThread();endController!yes -> ackToProgram?v -> Skip;
Dthreads Model(part of Program):
ThreadFunc1(t) = isCreated[t]?v -> Lock(t);Lock(t);UnLock(t);UnLock(t);Exit(t);
ThreadFunc2(t) = isCreated[t]?v -> Lock(t);Lock(t);UnLock(t);UnLock(t);Exit(t);
MainThread() = atomic{{TokenQue.Enqueue(main);} -> addliveThread{liveThread++;} ->Skip}; Create(main,Thread1);Create(main,Thread2);Join(main,Thread1);Join(main,Thread2);Exit(main);
Program() = (ThreadFunc1(Thread1)|||ThreadFunc2(Thread2))|| MainThread();endController!yes -> ackToProgram?v -> Skip;
BUILD SUCCESSFUL (total time: 0 seconds)
Output - Change (run)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
Third Process: Use algorithm to update FunctionElement
FunctionElement:
[*ThreadFunc1:[A<100]Exit(*ThreadFunc1)|B=B+100|A=A-100|Exit(*ThreadFunc1)|,
*ThreadFunc2:[A<100]Exit(*ThreadFunc2)|B=B+100|A=A-100|Exit(*ThreadFunc2)|,
mian:Create(main,Thread1)|Create(main,Thread2)|Join(main,Thread1)|Join(main,Thread2)|Exit(main)|]
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
Fourth Process: Convert to PAT code
Pthreads Model(part of Program):
ThreadFun1(t) = isCreated[t]?v -> if(A<100){Exit(t)}else{Write_ThreadFunc1_1();Write_ThreadFunc1_2();Exit(t)};
ThreadFun2(t) = isCreated[t]?v -> if(A<100){Exit(t)}else{Write_ThreadFunc2_1();Write_ThreadFunc2_2();Exit(t)};
MainThread() = addliveThread{liveThread++;} -> Create(main,Thread1);Create(main.Thread2);Join(main;Thread1);Join(main;Thread2);Exit(main);
Program() = (ThreadFunc1(Thread1)|||ThreadFunc2(Thread2))||MainThread();endController!yes -> ackToProgram?v -> Skip;
Dthreads Model(part of Program):
ThreadFun1(t) = isCreated[t]?v -> if(A<100){Exit(t)}else{Write_ThreadFunc1_1();Write_ThreadFunc1_2();Exit(t)};
ThreadFun2(t) = isCreated[t]?v -> if(A<100){Exit(t)}else{Write_ThreadFunc2_1();Write_ThreadFunc2_2();Exit(t)};
MainThread() = atomic{{TokenQue.Enque(main);} -> addliveThread{liveThread++;} -> Skip}; Create(main,Thread1);Create(main.Thread2);Join(main;Thread1);Join(main;Thread2);Exit(main);
Program() = (ThreadFunc1(Thread1)|||ThreadFunc2(Thread2))||MainThread();endController!yes -> ackToProgram?v -> Skip;
BUILD SUCCESSFUL (total time: 0 seconds)
Output - Change (run)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
Third Process: Use algorithm to update FunctionElement
FunctionElement:
[*ThreadFunc1:[A<100]Exit(*ThreadFunc1)|Lock(*ThreadFunc1,th_lock)|B=B+100|UnLock(*ThreadFunc1,th_lock)|Lock(*ThreadFunc1,th_lock)|A=A-100|UnLock(*ThreadFunc1,th_lock)|Exit(*ThreadFunc1)|,
*ThreadFunc2:[A<100]Exit(*ThreadFunc2)|Lock(*ThreadFunc2,th_lock)|B=B+100|UnLock(*ThreadFunc2,th_lock)|Lock(*ThreadFunc2,th_lock)|A=A-100|UnLock(*ThreadFunc2,th_lock)|Exit(*ThreadFunc2)|,
mian:Create(main,Thread1)|Create(main,Thread2)|Join(main,Thread1)|Join(main,Thread2)|Exit(main)|]
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
Fourth Process: Convert to PAT code
Pthreads Model(part of Program):
ThreadFun1(t) = isCreated[t]?v -> if(A<100){Exit(t)}else{Lock(t,th_lock);Write_ThreadFunc1_1();UnLock(t,th_lock);Lock(t,th_lock);Write_ThreadFunc1_2();UnLock(t,th_lock);Exit(t)};
ThreadFun2(t) = isCreated[t]?v -> if(A<100){Exit(t)}else{Lock(t,th_lock);Write_ThreadFunc2_1();UnLock(t,th_lock);Lock(t,th_lock);Write_ThreadFunc2_2();UnLock(t,th_lock);Exit(t)};
MainThread() = addliveThread{liveThread++;} -> Create(main,Thread1);Create(main.Thread2);Join(main;Thread1);Join(main;Thread2);Exit(main);
Program() = (ThreadFunc1(Thread1)|||ThreadFunc2(Thread2))||MainThread();endController!yes -> ackToProgram?v -> Skip;
Dthreads Model(part of Program):
ThreadFun1(t) = isCreated[t]?v -> if(A<100){Exit(t)}else{Lock(t);Write_ThreadFunc1_1();UnLock(t);Lock(t);Write_ThreadFunc1_2();UnLock(t);Exit(t)};
ThreadFun2(t) = isCreated[t]?v -> if(A<100){Exit(t)}else{Lock(t);Write_ThreadFunc2_1();UnLock(t);Lock(t);Write_ThreadFunc2_2();UnLock(t);Exit(t)};
MainThread() = atomic{{TokenQue.Enque(main);} -> addliveThread{liveThread++;} -> Skip}; Create(main,Thread1);Create(main.Thread2);Join(main;Thread1);Join(main;Thread2);Exit(main);
Program() = (ThreadFunc1(Thread1)|||ThreadFunc2(Thread2))||MainThread();endController!yes -> ackToProgram?v -> Skip;
BUILD SUCCESSFUL (total time: 0 seconds)
Output - Change (run)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
Third Process: Use algorithm to update FunctionElement
FunctionElement:
[*ThreadFunc1:Lock(*ThreadFunc1,th_lock)|[A<100]Exit(*ThreadFunc1)|B=B+100|A=A-100|UnLock(*ThreadFunc1,th_lock)|Exit(*ThreadFunc1)|,
*ThreadFunc2:Lock(*ThreadFunc2,th_lock)|[A<100]Exit(*ThreadFunc2)|B=B+100|A=A-100|UnLock(*ThreadFunc2,th_lock)|Exit(*ThreadFunc2)|,
mian:Create(main,Thread1)|Create(main,Thread2)|Join(main,Thread1)|Join(main,Thread2)|Exit(main)|]
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
Fourth Process: Convert to PAT code
Pthreads Model(part of Program):
ThreadFun1(t) = isCreated[t]?v -> Lock(t,th_lock);if(A<100){UnLock(t,th_lock);Exit(t)}else{Write_ThreadFunc1_1();Write_ThreadFunc1_2();UnLock(t,th_lock);Exit(t)};
ThreadFun2(t) = isCreated[t]?v -> Lock(t,th_lock);if(A<100){UnLock(t,th_lock);Exit(t)}else{Write_ThreadFunc2_1();Write_ThreadFunc2_2();UnLock(t,th_lock);Exit(t)};
MainThread() = addliveThread{liveThread++;} -> Create(main,Thread1);Create(main.Thread2);Join(main;Thread1);Join(main;Thread2);Exit(main);
Program() = (ThreadFunc1(Thread1)|||ThreadFunc2(Thread2))||MainThread();endController!yes -> ackToProgram?v -> Skip;
Dthreads Model(part of Program):
ThreadFun1(t) = isCreated[t]?v -> Lock(t);if(A<100){UnLock(t);Exit(t)}else{Write_ThreadFunc1_1();Write_ThreadFunc1_2();UnLock(t);Exit(t)};
ThreadFun2(t) = isCreated[t]?v -> Lock(t);if(A<100){UnLock(t);Exit(t)}else{Write_ThreadFunc2_1();Write_ThreadFunc2_2();UnLock(t);Exit(t)};
MainThread() = atomic{{TokenQue.Enque(main);} -> addliveThread{liveThread++;} -> Skip}; Create(main,Thread1);Create(main.Thread2);Join(main;Thread1);Join(main;Thread2);Exit(main);
Program() = (ThreadFunc1(Thread1)|||ThreadFunc2(Thread2))||MainThread();endController!yes -> ackToProgram?v -> Skip;
BUILD SUCCESSFUL (total time: 0 seconds)
```

Figure 8. Processing procedure of changing the C code of the example programs

deadlocks in Pthreads, Dthreads can eliminate the bad situations. As explained in the introduction, race conditions are important concurrent issues as well. In the following subsections, we show how we will verify the three properties (deadlocks, data races and race conditions) in Pthreads and Dthreads for our illustrative programs, with the help of the model checker PAT [27, 28], which has been applied in various places [38, 39, 40]. To carry out the verification, we have implemented

Prepared using smrauth.cls
DOI: 10.1002/smr

the formal models in Section 4 and Section 5 in PAT[‡]. Note that our comparative verification is to check if Dthreads performs better than Pthreads on eliminating data races, preventing deadlocks and avoiding race conditions.

7.1. Verification of Deadlock

In this subsection, we consider the verification of the deadlock freedom in Pthreads and Dthreads. As we have constructed the models in PAT, there is a primitive assertion to describe this situation as below.

Property 1: Deadlock Freedom

#assert PthreadsDeadLock() deadlockfree;

#assert DthreadsDeadLock() deadlockfree;

There are many reasons of causing deadlocks. Our illustrative program shows a very typical scenario where the two threads are waiting for each other to release a mutex.

Fig. 9 shows the verification results of deadlock freedom in Pthreads and Dthreads. The assertion of *PthreadsDeadLock()* is invalid, indicating that there is a deadlock existing in the Pthreads model. Meanwhile, the verification result shows that Dthreads model has no deadlocks, as the assertion of *DthreadsDeadLock()* is valid.

PAT can provide a failure trace for the failure assertion. Fig. 25 (in appendix) shows a deadlock track of the Pthreads model. We can see that the main thread first creates *Thread1*. Then *Thread1* locks the mutex *m0* successfully. After that, the main thread also creates *Thread2*, and *Thread2* also locks the mutex *m1*. Then *Thread1* tries to lock mutex *m1*, but it fails. The request from *Thread2* to lock mutex *m0* blocks as well. Therefore, a deadlock appears.

7.2. Verification of Data Race

In this subsection, we focus on the verification of data race freedom of Pthreads and Dthreads. In Section 3, our illustrative programs with respect to data races and race conditions describe the most common scenario when a thread executes a assignment statement after a conditional statement. The statements check whether global variable *A* is less than 100. If not, *B* will add 100 and *A* will minus 100. Here we list the core conditional statement and assignment statements again for convenience.

```
if (A < 100){
    return NULL;
}
B = B + 100;
A = A - 100;
```

The threads in *PthreadsTransfer1()* and *DthreadsTransfer2()* run the same statements as above. In *PthreadsTransfer2()* and *DthreadsTransfer2()*, the threads add extra lock and unlock operations at each assignment statements. While in *PthreadsTransfer3()* and *DthreadsTransfer3()* the threads apply one lock operation and one unlock operation at the beginning and end of the three statements respectively.

John Erickson et al. [41] at Microsoft Research define data races as that if two memory accesses in a program access the same memory location concurrently and at least one of them is a write. In

[‡]As assertions in PAT can only deal with the process with no parameters, the models are renamed as below.

$$PthreadsDeadLock() = SystemPthreads(deadlock); DthreadsDeadlock() = SystemDthreads(deadlock);$$
$$PthreadsTransfer1() = SystemPthreads(transfer1); DthreadsTransfer1() = SystemDthreads(transfer1);$$
$$PthreadsTransfer2() = SystemPthreads(transfer2); DthreadsTransfer2() = SystemDthreads(transfer2);$$
$$PthreadsTransfer3() = SystemPthreads(transfer3); DthreadsTransfer3() = SystemDthreads(transfer3);$$

DOI: 10.1002/smr

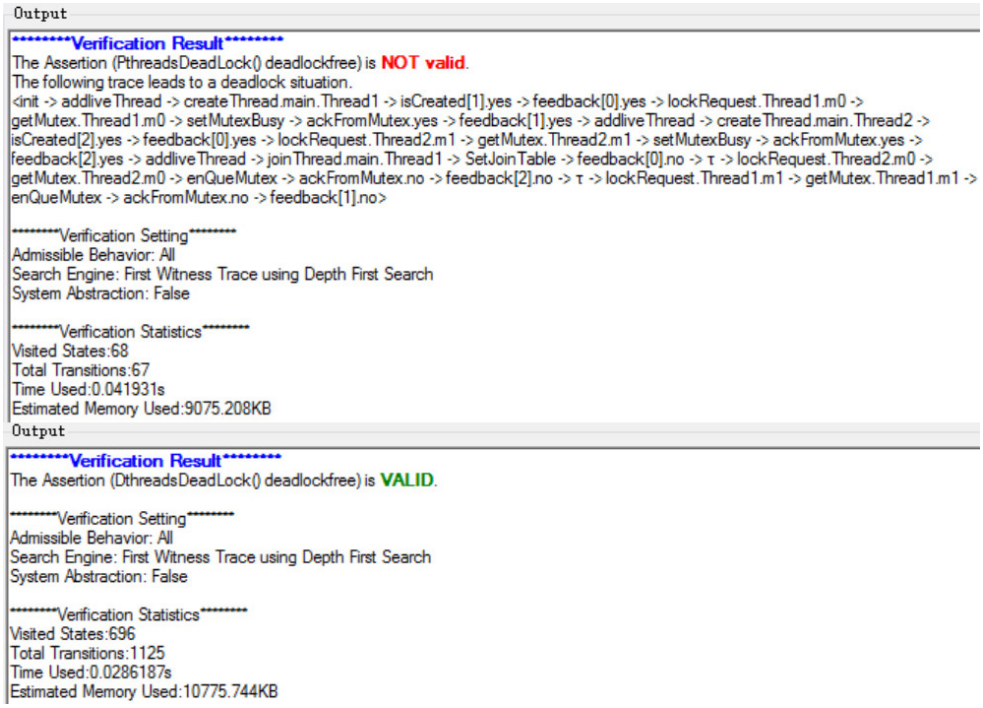


Figure 9. Result of Deadlock Freedom Verification of Pthreads and Dthreads Models

order to accomplish synchronization actions and interleaving actions, we translate the memory write access in CSP models to PAT models by spacial methods. For example, in PAT model of Pthreads, the assignment *B=B+100;* in function *ThreadFunc1* is implemented like this.

```
Write_ThreadFunc1_1() = Write_ThreadFunc1_1_I()
                        [] Write_ThreadFunc1_1_S();
Write_ThreadFunc1_1_I() = WriteVal_I{B=B+100;} -> Skip;
Write_ThreadFunc1_1_S() = DataRaceStart -> WriteVal_S{B=B+100;} ->
                          DataRaceEnd -> Skip;
```

Write_ThreadFunc1_1_I() represents the interleaving actions. And *Write_ThreadFunc1_1_S()* describes the synchronization actions. As *[]* is a choice operator, *Write_ThreadFunc1_1()* may do synchronization actions or interleaving actions. Action *DataRaceStart* and *DataRaceEnd* are used for synchronization. On this basis, we can simulate that the modification of the same location currently.

When action *WriteVal_S* is done, two threads are both modifying global variable A or B, or one of them is accessing A while another is changing B. It indicates that two threads change the shared data concurrently, which means a data race happens. The assertion checks that whether all the running traces contain *WriteVal_S*. The ideal result is that action *WriteVal_S* does not appear. Therefore, if the process passes the assertion, it means that action *WriteVal_S* is never performed.

Property 2: Data Race Freedom

#assert PthreadsTransfer1() |= [] ! *WriteVal_S*;

#assert DthreadsTransfer1() |= [] ! *WriteVal_S*;

#assert PthreadsTransfer2() |= [] ! *WriteVal_S*;

#assert DthreadsTransfer2() |= [] ! *WriteVal_S*;

#assert PthreadsTransfer3() |= [] ! *WriteVal_S*;

#assert DthreadsTransfer3() |= [] ! *WriteVal_S*;

DOI: 10.1002/smr

Figure 10. Result of Data Race Verification of Pthreads and Dthreads Models

Fig. 10 shows the verification results of data races in Pthreads and Dthreads for three scenarios of bank transfers. For the first bank transfer, we get an invalid result of the assertion of *PthreadsTransfer1()*, which means data races appear in the Pthreads model. However, *DthreadsTransfer1()* obtains a valid assertion result, indicating that the Dthreads model is data race free. Likewise, the valid results of the assertion of *PthreadsTransfer2()*, *DthreadsTransfer2()*, *PthreadsTransfer3()* and *DthreadsTransfer3()* show the second and the third programs of bank transfers will not cause data races.

In PAT, a failure trace is generated for the invalid assertion of *PthreadsTransfer1()*, shown in Fig. 26 (in appendix). The trace explains why the assertion fails. At the beginning, the main thread creates *Thread1* and *Thread2*, and waits for their ending by the join operation. Then *Thread1* first finds out that the global variable A is not less than 100, so it is allowed to change the value of the global variable B to $B + 100$. Then *Thread2* also checks the global variable A is not less than 100,

DOI: 10.1002/smr

it changes B to $B + 100$ while A is assigned to $A - 100$ by *Thread1*. At this moment, *Thread1* and *Thread2* are accessing the same memory location at the same time. This is not what we expect, as we hope action *WriteVal_S* never appears. Hence, the assertion does not stand.

In general, the second and third programs of bank transfers do not cause data races in Pthreads, Dthreads can also guarantee this property. In particular, a data race does appear in the first program of Pthreads, while Dthreads successfully avoids the bad situation.

7.3. Verification of Race Condition

In this subsection, we deal with the verification of the race condition in Pthreads and Dthreads. Race conditions and data races have the same word race literally, whereas there is the overlap and difference between them. While they frequently go hand in hand, there is no subset relationship between them and neither one is the sufficient or necessary condition for another. As long as two threads access the same memory location at the same time and one of them is doing a write operation, a data race occurs. By contrast, only the improper timing or ordering of events, which affects the program correctness, can cause a race condition. In other words, data races and race conditions can both appear in one program. A program can only have data races and also can only own race conditions. Using the models in PAT, assertions are listed as below.

Property 3: Race Condition Freedom

#define goal $A < 0$;

#assert PthreadsTransfer1() *reaches goal*;

#assert DthreadsTransfer1() *reaches goal*;

#assert PthreadsTransfer2() *reaches goal*;

#assert DthreadsTransfer2() *reaches goal*;

#assert PthreadsTransfer3() *reaches goal*;

#assert DthreadsTransfer3() *reaches goal*;

As we know that in a bank transfer, the balance in the account can not be less than 0. Considering our example bank transfer programs, we need to ensure that A can not be negative guaranteeing the program correctness. During the transfer, once A is less than 0, this means that a race condition exists. We use the assertions above to check whether the process can reach a state at which A is negative.

Fig. 11 displays the verification results of race conditions in Pthreads and Dthreads. The four valid results of *PthreadsTransfer1()*, *DthreadsTransfer1()*, *PthreadsTransfer2()* and *DthreadsTransfer2()* illustrate the situation that the variable A is negative occurs after running the responding processes. The last two invalid results indicate that in *PthreadsTransfer3()* and *DthreadsTransfer3()* the variable A can always be positive or zero.

Our above results indicate that the first two programs of bank transfers in Pthreads and Dthreads both have race conditions and for the last program there is no race condition. Consequently, both Pthreads and Dthreads can not prevent race conditions. Appearing of race conditions may cause bad situations, programmers need to be more careful.

8. MODEL EXTENSION

This section is devoted to the extensions we have made to enrich our Pthreads and Dthreads model. We model four more API functions *pthread_cond_wait()*, *pthread_cond_signal()*, *pthread_barrier_wait()* and *pthread_cancel()*. That is, we have now modelled all the synchronization operations described in [1]. Consequently, our Pthreads and Dthreads model becomes more general and can now specify more programs using synchronization operations of the Pthreads and Dthreads API functions.

DOI: 10.1002/smr

Figure 11. Result of Race Condition Verification of Pthreads and Dthreads Models

8.1. Condition Variables in Pthreads

Condition variables are synchronization primitives. It will block the threads until a special condition happens. For simplicity, we assume every user-defined condition owns a queue of threads that wait for it in Pthreads. When a thread calls *pthread_cond_wait()*, it must have owned a mutex. Then the function will unlock the mutex and put the calling thread into the wait queue of the condition variable. Meanwhile, the thread decreases the live thread count, which means it can not keep on doing other things. *pthread_cond_signal()* is the corresponding function of *pthread_cond_wait()*. It wakes up one of the threads that waits on the corresponding condition variable. Then the thread needs to compete for the related mutex.

Firstly let us add some new subprocesses in the process *Program* to simulate the API *pthread_cond_wait()* and *pthread_cond_signal()*. They are expressed as $CondWait_{t,c,m}$ and $CondSignal_{t,c}$.

$$\begin{aligned} CondWait_{t,c,m} &=_{df} condWaitRequest_t!c.m \rightarrow condWaitSub_t \\ CondWaitSub_t &=_{df} feedback_t?v \rightarrow (Skip \lhd (v == yes) \rhd condWaitSub_t) \\ CondSignal_{t,c,m} &=_{df} condSignalRequest_t!c.m \rightarrow feedback_t?v \rightarrow Skip \end{aligned}$$

Secondly, we also need to put two components $CConWait_t$ and $CCondSignal_t$ into *Controller* to deal with requests considering condition variables, and they are in parallel with other components such as $CLock_t$. They both receive the relative requests from *Program*, send the information to *Condition* (described later), then await the acknowledgement.

$$\begin{aligned} CConWait_t =_{df}\ & condWaitRequest_t?c.m \rightarrow condWait_t!c.m \rightarrow ackFromCondition?v \rightarrow \\ & feedback_t!v \rightarrow Skip; Controller \\ CCondSignal_t =_{df}\ & condSignalRequest_t?c \rightarrow condSignal_t!c \rightarrow ackFromCondition?v \rightarrow \\ & feedback_t?v \rightarrow Skip; Controller \end{aligned}$$

DOI: 10.1002/smr

Thirdly, a new module *Condition* is introduced into our model. *Condition* is responsible for the main duty of calling the functions *pthread_cond_wait()* and *pthread_cond_signal()*. It has connections with *Controller*, *Program* and *Buffer*.

$$
\begin{aligned}
Condition =_{df}\ & (\Box_{t\in 1..(T_n-1)} CDConWait_t) \\
& \Box\ (\Box_{t\in 1..(T_n-1)} CDConSignal_t) \\
& \Box\ (endCondition?v \rightarrow ackFromCondition!yes \rightarrow Skip) \\
CDConSignal_t =_{df}\ & condSignal_t?c \rightarrow ConditionQueLength_c?length \rightarrow \\
& \left(\begin{array}{l} \left(\begin{array}{l} ConditionQueDequeue_c?t' \rightarrow ConditionQueLength_c!Decrease \rightarrow \\ liveThread!Increase \rightarrow RelatedMutex_c?m \rightarrow getMutex_{t'}!m \rightarrow \\ ackFromMutex?v \rightarrow feedback_{t'}!v \rightarrow ackFromCondition!yes \rightarrow Skip \end{array} \right) \\ \lhd (length > 0) \rhd \\ (ackFromCondition!yes \rightarrow Skip) \end{array} \right); \\
& Condition
\end{aligned}
$$

$CDConWait_t$ primarily simulates *pthread_cond_wait()*. It tells *Mutex* to release the mutex and decreases the live thread count. Then thread t is inserted into the condition variable queue of condition variable c. In Pthreads the thread must acquire a mutex before calling *pthread_cond_wait()*, but it will release the mutex when calling the function. Additionally, it needs to get the mutex again when it is woken up by *pthread_cond_signal()*. Therefore, we need to record the mutex. A negative acknowledge is sent to *Controller*. Subsequently, *Controller* will give *Program* a negative feedback, causing it to keep on waiting until some other threads wake it up.

$$
\begin{aligned}
CDConWait_t =_{df}\ & condWait_t?c.m \rightarrow releaseMutex_t!m \rightarrow ackFromMutex?v \rightarrow \\
& liveThread!Decrease \rightarrow ConditionQue_{t,c}!Enqueue \rightarrow \\
& RelatedMutex_c!m \rightarrow ackFromCondition!no \rightarrow Skip; Condition
\end{aligned}
$$

8.2. Condition Variables in Dthreads

The situation becomes more complicated in Dthreads when taking into account conditional variables. The modification to *Program* is the same as the one in Pthreads notwithstanding it is totally different concerning *Controller*. As a result of guaranteeing determinism for condition variables, more subprocesses need to be introduced into *Controller*.

We simulate the main function of *pthread_cond_wait()* and *pthread_cond_signal()* in *Controller*. As the threads in Dthreads own two different phases, we first update the subprocess *ParallelPhase* of *Controller*. Two sub components $PPCondWait_t$ and $PPCondSignal_t$ are added in parallel with other components, such as $PPLock_t$. Their duty is to record the synchronization operations and announce their arrivals to *Fence*.

$$
\begin{aligned}
PPCondWait_t =_{df}\ & condWaitRequest_t?c \rightarrow setCondVar_t!c \rightarrow ThreadState_t!condw \rightarrow \\
& arriveFence!t \rightarrow Controller \\
PPCondSignal_t =_{df}\ & condSignalRequest_t?c \rightarrow setCondVar_t!c \rightarrow ThreadState_t!conds \rightarrow \\
& arriveFence!t \rightarrow Controller
\end{aligned}
$$

The core task of the functions is done by $SPCondWait_t$ and $SPCondSignal_t$, which are introduced into the subprocess *SerialPhase* of *Controller*. We no longer need to deal with locking and unlocking of mutex of *pthread_cond_wait()*, since the operations on the token can do the same things.

$SPCondWait_t$ describes how *Controller* implements *pthread_cond_wait()*. Thread t needs to acquire the token and then commits the changes to shared memory. In addition, we decrease the live thread count and put thread t to the condition queue of condition variable c. Lastly, Controller makes t release the token and sends a failure feedback to *Program*.

$$
\begin{aligned}
SPCondWait_t =_{df}\ & AcquireToken_t; commitWrite!t \rightarrow ackFromCommit?v \rightarrow getCondVar_t?c \rightarrow \\
& liveThread!Decrease \rightarrow ConditionQue_{t,c}!Enqueue \rightarrow ConditionQueLength_c!Increase \rightarrow \\
& releaseToken!t \rightarrow ackFromToken?v \rightarrow feedback_t!no \rightarrow Controller
\end{aligned}
$$

DOI: 10.1002/smr

Similarly, $SPCondSignal_t$ defines how *pthread_cond_signal()* works. Firstly, thread t acquires the token and also commits all the modifications. If no thread is waiting on this condition, thread t passes the token to the next thread and *Program* receives a success feedback. Otherwise, we get thread t' by dequeuing an element of condition queue of condition variable c. It is woken up by increasing the live thread count, and inserting it into the token queue. The following steps are the same as when there is no thread waiting on the condition.

$$\begin{aligned} SPCondSignal_t =_{df}\ & AcquireToken_t; commitWrite!t \rightarrow ackFromCommit?v \rightarrow getCondVar_t?c \rightarrow \\ & ((ConditionQueDequeue_c?t' \rightarrow ConditionQueLength_c!Decrease \rightarrow \\ & TokenQueEnqueue!t' \rightarrow TokenQueLength!Increase \rightarrow liveThread!Increase \rightarrow Skip) \\ & \lhd (ConditionQueLength_c > 0) \rhd Skip); \\ & releaseToken!t \rightarrow ackFromToken?v \rightarrow feedback_t!yes \rightarrow Controller \end{aligned}$$

In Section 5.4 we give the formal definition for *ParallelPhase* and *SerialPhase* (see page 12). Now we want to update them by adding $PPCondWait_t$ and $PPCondSignal_t$ to *ParallelPhase* and inserting $SPCondWait_t$ and $SPCondSignal_t$ to *SerialPhase*. In this way, we extend the capabilities of process Controller to deal with *pthread_cond_signal()* and *pthread_cond_wait()*.We extend the modelling of frequently-used API function as more as possible. In this way, the performance and practicability of our models are improved greatly.

$$\begin{aligned} ParallelPhase =_{df}\ & (\Box_{t\in 1..(T_n-1)} PPLock_t) \Box (\Box_{t\in 1..(T_n-1)} PPUnLock_t) \\ & \Box (\Box_{t\in 1..(T_n-1)} PPCreate_t) \Box (\Box_{t\in 1..(T_n-1)} PPJoin_t) \\ & \Box (\Box_{t\in 1..(T_n-1)} PPCondWait_t) \Box (\Box_{t\in 1..(T_n-1)} PPCondSignal_t) \\ & \Box EndController \\ SerialPhase =_{df}\ & leaveFence?v \rightarrow TokenQueDequeue?t \rightarrow TokenQueLength!Decrease \rightarrow \\ & ThreadState_t?state \rightarrow (SPLock_t \lhd (state == lock) \rhd \\ & (SPUnLock_t \lhd (state == unlock) \rhd (SPCreate_t \lhd (state == create) \rhd \\ & (SPJoin_t \lhd (state == join) \rhd (SPExit_t \lhd (state == exit) \rhd \\ & (SPConWait_t \lhd (state == conw) \rhd (SPConSignal_t \lhd (state == cons) \rhd \\ & Contoller)))))) \end{aligned}$$

8.3. Barriers in Pthreads

The *pthread_barrier_wait()* function tries to synchronize all the threads using the same barrier, hence a thread will be blocked until each of the involved threads has arrived at the barrier.

In the beginning, a new process $BarrierWait_{t,b}$ is required to help the programmer model the related API directly.

$$\begin{aligned} BarrierWait_{t,b} =_{df}\ & barrierWaitRequest!t.b \rightarrow BarrierWaitSub_t \\ BarrierWaitSub_t =_{df}\ & feedback_t?v \rightarrow (Skip \lhd (v == yes) \rhd BarrierWaitSub_t) \end{aligned}$$

Some new adjustments happen in *Controller*. $CBarrierWait_t$ is in parallel with other subprocesses in *Controller*. Its function is to record the barrier status, send the request to *Barrier* (described later) and finally give a feedback to *Program*.

$$\begin{aligned} CBarrierWait_t =_{df}\ & barrierWaitRequest_t?b \rightarrow ThreadState_t!barrier \rightarrow barrierWait_t!b \rightarrow \\ & ackFromBarrier?v \rightarrow feedback_t!v \rightarrow Skip; Controller \end{aligned}$$

DOI: 10.1002/smr

We also introduce a new module *Barrier* to deal with the barrier specially. It communicates with *Controller*, *Program* and *Buffer*.

$$
\begin{aligned}
Barrier =_{df}\ & (\Box_{t\in 1..(T_n-1)} BBarrierWait_t) \\
& \Box\ (endBarrier?v \rightarrow ackFromBarrier!yes \rightarrow Skip) \\
BBarrierWait_t =_{df}\ & barrierWait_t!b \rightarrow liveThread!Decrease \rightarrow BarrierSetIn_b!t \rightarrow \\
& BarrierSetSize_b!Increase \rightarrow liveThread!Decrease \rightarrow \\
& (\ (BarrierWaitSub_b; ackFromBarrier!yes \rightarrow Barrier) \\
& \lhd (BarrierSetSize_b == BarrierNum_b) \rhd \\
& (ackFromBarrier!no \rightarrow Barrier)\ ) \\
BBarrierWaitSub_b = df\ & (\ (\ BarrierSetOut_b?t \rightarrow BarrierSetSize_b!Decrease \rightarrow \\
& ThreadState_t!run \rightarrow liveThread!Increase \rightarrow feedback_t!yes \rightarrow \\
& BBarrierWaitSub_b\ ) \lhd (BarrierSetSize_b > 0) \rhd Skip\ )
\end{aligned}
$$

In Pthreads, the waking up order of the threads blocked at the barrier is nondeterministic. Consequently, we use sets to store IDs of threads waiting at barriers instead of queues, assuming that the element is taken out of a set randomly. The main responsibility of $BBarrierWait_t$ is to lift a barrier until all the corresponding threads have arrived, then it will wake them up arbitrarily. More specifically, it decreases the live thread count, puts the threads into the related barrier set, then judges whether all the threads involved with the barrier have called *pthread_barrier_wait()*. If so, it wakes up the elements in the barrier set arbitrarily, which is handled by $BBarrierWaitSub_b$.

8.4. Barriers in Dthreads

The modification concerning barriers in *Program* is the same as the situation in Pthreads.It may be confusing to figure out the difference between the fence and the barrier, as they have the same meaning literally. The fence is the one that can stop the threads doing synchronization operations in Dthreads, and it releases the threads when all the living threads have been blocked. Besides, the waking up order of the threads is deterministic. The barrier is the concept used in Pthreads API functions. The barrier also bars all the threads that have called *pthread_barrier_wait()*, and it frees them when all of them arrived. In addition, the waking up order of the threads is random.

As Dthreads re-implements Pthreads, we also need to modify the mechanism of implementing *pthread_barrier_wait()*. We primarily translate the implementation of *pthread_ barrier_wait()* in Dthreads as follows. *Controller* is updated in Dthreads to achieve the goal. $PPBarrierWait_t$ is used to record barrier operation from *Program* in *Arrival Phase* and tell *Fence* the arrival of the thread. We add $PPBarrierWait_t$ to *ParallelPhase* in *Controller*, which is in parallel with other parts, such as $PPLock_t$.

$$
\begin{aligned}
PPBarrierWait_t =_{df}\ & BarrierWaitRequest_t?b \rightarrow setBarrierVar_t!b \rightarrow ThreadState_t!barrier \rightarrow \\
& arriveFence!t \rightarrow Controller
\end{aligned}
$$

SerialPhase of *Controller* also needs to be refreshed. We introduce the subprocess $SPBarrierWait_t$ to depict the situation when a thread leaves the fence whose *state* is barrier.

$$
\begin{aligned}
SPBarrierWait_t =_{df}\ & AcquireToken_t; commitWrite!t \rightarrow ackFromCommit?v \rightarrow getBarrierVar_t?b \rightarrow \\
& BarrierQueEnqueue_b!t \rightarrow BarrierQueLength_b!Increase \rightarrow \\
& liveThread!Decrease \rightarrow (\ SPBarrierWaitSub_{t,b} \\
& \lhd (BarrierQueLength_b == BarrierNum_b) \rhd (\ releaseToken!t \rightarrow \\
& ackFromToken?v \rightarrow feedback_t!no \rightarrow Controller\ )\ ) \\
SPBarrierWaitSub_{t,b} =_{df}\ & (\ (BarrierQueDequeue?t' \rightarrow BarrierQueLength!Decrease \rightarrow \\
& TokenQueEnqueue!t' \rightarrow TokenQueLength!Increase \rightarrow liveThread!Increase \rightarrow \\
& SPBarrierWaitSub_{t,b}\ ) \lhd (BarrierQueLength_b > 0) \rhd \\
& (\ releaseToken!t \rightarrow ackFromToken?v \rightarrow feedback_t!yes \rightarrow Skip\ )\ )
\end{aligned}
$$

First, thread t gets the token when it leaves the fence and commits its private changes to shared memory. Then thread ID is appended to the barrier queue of b and the living thread number is

DOI: 10.1002/smr

decreased, which means that thread t is blocked by the barrier. We check whether all the related threads have arrived at the barrier. If so, thread t releases the token and feeds Program with a failure acknowledgement. If not, all the threads are retrieved from the barrier queue of b. And we add them into the token queue and increase the live thread count, which indicates that the blocked threads have been able to get the token again. Eventually, thread t passes the token to the next thread and sends a success feedback to *Program*. Similarly, $SPBarrierWait_t$ is combined into process *SerialPhase* and $PPBarrierWait_t$ is also added into process *ParallelPhase*.

8.5. Thread Cancellation in Pthreads and Dthreads

In this section, we list the modelling of the API *pthread_cancel()* in our Pthreads model and Dthreads model. As mentioned before, Dthreads share the same API names with Pthreads. Thus, the changes in *Program* are the same. It sends the cancel request to *Controller* and waits for reply.

$$Cancel_t =_{df} condCancelRequest?t \rightarrow feedback_t?v \rightarrow Skip$$

We also add a new subprocess in *Controller*. $CCancel_t$ belongs to our Pthreads model, while $PPCancel_t$ and $SPCancel_t$ pertain to *SerialPhase* and *ParallelPhase* in our Dthreads model respectively. They reduce the live thread count and remove the thread from the condition queues, the barrier sets or queues, or the mutex sets. There is one big difference though. In Dthreads, *Controller* needs to get the token before doing the things described before.

$$\begin{aligned}
CCancel_t =_{df}\ & condCancelRequest?t \rightarrow ThreadState_t!cancel \rightarrow ConditionQueDelete!t \rightarrow \\
& MutexSetDelete!t \rightarrow BarrrierSetDelete!t \rightarrow liveThread!Decrease \rightarrow \\
& feedback_t!yes \rightarrow Controller \\
PPCancel_t =_{df}\ & CancelRequest?t \rightarrow ThreadState_t!cancel \rightarrow arriveFence!t \rightarrow Controller \\
SPCancel_t =_{df}\ & AcquireToken_t; ConditionQueDelete!t \rightarrow BarrrierQueDelete!t \rightarrow \\
& liveThread!Decrease \rightarrow releaseToken!t \rightarrow ackFromToken?v \rightarrow feedback_t!yes \rightarrow Controller
\end{aligned}$$

9. VERIFICATION EXTENSION

As we have extended our model by adding new API functions, in this section we introduce new example programs to test whether the new parts work well with original models. More importantly, there is still some thing of deadlock avoidance of Dthreads we want to check. [1] states that it is possible for a program run with Dhreads to deadlock, but only for programs that can also deadlock with Pthreads. We find more proper examples to verify this. Two classic producer and consumer programs are introduced to dig out more about the performance of avoiding deadlocks for Pthreads and Dthreads, while applying our new API functions modelled in Section 8. Meanwhile, we record the time of verification to explain the complexity of our Pthreads and Dthreads models.

9.1. Example Programs

The core functions of the classic producer and consumer programs are illustrated below.

```
void consume(){
    pthread_mutex_lock(&th_lock);
    if (val == 0)
      pthread_cond_wait(&th_cond,&th_lock);
    }
    val = val - 1;
    pthread_mutex_unlock(&th_lock);
    pthread_cond_signal(&th_cond);
}

void produce(){
    pthread_mutex_lock(&th_lock);
```


DOI: 10.1002/smr

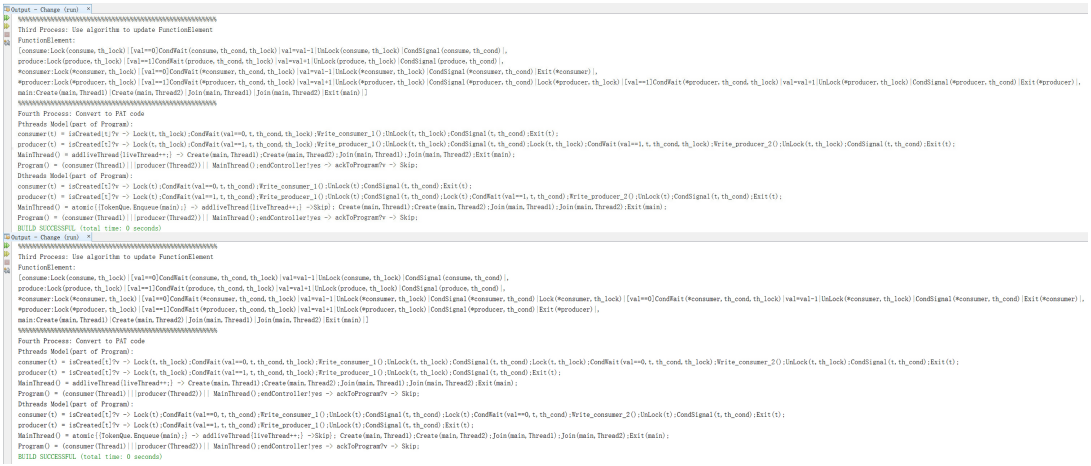


Figure 12. Processing procedure of changing the C code of the consumer and producer programs

```
    if (val == 1)
      pthread_cond_wait(&th_cond,&th_lock);
    }
    val = val + 1;
    pthread_mutex_unlock(&th_lock);
    pthread_cond_signal(&th_cond);
}
```

Function *consume* locks the mutex *th_lock* and then inspects if the global variable *val* is 0. If it is, *consume* has to wait for *val* to be a larger number. Because it will consume *val* by reducing it. Finally, it unlocks the mutex *th_lock* and signals other threads that wait on the condition *th_cond*. Similarly, function *produce* almost does the same things. But it waits for *val* not to be equal to 1 and then increments *val*.

```
void *consumer(void *arg){
    consume();
    return NULL;
}

void *producer(void *arg){
    produce();
    produce();
    return NULL;
}
```

In the first program, we apply the functions to represent the mission for each thread. Thread *consumer* wants to reduce *val* and thread *producer* demands to increment *val* twice.

```
void *consumer(void *arg){
    consume();
    consume();
    return NULL;
}

void *producer(void *arg){
    produce();
    return NULL;
}
```

For the second program, *consumer* and *producer* do the opposite things compared with the one in the first program. Thread *consumer* wants to consume val twice, and *producer* only produces once. We use our method to convert C code to PAT code automatically, shown in Fig.12.

DOI: 10.1002/smr

Figure 13. Result of Deadlock Freedom Verification of Pthreads and Dthreads Models

9.2. Verification of Deadlock

We verify the property of deadlock freedom of the two programs running in Pthreads and Dthreads.

Property 4: Deadlock Freedom

#assert PthreadsProCon1() deadlockfree;

#assert DthreadsProCon1() deadlockfree;

#assert PthreadsProCon2() deadlockfree;

#assert DthreadsProCon2() deadlockfree;

The results of verification are shown in Fig.13. *PthreadsProCon1()* and *DthreadsProCon1()* both pass the assertion, indicating that the first program can never cause a deadlock in Pthreads and Dthreads. The failure of *PthreadsProCon2()* and *DthreadsProCon2()* shows that neither Pthreads nor Dthreads can prevent the deadlock in the second program. We give the analysis of deadlock in the following subsection in more details.

9.3. Analysis of Deadlock

As the property of deadlock has been verified several times in Section 7.1 and Section 9.2, in this subsection, we introduce RAG (Resource Allocation Graph) to analyse the deadlocks. RAGs are directed labeled graphs used to represent the current state of a system, which illustrates how interacting processes can deadlock. As it is applied for processes and resources, we do some modification to make it suitable for our six models involved with deadlock. They are *PthreadsDeadLock()*, *DthreadsDeadLock()*, *PthreadsProCon1()*, *DthreadsProCon1()*, *PthreadsProCon2()* and *DthreadsProCon2()*. The whole system consists of three types of elements illustrated in Fig.14. We use threads instead of processes. Mutexes and val are considered to be resources. Request edge represents that thread T_i is requesting for resource R_j and assignment edge denotes that thread T_i is assigned to resource R_j.

In *PthreadsDeadLock()*, there are three possible RAGs depicted in Fig.15. We reduce the RAGs

DOI: 10.1002/smr

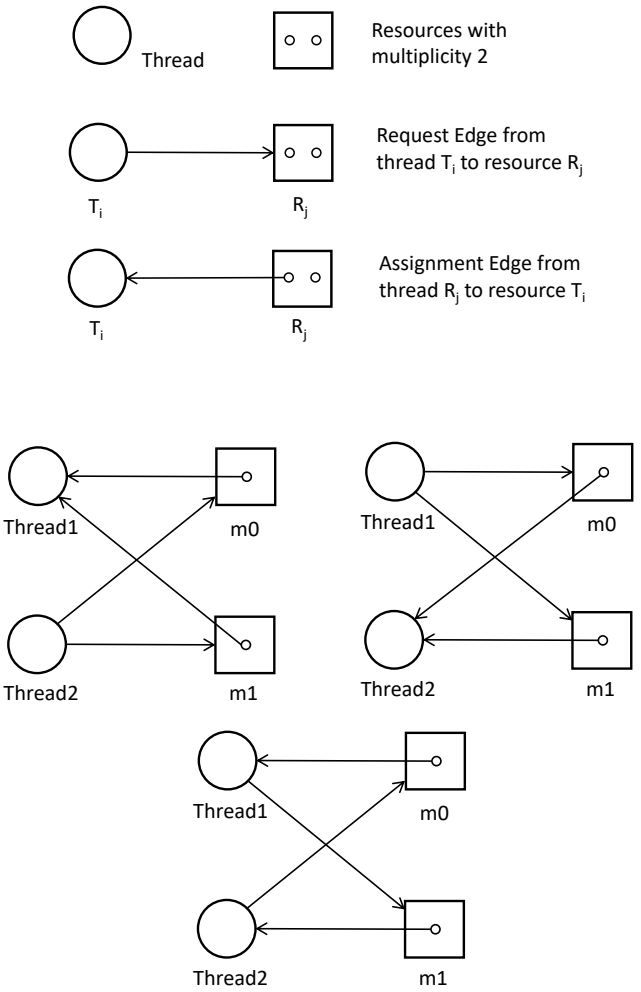


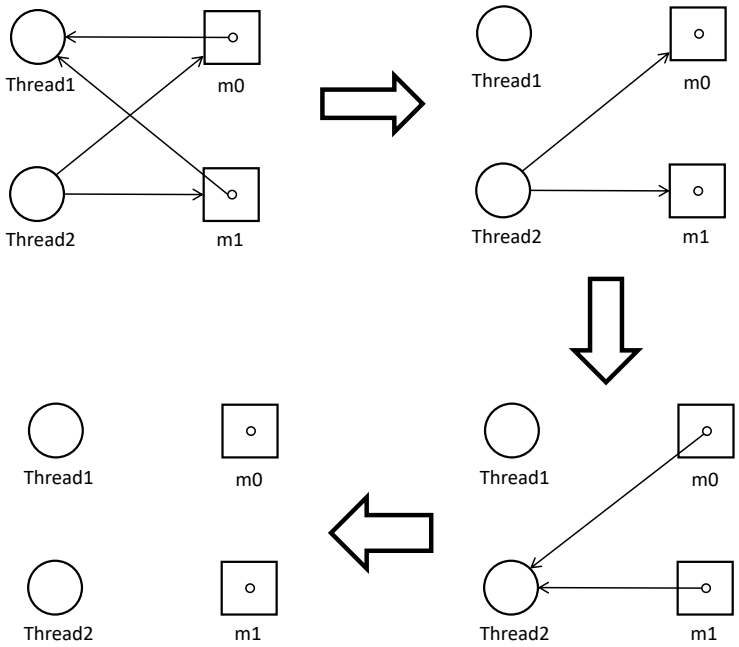
Figure 15. RAGs of Pthreads model *PthreadsDeadLock()*

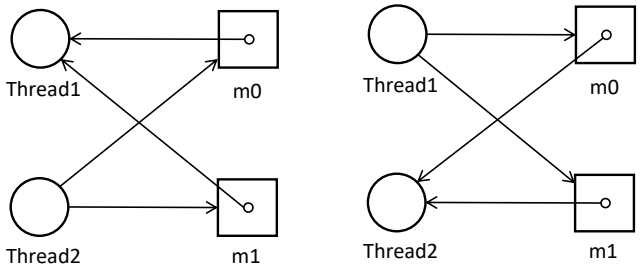
to detect deadlocks. The reducing steps are as follows. First, we find a non-isolated thread node which only owns assignment edges, cut off the related assignment edges and turn it to an isolated node. As a result, several resources are released. Then the related resources are assigned to one thread which is waiting for them. That is to say, the request edges of this thread are changed to assignment edges. Repeat the above steps until no non-isolated thread node which only owns assignment edges is found. Finally, if all the thread nodes become isolated, there is no deadlock in the system. Otherwise, deadlock occurs. As depicted in Fig.14, the edge can only appear between a thread node and a resource node. An RAG is called fully reducible if it can be reduced to a graph without any edges. Fig.16 demonstrates the reduction process of the first RAG in Fig.15.

According to the reducing steps of the RAG, the first two RAGs in Fig.15 are fully reducible. However, we can not find any non-isolated node which only owns assignment edges in the last RAG in Fig.15, which means it is not reducible. Obviously, deadlock occurs in Pthreads model *PthreadsDeadLock()*. Meanwhile, two RAGs can be built from *DthreadsDeadLock()* in Fig.17, which are the same as the first two RAGs of *PthreadsDeadLock()* illustrated in Fig.15. As both of the RAGs are reducible, no deadlock appears in Dthreads model *DthreadsDeadLock()*.

We construct RAGs from *PthreadsProCon1()*, *DthreadsProCon1()*, *PthreadsProCon2()* and *DthreadsProCon2()* as well. As *val* is considered to be the resource and the initial value of *val* is zero. Thread *producer* can not move on until *consumer* has increased *val*. As *consumer* does not request the resource, we convert *producer* from the thread to the resource in the RAG for

DOI: 10.1002/smr





Figur

Figure 1

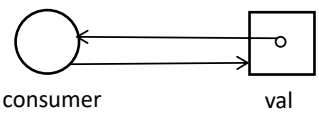


Figure 19. RAG of *PthreadsProCon2()* and *DthreadsProCon2()*

its producing function. The RAGs of *PthreadsProCon1()*, *DthreadsProCon1()*, *PthreadsProCon2()* and *DthreadsProCon2()* are shown in Fig.18 and Fig.19 respectively. If we apply the reduction to these two RAGs, the first one is reducible while the second one is not. This means that there is no deadlock in *PthreadsProCon1()* and *DthreadsProCon1()*. Meanwhile, *PthreadsProCon2()* and *DthreadsProCon2()* will confront deadlock.

DOI: 10.1002/smr

The deadlocks due to resource allocations can only occur if four conditions hold simultaneously. They are listed as follows:

- **Mutual Exclusion**: One resource can only be used by one thread.
- **Hold and Wait**: At least one resource is held by a thread and the thread is waiting for new resources.
- **No Preemption**: The requester of resource can not take the resource until it is released by the holder.
- **Circular Wait**: A list of processes are waiting for each other in circular form.

As shown by the irreducible RAG in Fig.15, the implementation of Pthreads can cause deadlock by meeting these conditions simultaneously. But special mechanisms in Dthreads implement deadlock prevention, demonstrated by reducible RAGs in Fig.17. The irreducible RAG depicted in Fig.19 shows the other reason causing deadlock. It indicates that Dthreads can do nothing for the deadlock situation, where the reason of deadlock comes from environment that the thread requires more resources than the total number of resource in the whole system.

Therefore, combining with the results in Section 7.1 and Section 9.2, we can conclude that Dthreads can not eliminate all the deadlock situations, but it does perform better than Pthreads for some special deadlock programs. It also means that deadlocks stemming from particular resource problems can be handled better by Dthreads than by Pthreads.

9.4. Complexity of Modeling

In order to deal with the complexity of our Pthreads and Dthreads models, we utilize the verification time of the assertions. PAT provides the verification time it used, so we can figure out the complexity of our model broadly. If an assertion is passed, the whole model is checked deeply to find whether there is any violation. If a fault trace is found, the time will be equal or less than the time of exploring the whole model. As a result, we choose an assertion to verify a property that will be valid for every model. Divergence free property is suitable, for its verification time can reflect the complexity of modelling, as all of our models can pass it. Consequently, we choose the assertion of divergence free rather than deadlock free. If a process performs an infinite internal actions, it is said to be divergent. This assertion is applied to the twelve models as below.

Property 5: Divergence Freedom

#assert PthreadsDeadLock() *divergencefree*;
#assert DthreadsDeadLock() *divergencefree*;
#assert PthreadsTransfer1() *divergencefree*;
#assert DthreadsTransfer1() *divergencefree*;
#assert PthreadsTransfer2() *divergencefree*;
#assert DthreadsTransfer2() *divergencefree*;
#assert PthreadsTransfer3() *divergencefree*;
#assert DthreadsTransfer3() *divergencefree*;
#assert PthreadsProCon1() *divergencefree*;
#assert DthreadsProCon1() *divergencefree*;
#assert PthreadsProCon2() *divergencefree*;
#assert DthreadsProCon2() *divergencefree*;

We present the time used for verification in Table III. From the results, except *PthreadsTransfer1()* and *DthreadsTransfer1()*, the verification of Pthreads models requires more time than Dthreads models. Only these two processes do not use lock operations. We think it is because the lock operations will increase the frequency of feedback no and re-lock operations in Pthreads

DOI: 10.1002/smr

models.We conclude that our Pthreads models are larger than Dthreads models in most cases, except *PthreadsTransfer1()* and *DthreadsTransfer1()*.

Table III. The Verification Time of Divergence-free Assertion

Model \ Times(s)	1	2	3	4	5	6	7	8	avg
PthreadsDeadLock()	0.1360	0.1239	0.1227	0.1213	0.1199	0.1211	0.1202	0.1208	0.1232
DthreadsDeadLock()	0.0462	0.0383	0.0409	0.0394	0.0389	0.0381	0.0381	0.0391	0.0448
PthreadsTransfer1()	0.0357	0.0314	0.0313	0.0320	0.0334	0.0325	0.0311	0.0330	0.0326
DthreadsTransfer1()	0.1602	0.1529	0.1505	0.1492	0.1482	0.1505	0.1478	0.1490	0.1510
PthreadsTransfer2()	0.2459	0.2300	0.2305	0.2286	0.2281	0.2276	0.2289	0.2265	0.2308
DthreadsTransfer2()	0.0738	0.0660	0.0656	0.0651	0.0677	0.0665	0.0663	0.0662	0.0672
PthreadsTransfer3()	0.0822	0.0756	0.0752	0.0786	0.0768	0.0780	0.0752	0.0781	0.0775
DthreadsTransfer3()	0.0185	0.0158	0.0180	0.0183	0.0164	0.0174	0.0163	0.0180	0.0173
PthreadsProCon1()	0.4378	0.4182	0.4210	0.4206	0.4216	0.4201	0.4215	0.4247	0.4232
DthreadsProCon1()	0.0804	0.0689	0.07505	0.0712	0.0708	0.0703	0.0708	0.0681	0.0714
PthreadsProCon2()	0.1456	0.1394	0.1332	0.1354	0.4216	0.1379	0.1372	0.1364	0.1377
DthreadsProCon2()	0.0510	0.0441	0.0461	0.0432	0.0434	0.0427	0.0460	0.0474	0.0455

10. A NEW PROGRAMMING MODEL OF BANK TRANSFER

Dthreads has problems on scalability over dozens or hundreds of threads, as it is fine-grained, while Pthreads is more scalable, but it is also vulnerable to deadlocks. In this section, we introduce a new example of bank transfer with several transfer actions, and change its granularity by manipulating mutexes. Then new Pthreads and Dthreads models are built to illustrate how a coarse-grained programming model works. Finally, analysis and verification are done for better understanding of the essence of reducing the cost of parallelism.

10.1. Example program

Our new example program consists of five bank transfers illustrated in Fig.20. To ensure all the transfers to be performed successfully, we assign the initial value of accounts A, B, C, D, and E to 100, 100, 200, 100 and 0, respectively. For simplicity, each transfer will move 100 from one account to another. We introduce an algorithm to support coarse-grained programming, which also avoids data races in Pthreads of our bank transfer model effectively.

Fig.21 illustrates the application of the algorithm to the example program. First of all, a directed graph is built. The accounts are denoted by nodes and the transfers between accounts are represented by directed edges. As each thread is only related to one bank transfer, that is to say, one directed edge represents one thread. A transfer involves two accounts, and the two accounts may connect other accounts because of other transfers. Therefore, we need to divide the accounts into groups according to related transfers. As we have used the directed graph to describe the transfers and accounts, searching for connected components is a solution. WCCs (Weakly Connected Components) are chosen instead of SCCs (Strongly Connected Components), because we focus on the relation between two accounts involving the transfer rather than the direction of the transfer. An SCC is

DOI: 10.1002/smr

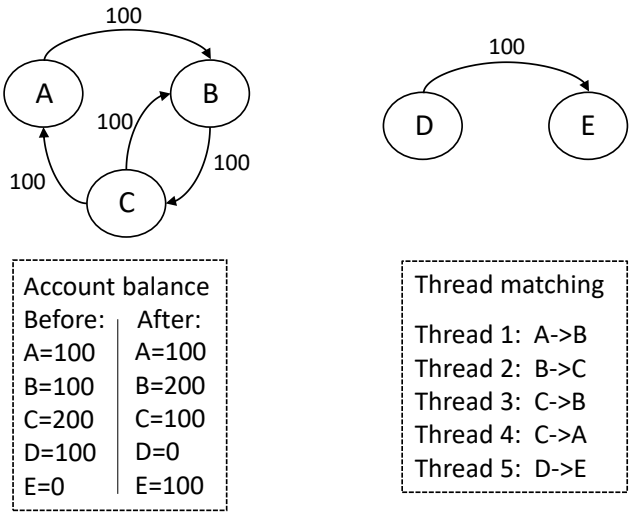


Figure 20. Bank Transfer, Account Balance and Thread Matching

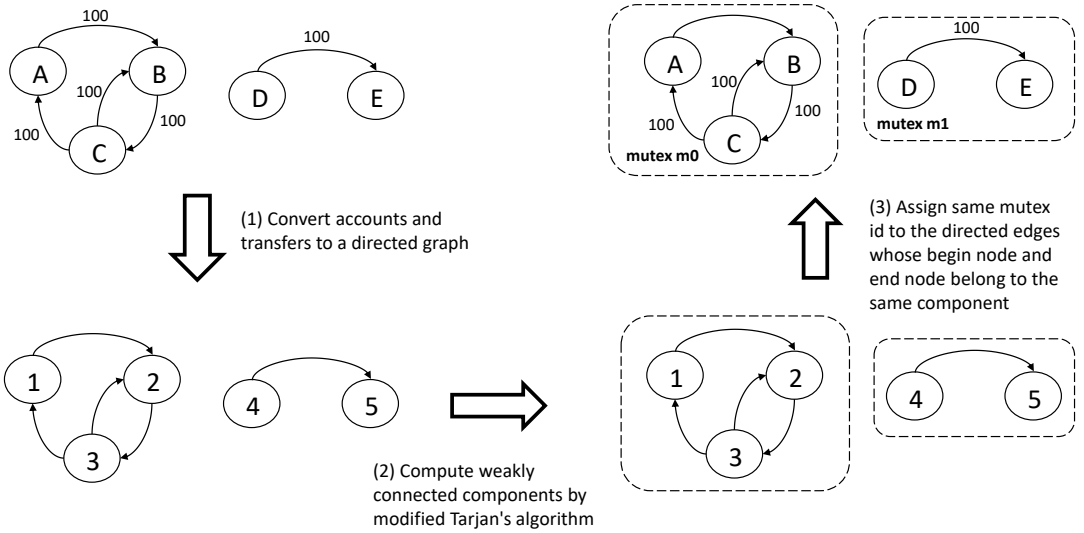


Figure 21. Processing of assigning mutexes

a sub-graph where there is a path from every node to every other node. While, a WCC is a sub-graph in which any two nodes are connected by some path, ignoring directions. As a result, we modify Tarjan's Algorithm of finding SCCs by relaxing the conditions of judging connected edges. This makes the Tarjan's Algorithm suitable to search for WCCs.

Our algorithm computes how many weakly connected components are in the graph and records the belonging component of every node. Using the recordings, we can assign the same mutex *id* to the directed edges in the graph whose begin node and end node belong to the same component. As each directed edge is related to a thread, we add new statements of manipulating mutexes into programs. In this way, the coarse-grained programming model is built.

We use an adjacency matrix to store the graph and represent the main part of WCC computing coded in C. Our core function *DFS_modify* is a depth first search. Array *stack* simulates a stack to record traversing information about nodes. Boolean array *instack* marks whether the node is in *stack* or not. Array *visit* records the unique time stamp when the node is visited and array *low* stores the oldest time stamp looking back from the node. Variable *t* represents the time and *count* records the

Prepared using smrauth.cls
DOI: 10.1002/smr

component *ID*. We use array *belong* to declare the relationship between the node and the connected component.

```
void DFS_modify(int i)
{
    visit[i] = low[i] = ++t;
    stack[top++] = i;
    instack[i] = true;

    for(int j=1; j<=NODEN; ++j)
        if(adj[i][j]||adj[j][i])
        {
            if(!visit[j])
                DFS_modify(j);

            if(instack[j])
                low[i] = min(low[i], low[j]);
        }

    if(visit[i] == low[i])
    {
        ++count;
        int j;
        do
        {
            j = stack[--top];
            instack[j] = false;
            belong[j] = count;
        } while (j != i);
    }
}
```

As the core function *DFS_modify* has been defined, we call it in a loop in function *Tarjan_modify*. Finally, we can learn the number of connected components from *count* and the belonging connected components of the nodes from array $belong$.

```
void Tarjan_modify()
{
    count = 0;
    t = 0;
    memset(visit, 0, sizeof(visit));
    for (int i=1; i<=NODEN; i++)
        if (!visit[i])
            DFS_modify(i);
}
```

As depicted in Fig.21, we acknowledge that in order to support coarse-grained programming mutex *m0* should be shared by *Thread 1*, *Thread 2* and *Thread 3*. While mutex *m1* is assigned to *Thread 4* and *Thread 5*. As a consequence, we can update our program as below.

```
void *ThreadFunc1(void *arg){
   if (A < 100){
     return NULL;
   }
   pthread_mutex_lock(&m0);
   B = B + 100;
   A = A - 100;
   pthread_mutex_unlock(&m0);
   return NULL;
}

void *ThreadFunc2(void *arg){
   if (B < 100){
     return NULL;
   }
   pthread_mutex_lock(&m0);
   C = C + 100;
   B = B - 100;
   pthread_mutex_unlock(&m0);
   return NULL;
```


DOI: 10.1002/smr

```
}

void *ThreadFunc3(void *arg){
    if (C < 100){
      return NULL;
    }
    pthread_mutex_lock(&m0);
    B = B + 100;
    C = C - 100;
    pthread_mutex_unlock(&m0);
    return NULL;
}

void *ThreadFunc4(void *arg){
    if (C < 100){
      return NULL;
    }
    pthread_mutex_lock(&m1);
    A = A + 100;
    C = C - 100;
    pthread_mutex_unlock(&m1);
    return NULL;
}

void *ThreadFunc5(void *arg){
    if (D < 100){
      return NULL;
    }
    pthread_mutex_lock(&m1);
    E = E + 100;
    D = D - 100;
    pthread_mutex_unlock(&m1);
    return NULL;
}
```

10.2. Verification

In this subsection, we consider the verification of the correctness of bank transfer in our example program running in Pthreads and Dthreads, as well as analyze the evidence of coarse granularity in our Pthreads model performs better then fine granularity in Dthreads model. The correctness means that all the transfers have been taken and no money is lost. As we have constructed the models in PAT, the following assertion is to describe this situation.

Property 6: Correctness of Bank Transfer

#define goal1() $A == 100$;

#define goal2() $B == 200$;

#define goal3() $C == 100$;

#define goal4() $D == 0$;

#define goal5() $E == 100$;

#assert PthreadsTransferNew() $|=<>$ (*goal1*&&*goal2*&&*goal3*&&*goal4*&&*goal5*);

#assert DthreadsTransferNew() $|=<>$ (*goal1*&&*goal2*&&*goal3*&&*goal4*&&*goal5*);

As depicted in Fig.20, we need to ensure all the transfers have worked in all possible interleaving or parallel running of threads. As Linear Temporal Logic (LTL) is supported by PAT, $|=<>$ here means that the model on the left side will eventually satisfy the boolean formula on the right side. Fig. 22 displays the verification results of correctness of bank transfer in Pthreads and Dthreads. The assertions of *PthreadsTransferNew()* and *DthreadsTransferNew()* are valid, indicating that all the bank transfers are performed successfully.

We need to check whether our strategy of coarse-grained modeling worked in our Pthreads model. Checking the state of the mutex can indicate whether the threads are paralleled. We use

DOI: 10.1002/smr

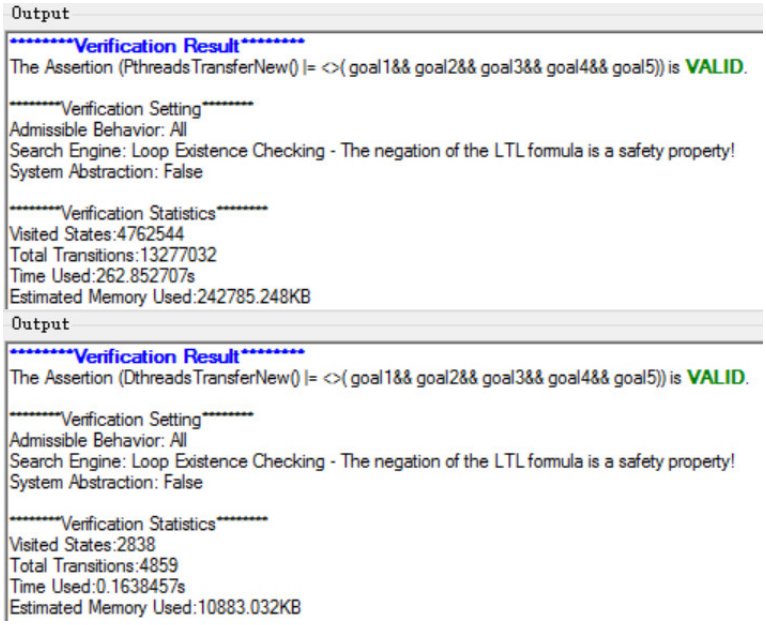


Figure 22. Result of Correctness of Bank Transfer Verification of Pthreads and Dthreads Models

this clue because if a mutex is occupied by a thread this means that a bank transfer is processing. Meanwhile, if two mutexes are owned at the same time, we can deduce that two threads are parallelled with each other without influencing each other. This also represents that coarse-grained programming helps to reduce costs of time. The assertions in PAT are illustrated as below.

Property 7: Granularity Check: the First Step

#define isparalleled MutexState[m0] == busy&&MutexState[m1] == busy;

#assert PthreadsTransferNew() reaches isparalleled;

#assert DthreadsTransferNew() reaches isparalleled;

Fig. 23 shows the verification results of first granularity check in Pthreads and Dthreads for the new bank transfer program. For *PthreadsTransferNew()*, the valid result describes that there is a situation that coarse-grained programming benefits to time reducing, which provides evidence for supporting scalability of Pthreads with coarse-grained programming. The invalid result for *DthreadsTransferNew()* suggests that the modified version of the program did not influence Dthreads, as Dthreads supports fine granularity. Based on the above analysis, we can explore more properties about granularity of our models. As the previous result indicates that two bank transfers were processed simultaneously in our Pthreads model. We want to determine whether this happens every time the model runs. For Dthreads, a much more indicative view of verifying the fine granularity of Dthreads is provided. The related assertions are list as below.

Property 8: Granularity Check: the Second Step

#assert PthreadsTransferNew() | =<> isparalleled;

#assert DthreadsTransferNew() | =<> !isparalleled;

The results of verification are shown in Fig.24. *PthreadsTransferNew()* failed to pass the assertion, which means that simultaneous bank transferring will not happen in all the running of the model. It is possible because different interleavings of threads can affect the results. Even though we can still claim that coarse-grained programming generally contributes to reducing costs of time in Pthreads.

DOI: 10.1002/smr

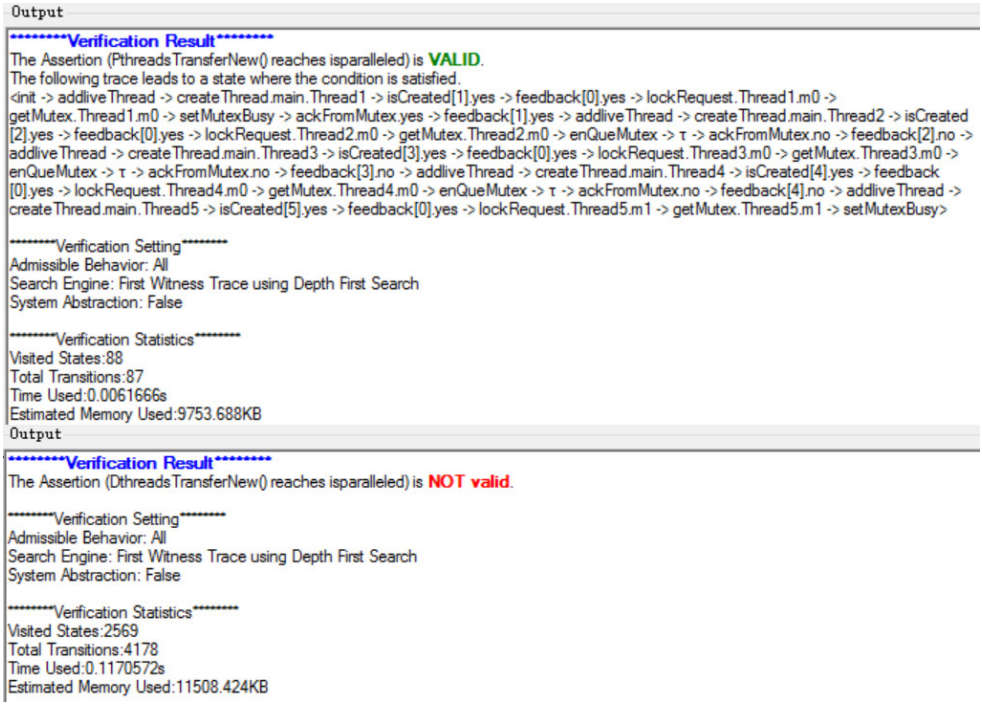


Figure 23. Result of First Step Granularity Check Verification of Pthreads and Dthreads Models

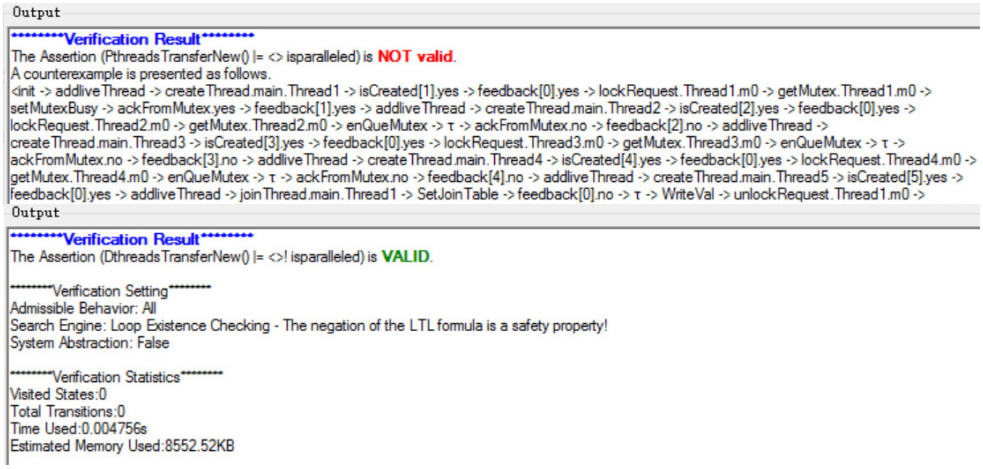


Figure 24. Result of Second Step Granularity Check Verification of Pthreads and Dthreads Models

.

The valid verification result of *DthreadsTransferNew()* displays that due to fine-grained feature in Dthreads all the bank transfers can not be taken simultaneously in all the situations.

After verifying of our related Pthreads and Dthreads model, we have found out that Dthreads keeps its fine-grained feature while Pthreads benefits from flexible mutex assignment to reduce the cost of time. It indicates that it is possible to take advantage of scalability of Pthreads while avoiding data race by using special programming model.

DOI: 10.1002/smr

11. CONCLUSION AND FUTURE WORK

In this paper we have conducted a comparative formal modelling and verification for Pthreads and Dthreads. We have formalized the thread API functions of Pthreads and Dthreads using the CSP process algebra, as well as the four illustrative programs. We have also transformed C code into the models automatically that are recognized by the model checker PAT and employed assertions to specify and verify three key concurrency properties (deadlocks, data races and race conditions) for these models. Our comparative modelling and verification of Pthreads and Dthreads shows that Dthreads is better than Pthreads on eliminating data races and preventing deadlocks. Additionally, their performances are similar in terms of their incapability of dealing with race conditions. We also have extended our modelling by covering the synchronization operations in Liu *et al.*'s work. Then two new programs are introduced to test our new API functions and figure out that Dthreads can not prevent all the deadlocks, but it indeed performs better than Pthreads for some special deadlock programs, which satisfy four necessary conditions of deadlock. Considering limited scalability of Dthreads, we have introduced a new programming model to support coarse granularity in bank transfer, which turns out efficient in Pthreads.

As for future work, we would conduct further comparative verification on Pthreads and Dthreads. We would like to improve the robustness of our system by taking into account more components, e.g. those that manage the error situation in programs. We would also like to consider more APIs so as to extend the functionality further, so as to model more multi-threaded programs. We would also like to find other methods to check the complexity of modelling and improve the efficiency of our conversion method from C code to PAT code. We also want to explore more properties for the comparative study between Pthreads and Dthreads.

Acknowledgement. This work was partly supported by the Danish National Research Foundation and the National Natural Science Foundation of China (Grant No. 61361136002) for the Danish-Chinese Center for Cyber Physical Systems. It was also supported by National Natural Science Foundation of China (Grant No. 61321064) and Shanghai Collaborative Innovation Center of Trustworthy Software for Internet of Things (No. ZF1213).

REFERENCES

1. Liu T, Curtsinger C, Berger ED. Dthreads: efficient deterministic multithreading. *SOSP*, 2011; 327–336.
2. Fei Y, Zhu H, Wu X, Fang H. Comparative modeling and verification of Pthreads and Dthreads. *17th IEEE International Symposium on High Assurance Systems Engineering, HASE 2016, Orlando, FL, USA, January 7-9, 2016*, 2016; 132–140.
3. Xu J, Zhang Z, Chan WK, Tse TH, Li S. A general noise-reduction framework for fault localization of java programs. *Information & Software Technology* 2013; **55**(5):880–896.
4. Flanagan C, Freund SN. Fasttrack: efficient and precise dynamic race detection. *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009, Dublin, Ireland, June 15-21, 2009*, 2009; 121–133.
5. Agarwal R, Wang L, Stoller SD. Detecting potential deadlocks with static analysis and run-time monitoring. *Hardware and Software Verification and Testing, First International Haifa Verification Conference, Haifa, Israel, November 13-16, 2005, Revised Selected Papers*, 2005; 191–207.
6. Bensalem S, Havelund K. Dynamic deadlock analysis of multi-threaded programs. *Hardware and Software Verification and Testing, First International Haifa Verification Conference, Haifa, Israel, November 13-16, 2005, Revised Selected Papers*, 2005; 208–223.
7. Bishop M, Dilger M. Checking for race conditions in file accesses. *Computing Systems* 1996; **2**(2):131–152.
8. Uppuluri P, Joshi U, Ray A. Preventing race condition attacks on file-systems. *Proceedings of the 2005 ACM Symposium on Applied Computing (SAC), Santa Fe, New Mexico, USA, March 13-17, 2005*, 2005; 346–353.
9. Deshmukh JV, Emerson EA, Sankaranarayanan S. Symbolic modular deadlock analysis. *Autom. Softw. Eng.* 2011; **18**(3-4):325–362.
10. Luo ZD, Das R, Qi Y. Multicore SDK: A practical and efficient deadlock detector for real-world applications. *Fourth IEEE International Conference on Software Testing, Verification and Validation, ICST 2011, Berlin, Germany, March 21-25, 2011*, 2011; 309–318.
11. Voung JW, Jhala R, Lerner S. RELAY: static race detection on millions of lines of code. *Proceedings of the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2007, Dubrovnik, Croatia, September 3-7, 2007*, 2007; 205–214.
12. Savage S, Burrows M, Nelson G, Sobalvarro P, Anderson TE. Eraser: A dynamic data race detector for multithreaded programs. *ACM Trans. Comput. Syst.* 1997; **15**(4):391–411.

DOI: 10.1002/smr

13. Ilgun K. USTAT: a real-time intrusion detection system for UNIX. *1993 IEEE Computer Society Symposium on Research in Security and Privacy, Oakland, CA, USA, May 24-26, 1993*, 1993; 16–28.
14. Tsyrklevich E, Yee B. Dynamic detection and prevention of race conditions in file accesses. *Proceedings of the 12th USENIX Security Symposium, Washington, D.C., USA, August 4-8, 2003*, 2003.
15. Lafortune S, Wang Y, Reveliotis SA. Eliminating concurrency bugs in multithreaded software: An approach based on control of petri nets. *Petri Nets*, 2013; 21–28.
16. Edwards SA, Vasudevan N, Tardieu O. Programming shared memory multiprocessors with deterministic message-passing concurrency: Compiling shim to pthreads. *DATE*, 2008; 1498–1503.
17. Berger ED, Yang T, Liu T, Novark G. Grace: safe multithreaded programming for C/C++. *OOPSLA*, 2009; 81–96.
18. IEEE. IEEE POSIX 1003.1c standard. URL `http://standards.ieee.org/findstds/interps/1003-1c-95_int/index.html`.
19. Johnson R. Pthread win-32: Level of standards conformance. URL `http://www.sourceware.org/pthreads-win32/conformance.html`.
20. Lucia B, Ceze L. Finding concurrency bugs with context-aware communication graphs. *42st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-42 2009), December 12-16, 2009, New York, New York, USA*, 2009; 553–563.
21. Huang Y, He J, Zhu H, Zhao Y, Shi J, Qin S. Semantic theories of programs with nested interrupts. *Frontiers of Computer Science* 2015; **9**(3):331–345.
22. Lu K, Zhou X, Wang X, Bergan T, Chen C. An efficient and flexible deterministic framework for multithreaded programs. *J. Comput. Sci. Technol.* 2015; **30**(1):42–56.
23. Gradara S, Santone A, Villani ML, Vaglini G. Model checking multithreaded programs by means of reduced models. *Electr. Notes Theor. Comput. Sci.* 2004; **110**:55–74.
24. Li J, Hei D, Yan L. Correctness analysis based on testing and checking for OpenMP programs. *Fourth ChinaGrid Annual Conference, ChinaGrid 2009, Yantai, Shandong, China, 21-22 August, 2009*, 2009; 210–215.
25. Yang Y, Chen X, Gopalakrishnan G. Inspect: A runtime model checker for multithreaded C programs. *Technical Report* 2008.
26. Zhang D, Bosnacki D, van den Brand M, Huizing C, Kuiper R, Jacobs B, Wijs A. Verification of atomicity preservation in model-to-code transformations using generic java code. *MODELSWARD 2016 - Proceedings of the 4rd International Conference on Model-Driven Engineering and Software Development, Rome, Italy, 19-21 February, 2016.*, 2016; 578–588.
27. PAT. PAT: Process analysis toolkit. URL `http://pat.comp.nus.edu.sg/`.
28. Sun J, Liu Y, Dong JS. Model checking CSP revisited: Introducing a process analysis toolkit. *Leveraging Applications of Formal Methods, Verification and Validation, Third International Symposium, ISoLA 2008, Porto Sani, Greece, October 13-15, 2008. Proceedings*, 2008; 307–322.
29. Brookes SD, Hoare CAR, Roscoe AW. A theory of Communicating Sequential Processes. *Journal of ACM* 1984; **31**(7):560–599.
30. Hoare CAR. *Communicating Sequential Processes*. Prentice-Hall, 1985.
31. Ferreira JF, Gherghina C, He G, Qin S, Chin W. Automated verification of the freertos scheduler in hip/sleek. *STTT* 2014; **16**(4):381–397.
32. Qin S, He G, Luo C, Chin W, Yang H. Automatically refining partial specifications for heap-manipulating programs. *Sci. Comput. Program.* 2014; **82**:56–76.
33. Roscoe AW. *The theory and practice of concurrency*. 1998.
34. Roscoe A. *Understanding Concurrent Systems*. 2010.
35. Lowe G, Roscoe AW. Using CSP to detect errors in the TMN protocol. *IEEE Trans. Software Eng.* 1997; **23**(10):659–669.
36. Kapoor HK. A process algebraic view of latency-insensitive systems. *IEEE Trans. Computers* 2009; **58**(7):931–944.
37. Regehr J. Race condition vs. data race. `http://blog.regehr.org/archives/490`. Accessed May 22, 2017.
38. Sun J, Liu Y, Dong JS, Liu Y, Shi L, André É. Modeling and verifying hierarchical real-time systems using stateful timed CSP. *ACM Trans. Softw. Eng. Methodol.* 2013; **22**(1):3.
39. Liu Y, Sun J, Dong JS. Developing model checkers using PAT. *ATVA*, 2010; 371–377.
40. Si Y, Sun J, Liu Y, Dong JS, Pang J, Zhang SJ, Yang X. Model checking with fairness assumptions using PAT. *Frontiers of Computer Science* 2014; **8**(1):1–16.
41. Erickson J, Musuvathi M, Burckhardt S, Olynyk K. Effective data-race detection for the kernel. *9th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2010, October 4-6, 2010, Vancouver, BC, Canada, Proceedings*, 2010; 151–162.

APPENDIX

In the verification of our considered examples, we find out that deadlocks and data races both exit in Pthreads. At the same time, both of them can not avoid race conditions. Fig.25 illustrates the trace that can cause a deadlock in Pthreads model. Meanwhile, Fig.26 gives the data race trace in Pthreads model, which makes the assertion fail.

DOI: 10.1002/smr

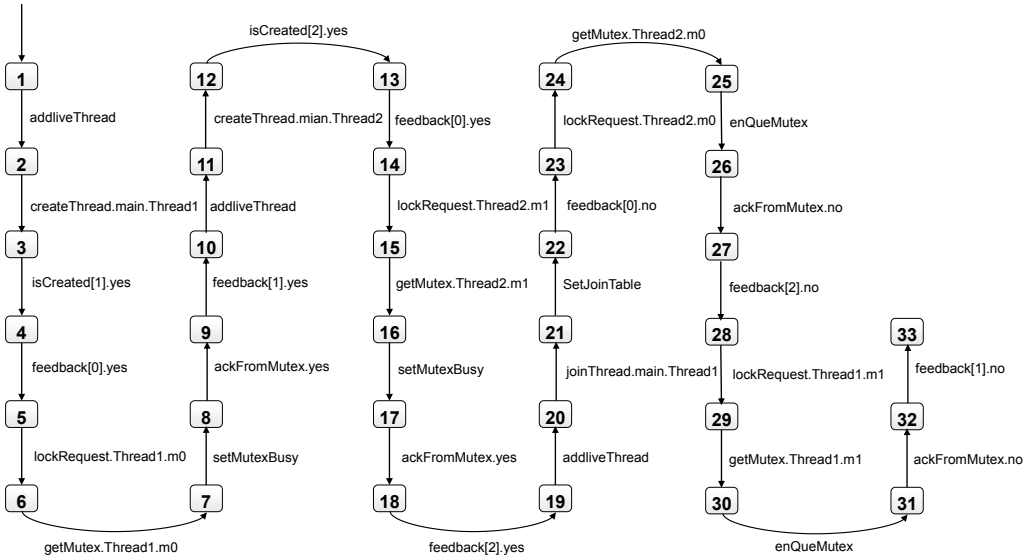


Figure 25. Simulation Result of Deadlock Freedom Trace of Pthreads Model

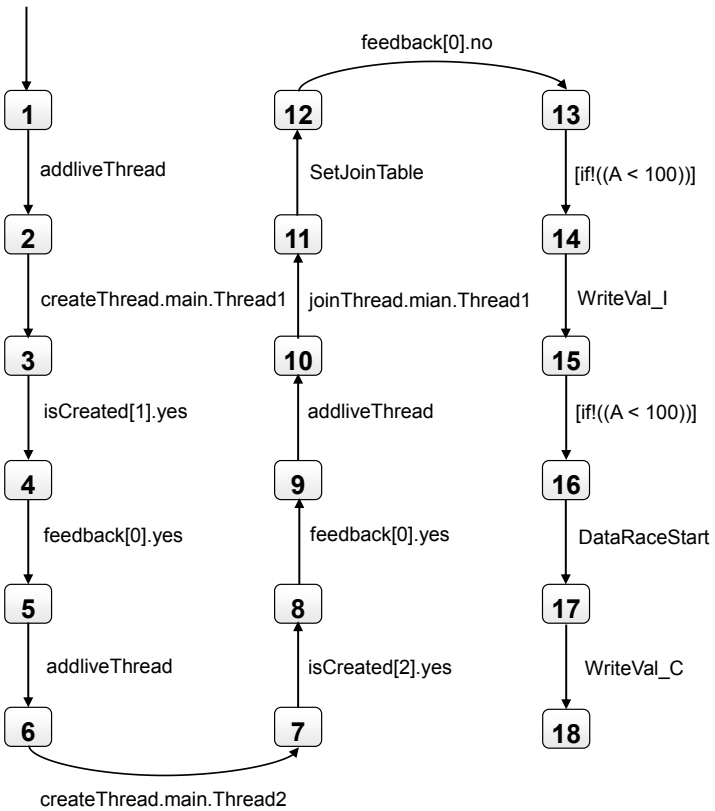


Figure 26. Simulation Result of Data Race Trace of Pthreads Model

DOI: 10.1002/smr