Formal Verification of Process Communications in Operational Flight Program for a Small-Scale Unmanned Helicopter

Dong-Ah Lee[1], **Junbeom Yoo**[2] and **Doo-Hyun Kim**[3]

[1, 2] School of Computer Science and Engineering
Konkuk University
Seoul, Republic of Korea
e-mail: ldalove@konkuk.ac.kr (corresponding author), [2] jbyoo@konkuk.ac.kr

[3] School of Internet and Multimedia Engineering
Konkuk University
Seoul, Republic of Korea
e-mail: [3] doohyun@konkuk.ac.kr

Abstract

Formal verification plays an important role in demonstrating safety and correctness of safety-critical systems such as airplanes and helicopters. Small-scale unmanned helicopters have been increasingly developed and deployed for various scientific, commercial and defense applications. The HELISCOPE project is aiming to develop an unmanned helicopter and its on-flight embedded computing system for navigation and real-time transmission of the motion video using wireless communication schemes. This paper introduces our experience on the formal verification of OFP (Operational Flight Program) in the HELISCOPE project. The OFP provides real-time controls with various sensors and actuators, and should be sufficiently verified through formal verification techniques. We focused on the formal verification of process communications between four sensing processes and one controller to access a critical section of shared memory area mutually exclusively.

1 Introduction

HELISCOPE [1] project is to develop on-flight computing system, embedded S/W and related services for unmanned helicopter that shall be used for disaster response and recovery. This project is aiming to develop an unmanned helicopter and its on-flight embedded computing system for navigation and real-time transmission of the motion video using wireless communication schemes. OFP (Operational Flight Program) [2] is developed as subpart of HELISCOPE project. It is a control program which provides real-time controls with various sensors and actuators equipped in the helicopter.

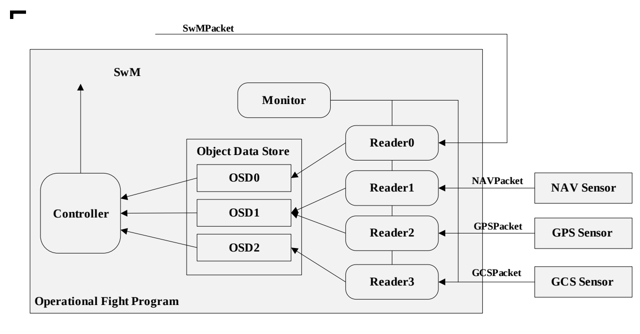
This paper specified above processes and their communications formally with Promela (Protocol Meta Language), and performed formal verification (model checking) using SPIN model checker [3]. First we are focusing on the correct communications between 4 reading processes and controller process through the shared memory area. We also consider the correctness of semaphore operation performed by monitor process in the paper. The OFP has a real-time feature too. But some real-time features of controller make us consider about other formal verification techniques such as using Statecharts[4] or UPPAAL[5]. We don't care of such a timing constraint in this paper.

In section 2, the OFP and SPIN are introduced briefly as related works and in section 3, formalization of OFP in Promela is described. In section 4, verification results with a model formalized in section 3 are analyzed and in section 5, we will conclude.

2 Related works

2.1 Operational Flight Program

The OFP is developed as a subpart of the HELISCOPE project and it is based on the well-known TMO scheme [6]. OFP support the unmanned helicopter's navigation that is done by commands on flight mode from GCS (Ground Control System). Figure 1 shows an overview of communications between processes in the OFP. We described it from aspect of the formal verification, which are pertinent to our discussion.



ISBN 978-979-16955-1-0

Figure 1: An overview of process communications in the OFP

The organization of the OFP is as follows: *Reader0* is a process collecting real-time operational information (*SwMPacket*) of the helicopter, while *Reader1* reads packets (*NAVPacket*) containing navigation information. *Reader2* read GPS data from a GPS equipped in the helicopter, and *Reader3* is a process collecting information from GCS (Ground Control Station). *Controller* is a main controller of the OFP, which reads data stored in a shared memory area ('Object Data Store' in Figure 1) and controls actuators equipped in the helicopter through *SwMPacket* command. The OFP has another process *Monitor* besides these five processes. It provides 4 reading processes with semaphore facility.

Some of reader process accesses the same shared data area to refer a data or write a data from sensors. And the controller process accesses all shared data area to compute next control data. For the same data in the shared area, accessing by Controller or four reader processes (*Reader0*, *Reader1*, *Reader2* and *Reader3*) should be performed mutually exclusively. Because if controller processes accesses to read a data area when another process is writing a data on same area, then the process which is reading it will have a wrong value expecting and the controller computes next value to move the helicopter with wrong values. The OFP also should guarantee for correctness and deadlock-freeness of semaphore facility, because the readers should get a data which is a source for computation of controller from sensors on time. If it is impossible to get a data on time, then the controller cannot

2.2 Model Checking using SPIN

SPIN is a formal verification system that supports the design and verification of distributed software systems. SPIN models consist of three types of objects: process, message channels and variables. Processes specify behavior, channels and global variables define the environment in which the processes run. Programs are implemented in Promela language which is quite similar to an ordinary programming language.

In 2001, NASA Ames Research Center in USA applies formal analysis on a Space Craft Controller using SPIN [7]. They formally analyzed a multi-threaded plan execution module. The plan execution module is one component of NASA's New Millennium Remote Agent [8], which is artificial intelligence based spacecraft control system architecture. The results are that they found 5 previously undiscovered concurrency errors were identified. They reported the results to development team, and according to the team the effort had a major impact.

3 Formal verification using SPIN model checker

3.1 Overview

Our discussion of the Promela model of the OFP focuses on communications between processes and semaphore management. Figure 2 describes the communications between 4 reading processes (*Reader*N) and controller process (*Controller*) through the global data area (*SharedVar*N). It also shows semaphore operations on the 4 processes by monitor process (*Monitor*).

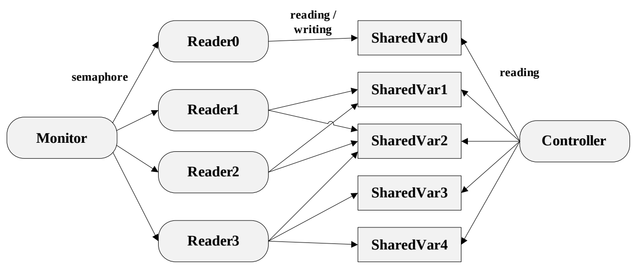


Figure 2: A schema of Promela Model in SPIN

3.2 Formalization in Promela

3.2.1 Share variable & semaphores

The three ODS may have multiple variables. The variables have variety types like character or double, and some of them are a large array. But they are too heavy to be represented in Promela. It may cause state explosion problem in model checking, and our model doesn't need to know the detail information. Therefore we abstract a property that is for checking whether it is stored or not from the ODS. Figure 3 shows the abstracted data type of ODS in Promela. It is defined as byte variable, and it can only have a value from 0 to 255.

```
byte sharedVar0;
byte sharedVar1;
byte sharedVar2;
byte sharedVar3;
byte sharedVar4;

bool semaphore0 = false;
bool semaphore1 = false;
bool semaphore2 = false;
bool semaphore3 = false;
```

Figure 3: Definition of shared variable & semaphore

In Figure 3, there are other variables for semaphore. These variables are set to `true` when the monitor process sense that a sensor sends a data. Reader processes which had been waiting for setting semaphore to true can run to receive the data, and they are set to `false` when a reader process finish writing the data on shared data area.

3.2.2 Data access operation

All processes of OFP can access ODS using functions that are defined the classical operations like *sharedVariable_set()*, *sharedvariable_get()*, etc.. We defined them as inlines (a stylized version of a macro) in Promela. An inline definition works much like a preprocessor macro, in the sense that it just defines a replacement text for a symbolic name, possibly with parameters.

```
inline accessGlobalData0() { … }
inline accessGlobalData1() { … }
inline accessGlobalData2() { … }
inline accessGlobalData3() { … }
inline accessGlobalData4() { … }
```

Figure 4: Functions for access to ODS

When processes try to access ODS calling the classical operations, the operations use mutex variable. If a process locks a mutex variable, then other process should waits to lock the mutex variable until the mutex variable is unlocked. We defined variables using bit data type for mutex variables and lock and unlock operations using an inline. It is represented in Figure 5.

```
bool mutex_0;
……
bool mutex_4;

inline mutex_lock(mutex) {
       atomic {
              if
              ::mutex == false ->
                            mutex = true
              fi
       }
}
inline mutex_unlock(mutex) {
       atomic {
              if
              ::mutex == true ->
                            mutex = false
              fi;
       }
}
```

Figure 5: Lock & unlock functions with mutex variable

These inline functions, Figure 4 and Figure 5, don't run itself. They are only called by other processes like *Controller* and *Reader*. *Controller* and *Reader* call *accessGlobalDataN()*. *accessGlobalDataN()* calls the *mutex_lock()* and *unlock_mutex()* operation to read or write on global data area. In addition, *atomic* in Promela indicates that the sequence is to be executed as one indivisible unit, non-interleaved with other processes.

3.2.3 Sensors

Sensors, which generate or receive information, are main devices to control the helicopter. But they are too complex to implement in Promela, and hence we tried to find a convenient way to represent them. Our solution is that the 4 sensors are defined as a process that can generate all data, and moreover the process only generates a data identified by reader process.

We defined an inline to simulate that sensors generate data randomly. The model focuses on the communication between processes, and so we modeled the sensors to do every possible operation. None of sensor can send a message to the system, and one or more sensors can send a message at once. We tried to implements the operation using random functions in Promela, but there is no predefined random number generation function unfortunately. So we defined another inline to work like random functions (see Figures 6).

```
bit sensor[4];

inline Sensors()
{
       if
       ::skip -> sensor[0] = false
       ::skip -> sensor[0] = true
       fi;
       if
       ::skip -> sensor[1] = false
       ::skip -> sensor[1] = true
       fi;
       if
       ::skip -> sensor[2] = false
       ::skip -> sensor[2] = true
       fi;
       if
       ::skip -> sensor[3] = false
       ::skip -> sensor[3] = true
       fi
}
```

Figure 6: Sensor operation

3.2.4 Processes

Monitor

Monitor process in the OFP monitors the serial ports. If the monitor senses a serial port that is sending data, then it makes reader processes work to get the data from the serial port.

A model in Promela does exactly same work with monitor in OFP. First it checks serial ports. Then it makes Reader processes works each time for checking serial ports. When the monitor makes readers work it posts semaphore variables. We decided to model semaphore variables and functions as channels. Channels have some of the same properties as them: A receiver should wait to receive a message through a channel until it receives the message. The posing is realized by the statement:

```
sema_ch0!true
```

monitor process has 4 channels connected with 4 *Reader* processes. It checks data generated by the inline function *Sensors()*. If a data is set to true, then *monitor* sends a message to *Reader* that is supposed to receive the data through channel. On the other hand, if the data isn't set, then it will skip sending a message and the *Reader* waits to receive a data. We defined a channel as an asynchronous

ISBN 978-979-16955-1-0

with only one buffer. Only a data can be sent at once. Figure 7 shows *monitor* process

```
proctype monitor
(chan sema_ch0,sema_ch1,
      sema_ch2, sema_ch3)
{
      do
      ::skip ->
            if
            ::sensor[0] == true ->
                        sema_ch0!true
            ::sensor[0] == false ->
                        skip
            fi;
            if
            ...........
            fi;
            if
            ...........
            fi;
            if
            ...........
            fi;
            Sensors()
      od
}
```

Figure 7: Monitor process

Reader

There are 4 reader processes in the OFP. They read a data sent from sensors through serial port and write the data on a shared data area, ODS. Figure 8 is a part of one of *reader* processes in Promela. A *reader* process has a channel connected with *monitor*. The *monitor* senses a data, and it sends a message for semaphore. *reader* waits the message to receive data from a sensor. The waiting is realized by the statement:

```
sema_ch?semaphoreN ->
```

This statement make the process be blocked. It runs when *monitor* sends a data through same channel.

```
proctype reader3(chan sema_ch)
{
      do
      ::sema_ch?semaphore3 ->
            sensor[3] = false;
            if
            ::skip ->
                  AccessGlobalData2();
                  AccessGlobalData4()
            ::skip ->
                  ......
            fi;
            semaphore3 = false
      od
}
```

Figure 8: reader3 process

After receiving a data from sensor and writing the data on shared data area, *reader* process set the `sensor` and `semaphore` to false. A meaning of setting sensor is that the reader process finished receiving, and a meaning of setting semaphore is that reader process finished running.

Basically, the other processes have similar procedure like *reader3*. But they access data different order and times, because each *reader* has different properties and it needs to compute data received from sensors differently. For example, reader0 accesses only *sharedVar0*, and *reader1* and *reader2* access *sharedVar1* and *sharedVar2*. The *reader3*, Figure 8, accesses *sharedVar2*, *sharedVar3* and *sharedVar4*. Here is a feature we are verifying that the access to same variable between *reader1~3*. Each reader process has specific order and times to access them.

Controller

The main controller of OFP controls a helicopter with a data computed with the data stored by readers. It has a running cycle and deadline to run in actual program. It is important that the controller computes a control data in deadline, because if a helicopter doesn't change its flight mode or flying direction on time, then it can be fall in dangerous. But we don't care of the cycle or deadline in this paper. We only consider whether there is any errors or faults in communication with readers.

```
proctype controller()
{
      do
      :: skip ->
            AccessGlobalData0();
            AccessGlobalData2();
            if
            ::skip->AccessGlobalData1()
            ......
            fi;
            ......
      od
}
```

Figure 9: Controller process

Figure 9 shows a part of *controller* modeled in Promela. The *controller* may access every shared data area, ODS, to compute next value for control a helicopter. We modeled the *controller* without timing constraints, and hence this process can run every situation it needs to run. It has very complicated access order, and it accesses ODS many times. Almost operations of access aren't indicated in this paper. But it is reflected a following of the OFP.

4 Verification and results

With the Promela model implemented in section 3.2, we perform SPIN model checking against these three properties below:

(1) The process *monitor*'s Semaphores on four reading processes should function correctly.

(2) Two processes *reader1*, *reader2* and *reader3* should access the same global data mutually exclusively.

(3) Reading process *controller* and four writing processes should be mutually exclusive.

We performed the SPIN simulation as described in Figure 10 below in order to make confirm the correctness of our modeling – Semaphore and shared variable accessing. The calling procedure *AccessGlobalData*N*()* in the simulation shows communications between the global shared data area and four reading processes. It is a model of mutex for the shared data variables (mutex_0, mutex_1, mutex_2, mutex_3 and mutex_4). Messages passing out of *monitor* in the simulation also simulate the Semaphore operations provided by the Monitor process too. After guaranteeing its correctness, we performed the SPIN model checking against the three properties.

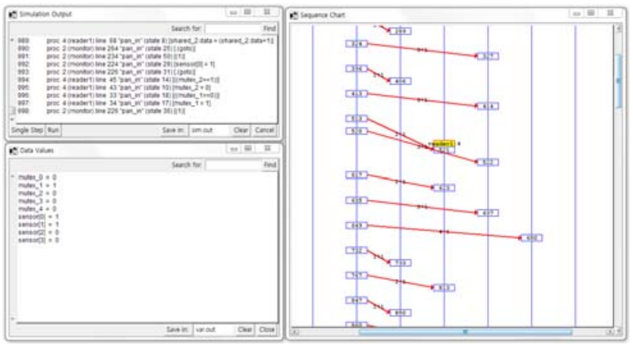


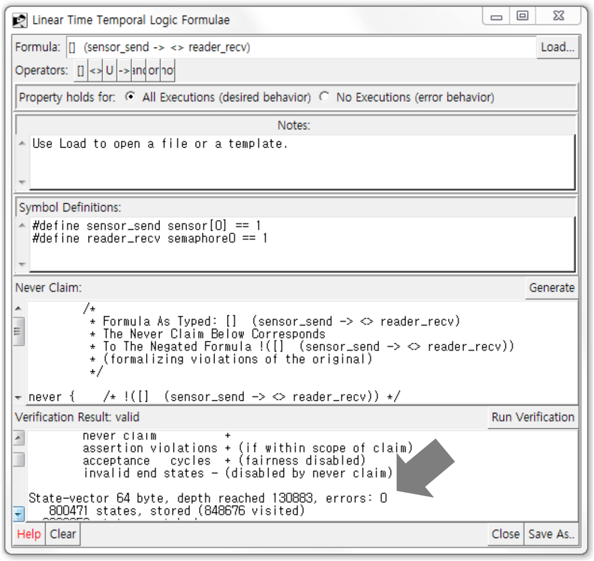
Figure 10: A screen-dump of SPIN simulation

The property (1) is refined into LTL property below:

```
[] ( sensor_send -> <> read_recv )
#define sensor_send sensor[0] == true
#deinfe reader_recv reader0.sema == true
```

This property states that "*in all stats, if* `sensor_send` *holds, then eventually either* `read_recv` *will hold*". If a sensor tries to send a data to the system, Monitor senses it first. And the Monitor posts a semaphore that makes that Reader process receives a data. The Monitor manages 4 Reader process with posting 4 semaphores like that. In this procedure, we verified whether the Monitor can manage the 4 Readers correctly. Monitor should post correct semaphore and Reader should run when its semaphore is posted.

Figure 11 shows the verification result with LTL property. There is LTL formula on top of window, and a predicates of `sensor_send` and `reader_recv` on Symbol Definitions section. We verified it with three properties more. The three properties are about `sensor[1]~[3]` and `reader1~3`. The results of four properties are all satisfied. We confirmed that the process *monitor* manage four *reader* processes correctly.



Linear Time Temporal Logic Formulae
Load...
Operators: [] <> U -> and or not
Generate
Run Verification
Help
Clear
Close
Save As..

Formula: [] (sensor_send -> <> reader_recv)

Property holds for: [✓ All Executions (desired behavior)] No Executions (error behavior)

Notes:

```
Use Load to open a file or a template.
```

Symbol Definitions:

```
#define sensor_send sensor[0] == 1
#define reader_recv semaphore0 == 1
```

Never Claim:

```
        /*
         * Formula As Typed: []  (sensor_send -> <> reader_recv)
         * The Never Claim Below Corresponds
         * To The Negated Formula !([]  (sensor_send -> <> reader_recv))
         * (formalizing violations of the original)
         */

never {    /* !([]  (sensor_send -> <> reader_recv)) */
```

Verification Result: valid

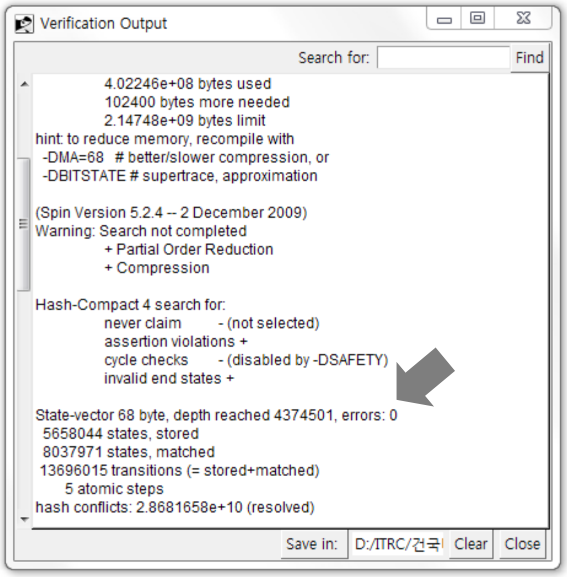
```
        never claim           +
        assertion violations + (if within scope of claim)
        acceptance   cycles  + (fairness disabled)
        invalid end states - (disabled by never claim)

State-vector 64 byte, depth reached 130883, errors: 0
   800471 states, stored (848676 visited)
```

Figure 11: Verification result with LTL property

We defined a process to verify properties (2) and (3) in Figure 12. The meaning that a mutex variable becomes over 1 is over 1 processes access a critical section using the mutex variable. It causes a problem that the processes write a data on shared data area at the same time, or the processes refer a wrong data. This process is defined `active`, because it always runs to check the variable.

```
active proctype assert_monitor()
{
       assert( (mutex_0 != 2) &&
       (mutex_1 != 2)&&(mutex_2 != 2) &&
       (mutex_3 != 2)&&(mutex_4 != 2) )
}
```

Figure 12: Process to verify mutexes



```
          4.02246e+08 bytes used
          102400 bytes more needed
          2.14748e+09 bytes limit
hint: to reduce memory, recompile with
  -DMA=68   # better/slower compression, or
  -DBITSTATE # supertrace, approximation

(Spin Version 5.2.4 -- 2 December 2009)
Warning: Search not completed
          + Partial Order Reduction
          + Compression

Hash-Compact 4 search for:
          never claim        - (not selected)
          assertion violations +
          cycle checks       - (disabled by -DSAFETY)
          invalid end states +

State-vector 68 byte, depth reached 4374501, errors: 0
 5658044 states, stored
 8037971 states, matched
 13696015 transitions (= stored+matched)
     5 atomic steps
hash conflicts: 2.8681658e+10 (resolved)
```

Figure 13: Verification result with assert

ISBN 978-979-16955-1-0

5 Conclusion

In this paper we apply formal verification on OFP using model checker SPIN. We focused on verifying process communications between four sensing processes and one controller to access a critical section of shared memory area mutually exclusively. And we also verified that managements of processes with semaphore technique. Results of verification are that there is no defect or fault about accessing shared data area and managing readers with semaphore.

It is worth to mentioning that we modeled the shared memory area (i.e. mutex) in the OFP with calling procedures. The Spin's strong merit – modeling communication protocols between independent processes through channels– made us model it in the way. It however may cause a modeling fault when combining with the other part of the OFP, Controller process. The process has strict timing scheduling and restrictions, so the difference between accessing to shared data area and calling a procedure might cause slightly different behavior of Controller. We are currently focusing on analyzing the timing-related behavior of Controller, and it may change the current model of the OFP. The timing related features of Controller, as we mentioned, may help us change formal verification techniques and tools, i.e. UPPAAL with timed automata model or Statecharts with hierarchical state machine models.

Acknowledgement

This research was supported by the MKE(Ministry of Knowledge Economy), Korea, under the ITRC(Information Technology Research Center) support program supervised by the NIPA(National IT Industry Promotion Agency). (NIPA-2010-(C1090-1031-0003), (NIPA-2010-(C1090-0903-0004))

References

[1] D. H. Kim, K. Nodir, C.H. Chang, J.G. Kim "HELISCOPE Project: Research Goal and Survey on Related Technologies", In the Proceeding of 12th IEEE International Symposium on Object / Component / Service-Oriented Real-Time Distributed Computing (ISORC), pp.112-118, Tokyo, 2009.

[2] S. G. Kim, S.H. Song, C. H. Chang, D. H. Kim, S. Hew, J. G. Kim "Design and implementation of an Operational Flight Program for an Unmanned Helicopter FCC Based on the TMO Scheme", Proceedings of the 7th IFIP WG 10.2 International Workshop on Software Technologies for Embedded and Ubiquitous Systems(SEUS), pp.1-11, Newport Beach, CA, USA, 2009.

[3] Holzmann, G. J. "The Model Checker SPIN", IEEE Transactions on Software Engineering, Vol. 23, No. 5, May 1997.

[4] D. Harel., "On Visual Formalism," *Communication of ACM*, Vo.31, 5, pp.514-530, 1988.

[5] UPPAAL, http://www.uppaal.com

[6] Kim, K.H., Kopetz, H. "A Real-Time Object Model RTO.k and an Experimental Investigation of Its Potentials", In: 18th IEEE Computer Software & Applications Conference, pp. 392–402, Los Alamitos, 1994

[7] Havelund, K. Lowry, M. Penix, J., "Formal analysis of a space-craft controller using SPIN", Software Engineering, IEEE Transactions on, Vol.27, 8, pp. 749 – 765, 2001

[8] B. Pell, E. Gat, R. Keesing, N. Muscettola, and B. Smith, "Plan Execution for Autonomous Spacecraft", In Proceedings of the International Joint Conference on Artificial Intelligence, pp. 1234-1239, Japan, 1997