A trustable software with a dynamic loop control mechanism

Nalinrat Srirajun and Pattarasinee Bhattacharayab

Department of Mathematics,
Faculty of Science,
Chulalongkorn University,
Bangkok, 10330, Thailand
E-mail: nalinrat.s@student.chula.ac.th
E-mail: pattarasinee.b@chula.ac.th
*Corresponding author

Panjai Tantasanawong

Department of Computing,
Faculty of Science,
Silpakorn University,
Sanam Chandra Palace Campus,
Rajamunka-nai Rd., Muang District,
Nakhon Pathom, 73000, Thailand
E-mail: panjai@su.ac.th

Sunyoung Han

Department of Computer Science and Engineering,
Konkuk University,
1 Hwayang-dong, Gwangjin-gu, Seoul 143-701, Korea
E-mail: syhan@konkuk.ac.kr

Abstract: Unexpected infinite loop during software deployment period is an undesirable for every organisation since it can cause critical damages in various aspects. Most of the current protection methods will perform during the software development process using verification and validation processes. However, these methods will not cover the dynamic loop control mechanism where the termination of the loop is relied on the value obtained during runtime process. Thus, many possible error values cannot be protected based on human’s limitation. Therefore, this paper proposes architecture for checking and terminating infinite loop situation during the runtime process. Additionally, the proposed mechanism can completely prevent the unlimited loop for these dynamic situations. The testing results have shown that the proposed solution provides the trustable system for every organisation.

Keywords: execution time; mean time interval; verification; validation; run-time system; monitoring and detecting; result-checking; dynamic loop control; communications.
1 Introduction

Currently, every organisation mainly drives their processes by software, both from in house or COTS. Therefore, a factor of success in all business organisations is definitely relied on software used in the business. Thus, developing software for the business organisations can be counted as a critical factor of success and must not have any hidden defects. Various software developers or software engineers tries to develop methods to protect and detect all possible defects during software development process, including software quality control mechanism. Unfortunately, there is no such mechanism can completely guarantee that there is no remaining defect over the delivered software.
In order to guarantee that software will have minor defects, or no possible defect, however processes during the software development processes must be performed; these processes are verification and validation processes. The verification process attempts to detect all functional defects before the software is implemented to users in the real life while the validation process attempts to prove that the developed software can serve all users expectations.

Under the verification mechanism, three types of errors will be detected. The first error is the simplest error which is the syntax error, such as missing of ‘)’, or ‘;’. This error is based on the syntax of each programming language. The second error is the logical error. This type of error is usually based on the logic thinking of the programmer, and most of the errors can be detected at the boundary of the condition, such as $0 \leq a \leq 7$. The last type of errors is the dynamic error, or run-time error. This error occurs while the software runs to perform its tasks. In this situation, there are various factors or causes of software malfunctions, such as I/O error, or computation error based on divided by zero case, including communication error for distributed computing system.

Based on all types of errors mentioned above, the most significant error that can cause critical damages is the run-time error because there is no guarantee that every process will be provided with valid input value; thus, there is no confirm that the software will not produced invalid computing value during the computation process. Finally, the software cannot ensure users that there is no possibility of run-time error while performing a significant task.

Every time software has a run-time error, the reliability of software will be dropped (Rosenberg et al., 1992). Moreover, the software overhead to re-perform the task is increase. Consequently, the software cannot be trusted for any critical system, such as medical organisation, or banking and financial system. Unfortunately, these organisations cannot avoid using software for their processes. Thus, this paper will propose a mechanism that can ensure users of these critical systems whenever the run-time error has potential to occur. Thus, no critical damage will have a chance to arise, organisation will be safe and software is fully trustable.

The remainder of the paper is organised as follows. In Section 2 describes literature review and background. Section 3 describes proposed method and designed architecture. Then, Section 4 will show the software architecture. The result of this design is presented in Section 5. Section 6 and Section 7 are discussion and conclusions respectively.

2 Literature review and background

The successful of every organisation over the world is relied on the quality of the implemented software, and the reliability of software is based on the number of errors that can be detected and protected during software processes.

In order to obtain qualified software, there are many factors that have to be considered during software development process which consists of analysis, design, implementing and testing phases. One protection technique is to consider the structure of software design related to the implemented parameters. This technique was proposed by Waters (1979) in checking loop programs by extending instructions in the checking loop programs using plan building method (PBM). This PBM will reconstruct the written program to be an easy form, then programmers can understand and trace for any bugs
A trustable software with a dynamic loop control mechanism

easily. However, the author did not mention that it can be applied to the recursion function that obtains a loop.

Moreover, many researchers have proposed solutions for run-time problems of software, such as Meyer and Ritchie (1967), Wasserman and Blum (1997), and Hangal and Lam (2002). The mechanism proposed by Meyer and Ritchie was analysing when a loop is running using theoretical approval. They used the concept on the primitive recursive function, bounding the running time and complexity classes. They believed this analysing can be applied to the real system.

In the year 1997, Wasserman and Blum proposed that an efficient debugger can increase the reliability of software when the run-time problem is added into a debugging module. Therefore, this research considered the result-checking, simple checkers, self-correctors on possible of input data, and boundary of time when software was executed. All these measurements were verified by formal verification method and the result had shown that the result-checking may improve the debugging tool for reliable systems.

Similar to the Wasserman et al., Hangal and Lam (2002) proposed Dynamic Invariant Detection U Checking Engine (DIDUCE) as a tool for detecting complex program errors in the debugging process. This tool focuses on various faults which are failure from some inputs, failures in long-running programs, component-based software. Moreover, the programs are tested with inputs for which the correct outputs are unknown. As a consequence of using four applications, this method can discover different types of errors, such as errors from algorithm errors, errors in inputs, and developer’s misconceptions of the APIs.

Although the run-time error was focused by many researchers, the event of changing states in the system and the execution time are also considered as other factors to uncover errors hidden in the software. This method was proposed in the year 1996 by Gergeleit et al. who designed method for testing time of running objects under the object-oriented environment. This time, called as the real time logic (RTL), was measured in the testing state and was extended as a constraint for the state change of software. The result showed that the model can detect of constraints violation. Unfortunately, this run-time checker is not the dynamic value since it cannot alter its value in the run-time environment. Therefore, Lee et al. (1998) had designed a framework for monitoring and checking (MAC) behaviour of the system using the run-time checker. However, this framework architecture, called as MAC architecture, is implemented on three techniques: filter, event recogniser, and run-time checker. As a consequence, this framework can fill in the gap occurring between static verification and testing mechanism of software development process.

The timing technique is also applied to monitor the real-time system using graph diagram by Mok and Liu in the year 1997. This diagram is drawn to detect violation during event transfer period. They implemented with Java run-time timing constraint monitor (JRTM). This method provided the shortest latency and low overhead for violation detection. However, there are some applications that can cause a long delay. Thus, Robert et al. (2008) proposed a method using automata theorem for analysing time of program’s event in run-time monitoring; the expected outcome is the detected errors of software. This prototype reveals itself to be efficient with respect to real-time operating system deployment and decreasing event overhead.

Various researches had studied behaviour of error. Groce et al. (2006) approached automate for assisting users in understanding and isolating errors based on distance.
metrics for program executions. In the year 2009, Sugaya et al., presented a monitoring method for detecting anomalies using only resource usage information on systems which modifies the kernel to detect program’s faults. They also used a completely application black-box approach based on machine learning methods to find anomalies by comparing the application resource usage with the learned model.

The one issue for software detection is infinite loop detecting. Which loop instruction can be infinite circle depends on the condition of the loop instruction. The effect is resources of the system are used enormity, such as memory and CPU. Incorrectness of software implementation has effect for a critical software, such as finance software, shipping software, medical software, and scientific software, etc.

The loop characteristics have been defined in (de Alba and Kaeli, 2001) which can be classified in 3 categories. The first characteristic is the static control loop, called as a well-structured loop. This type of the loop will perform in the static execution time, such as `for = 1; i < 10; i++){...};`. The second characteristic is the conditional control loop, such as `while(1)do{...if(i < j), break, ...};`. The execution of the loop will be terminated when the condition in the loop is satisfied. Therefore, the execution time is predictable based on its execution profile. The name of this characteristic is defined as an ill-structured loop. The last characteristic of the loop is the most significant loop since the termination of the loop is depended on the value of a variable at run-time, such as `while(i < j)do{...};`. Thus, this characteristic is called as a variable-dependent loop.

Referring to the loop characteristics defined above, there are two groups of the loop can be classified: static loop, and dynamic loop. The static loop is the loop that has the characteristic of the static control loop, while the dynamic loops are the conditional control loop and variable-dependent loop because the termination of the loops only occurs during the run-time process. Consequently, the critical effect is relied on the dynamic loop since users will not be able to predict the computation outcomes during their execution, and probability of critical false is not zero.

Although the dynamic loop is counted as a critical problem in software testing process, there is no method to replace this necessity. Most of the business software allow users to enter data for calculation, and no one can guarantee that the entering data will not cause software failure and run-time error defined above. Unluckily, when a run-time error occurs, there is no warning system alerts for users to correct their false, or mistyping, or even terminate their process. As a result, the business organisation is in the untrustworthy situation.

Another widely used area that the implementation of the loop is very important is the communication area. In the communication area, every data transmission procedure contains at least a loop command to accept data over the network; thus, if there is an error occurs from the data transfer loop, users may not receive any data or may receive incorrect data and critical damage occurs as the consequences. Therefore, the security and trust in the communication loop must be maintained so users will be able to receive accurate data on time. Nevertheless, some loops are the infinite-loop type and hard to be predicted.

Researches related in trust of system. Robinson (1997) and Sargent (2009), designed model for verification and validation with model comparison and simulation as the real system which modeller and users may know different thresholds for confidence of system. Many researches mentioned about trust model. In the year 2009, Moghaddam et al. proposed trust model in social network with static trust values and dynamically get updated user’s feedbacks. They recommended method with exportability and
A trustable software with a dynamic loop control mechanism

dependability to predict ratings scale. They studied the values from raw trust with data of user feedbacks. If data is positive feedback, it will lead to increasing the trust among users. If data is negative feedback, it will decrease the trust. In the year 2009, Cheng et al. studied researches of designing model for develop dynamic software with self-adaptive systems in part of development methods, techniques and tools for dynamic self-adaptive behaviour and including control loop.

Researches related in loop verification and detection, Shen et al. (2001), analysed loop checking for the logic program. They developed a new complete loop checking mechanisms with key technique to expanded variants called VAF-checks (variant atoms loop checks for logic programs with functions). The key structural characteristics of infinite loops were captured. In the year 2009, Burnim et al., proposed LOOPER technique, which was an automated technique for dynamically analysing a running program to prove non-terminating program. They analysed symbolic execution in value of arguments with construct of satisfiability modulo theories (SMT) solver.

Based on the problems described above, one factor to obtain the trust of the software is the error-free loop control. Unfortunately, there is no existing testing mechanism to ensure that the dynamic loop is an error-free loop. Therefore, instead of trying to protect or detect uncertainty situation, this paper will propose a method to warn users that there is a potential of a run-time error occurs during the execution time. So, users can terminate the running program before any losses occur.

3 Proposed method and designed architecture

3.1 Execution time interval and execution mean time interval

Considering an execution time of a software as a software metric with measurement unit is ‘second’, every software will have a certain number of the execution time for each execution period and conditions. For example, software A will spend 35 sec., execution time, to execute before normally terminated. Thus, when software is repeatedly executed, there will be an execution mean time (EMT) which obtained from the average value of every execution time of the software in a certain period. Moreover, execution mean time interval (EMTI) can be calculated based on the assumption that the distribution of the execution time is normal and the confidence level (McClave and Sincich, 2009) is equal to \((1 - \alpha)\), such as 95%. As a result, the upper bound to determine the false execution is obtained from this bound.

The EMTI is calculated using \(Z\) value with confidence level is equal to \((1 - \alpha)\), or significant level is \(\alpha\), the formula for EMTI is presented as \(\bar{x} \pm z_{\alpha} \frac{s}{\sqrt{n}}\), where \(\bar{x}\) is the EMT during a certain period, \(t\), \(s\) is the standard variation of the execution time of a sample execution time values during time \(t\) and \(n\) is the total number of the execution time values during time \(t\).

3.2 Packet time out mechanism

Similarly to software, every packet in the communication channel must have time to live (TTL) after delivery in order to protect congestion situation of the communication line.
Thus, when a packet is delivered and flew in the communication channel until it reaches the TTL value, the packet will be terminated, and the sender will be informed by ICMP (Forouzan, 2007) to retransmit the data.

The situation of the floating packet in the communication channel is similar to the situation of the infinite execution loop, where the loop cannot find the terminate condition or the terminate variable is identified. In such case, the external process cannot be performed according to blank warning message from the software system. Thus, the execution is leading the organisational process to the status of risk and unpredictable actions.

Therefore, to protect the unsecure execution of the dynamic loop control and to obtain the trusted software although the software contains a dynamic loop, the TTL concept is applied using the value of the upper bound of ETI. So, whenever the execution time of the software reaches the TTL, a warning message to the user is presented and proper external action from users or system administrator can perform. Consequently, the secure and trust of the software can be achieved.

3.3 Requirements of loop controlling system

In order to create a secure and trusted software system that contains the dynamic loop, the requirements that the software controlling systems are listed as follow.

- the software must be warn whenever the loop has potential to be infinite execution
- the software must be able to terminate whenever the user or system administrator identifies the infinite loop execution
- the termination of the execution must not affect to other part of software execution.

4 Software architecture

This paper proposes two main phases consisting with eight significant processes, as shown in Figure 1. The first phase is in the software process and the second phase is in the deployment step. In the testing phase of the software process, there are three important modules: checking module (CM), EMTI module (EMTIM), and embedded module (EM). Then, in the deployment phase, or running phase, there are two subsystems: controlling and learning subsystems. Under these two phases, there are eight mechanisms to be implemented: a mechanism for automatic tracking the input software, a mechanism for instruction insertion for time measurement, a mechanism for testing the base time by a tester, a mechanism for calculating EMT and EMTI from the base time, a mechanism for transforming input software with measure time instruction to an output file, a mechanism for checking time of the loop during the run-time using the upper bound of EMTI, a mechanism for controlling the running system, and a mechanism for adjusting the calculated EMT and EMTI.
4.1 The testing phase

4.1.1 Checking module

In order to protect the problem of the infinite loop which is a critical problem for some organisation, especially for every task that relates to lives, death, and assets, the checking of the termination condition before delivering the software is the significant responsibility of testers. Considering the termination condition of a loop, the termination of each loop can be classified in two types: static, and dynamic. The static condition of the loop refers to the situation that the loop has finite number to perform its task while the dynamic condition refers to the situation that the number of the loop cannot be determined when running the software, but the termination condition is depended on the value obtained from input or processing value during the run-time of the software itself.

The CM, CM, is responsible for checking the existing of the loop instruction, such as Do…While() , While() , or For() of the input software. The CM will also check whether the termination condition of each existing loop is static or dynamic. In the situation that the termination condition is relied on the input variable, or the computation value obtained during the run-time process, this CM will send this loop to the EM which responsible for embedded a set of commands to this loop module. After passing this module, the software will be send to other two modules: EMTIM and EM. In this module, the input software has two states of the pre-compile process.
A mechanism for automatic tracking the input software: when an input software was entered into this state, each command of this software will be scanned for loop instructions.

A mechanism for instruction insertion for time measurement: this state will insert instructions for time measurement into the input software. The measurement of time will start when the loop starts its process and stop when the loop is terminated, as show in Figure 2.

Figure 2  Input software with measure time instructions

```c
//Start measure time
TimeStartCounter(&start0);
do{
    statements ;
    TimeStopCounter(&stop0);
    if (terminate == true) {break; }
}while (condition)
// end do…while
statements ;

// Start measure time
TimeStartCounter(&start0);
for (condition) {
    statements ;
    TimeStopCounter(&stop0);
    if (terminate == true) {break; }
}
```

4.1.2  EMTI module

The EMTI module, or EMITM, is the module that records the execution time of the software when runs. The input data for this module is entered by testers and this testing must related to the execution of the loops in the testing software. Thus, the data test set must not cause the infinite execution of the loop since the normal execution time is the required outcome. Therefore, the data test set must be drawn under the following conditions.

- Values for controlling loops: testers must set the testing data in the valid interval for the loops execution.
- Values for terminating loops: testers must set the testing data in the valid interval for the loops termination.
- Relation of variables for controlling loops: the control variables are indentified into two types: internal and external variables. The internal variables are all variables existing in the loop; this includes variables that accept input from any input commands within the loops. The external variables are all variables that send data to be processed in the loops. Thus, relations of variables for loop control are the defined
relations between internal and external variables, and relations among internal variables that relate to the condition of the loops.

- Algorithms of the inner loops: the complication of the algorithms of the inner loops must be focused by testers, such as loop computation, and nested loop. Details of these algorithms are described below.
  1. Inner loop calculation: testers should consider the complexity of inner loop calculations and their outcomes that relates to the loop control.
  2. Outer loop calculation: testers must be aware for all functions and procedures calls existing in the loops in which the returned data is related to the control condition of the loop.
  3. Relation between nested loops: testers should consider number of nested loop, including with variables that relate to the loop control.

Under the testing process, the execution time will record every valid process time of the software in the EMTI database, EMTI_DB. These valid run-times will be calculated to be an execution mean time (EMT); using 95% confident level, the EMTI will be computed. As a result, the upper limit of the valid execution time is obtained as an initial value before the software is delivered. The upper limit, EMTI, will be stored in the EMTI_DB, with the delivered system.

Once the software is in the deployment state, this upper limit will be used in a certain period of time to confirm that the executing program is in the error-free state by comparing this upper bound of the EMTI with the time of the running software. If the running software has run-time value higher than the upper value of EMTI, a warning message will be displayed to the user or administrator to perform a suitable task.

In the situation that the software has run-time value less than the upper limit of the EMTI, software terminates before its run-time reaches to the EMTI’s boundary, this current run-time will be stored in the EMTI_DB in a certain period of time, by the time recording module (TRM), then all new run-times will be used to recalculate the new EMTI. Thus, the system will have a new EMTI every a certain period for the real situation.

4.1.3 Embedded module

The embedded module, or EM, is responsible for embedded a piece of commands to call all modules that deal with the infinite loop protection in the running phase. This EM will not embed the full pieces of codes to perform every controlling and checking of the time boundary because the response for every error’s detection for different software is different. Thus, this EM will embed only the commands to call all modules in the running phase to protect the unlimited computing from the loop. The input file after passing the CM, this part will create new output file for the run-time process.

4.2 The running phase

After passing the testing process, the software is installed and delivered to clients. With the dynamic loops, the software will be embedded with commands from the EM as described previously. On the other hand, the software without any dynamic loop will not be embedded with extra codes.
In the running phase, the software with embedded codes from EM will call two main subsystems: controlling subsystem and learning subsystem. The details of these subsystems are elaborated as follows.

### 4.2.1 Controlling subsystem

The controlling subsystem consists of 3 important modules: time checking module (TCM), reporting module (RM), and termination module (TM). Figure 3 shows the activities of the controlling subsystem; details of each module in this subsystem are described below.

**Figure 3** Activity diagram of the control process

#### 4.2.1.1 Time checking module

TCM performs the time-boundary checking by comparing the execution time with the EMTI boundary. In this research, the TCM will be performed for every five seconds; nevertheless, the clients can set another value based on the critical of the system. In the first period of the software deployment, determined by the client, the EMTI boundary is obtained from the testing period and stored in the EMTI_DB. However, the new EMTI boundary will be obtained from the execution of the Learning subsystem. Thus, the value of the EMTI will be changed for every computing period with the $(1-\alpha)\%$ confident level.

The result from the TCM can be valid and invalid loop execution. The valid loop execution is the situation that the execution time is less than or equal to the EMTI boundary, in contrast, the invalid loop is the situation that the execution time has potential to exceed the EMTI boundary.
4.2.1.2 **Reporting module**

Whenever the TCM found that there is a potential of infinite loop occurs for the running software, the process will be transferred to the RM. This RM is responsible for creating a warning message to users or administrators. The message will be displayed to users or administrators as a dialogue, and requests users to call a TM, or continue execution. If the user selects to continue the execution, the RM will return all checking processes back to the TCM as normal process with marking of invalid value. Therefore, if the software keeps running, every five seconds the warning message will be displayed again. The warning message will be disappeared only when the software finishes execution or the users call the TM.

4.2.1.3 **Termination module**

As a result of the TCM and RM, the TM will be performed whenever the user chooses to terminate the computing process. Therefore, there will be no un-expected result from the invalid loop execution because all proper processes are defined in the TM.

4.2.2 **Learning subsystem**

4.2.2.1 **Time recording module**

Referring to the EMTI boundary from the testing process mentioned previously. The number of the normal execution time is depended on the recording period that sets by the administrator or the organisational policy. However, the size of the time period indicates the critical of the system. For example, if the system is not the critical system, the size of time period to record every execution time before calculating the EMTI can be every one hour; otherwise, it may be every five min. TRM receives the execution time from the TCM and sends it to the EMTI_DB. The execution time can be classified as valid and invalid time. The valid time means every execution time is not over than upper bound of EMTI boundary while the invalid time refers to every execution time that is over than upper bound of EMTI boundary. These invalid times will be counted for number of errors and reported to system administrator for problem solving.

4.2.2.2 **EMTIM**

This EMTIM is quite similar to the EMTIM in the testing phase except that the recording of the executing time is obtained from the TRM. The EMTIM in the deployment process will select the execution time storing in the EMTI_DB, starting from the last calculation value of the last EMTI to the last execution time of the calculation boundary. For example, if the last execution time of EMTI is 15 seconds, then, in the last 5 seconds of the time recording period, there are 30 values of the execution times before the new EMTI be calculated. So, these 30 values will be selected to compute the EMTI boundary, meanwhile, the TM still records the incoming execution time of other rounds of the software.
5 Results

According to the architecture defined in the previous section, this architecture is implemented using Microsoft Visual C++, running 15 different software, each software uses the same test cases were the numbers in the test cases are generated using the random number generator program. There are 100 different test sets to put as input of the 15 testing software. However, these testing software have the testing patterns as follows.

- the software with the repeated instruction just 1 loop with instruction for, while and do...while
- the software has repeated instruction just 1 loop and call function, by the function have not loop
- the software has repeated instruction just 1 loop and call function, by the function has some loop
- the software has repeated instruction more than 1 loop, but have not nested loop
- the software has repeated instruction more than 1 loop, but have nested loop
- the software has repeated instruction and select instruction, but do not have overlap between them
- the software has repeated instruction and select instruction, but do not have overlap between them.

Table 1 shows condition characters in the test file where each condition can encounter the infinite loop situation.

<table>
<thead>
<tr>
<th>Condition characters of loop instruction</th>
<th>Condition for exit loop</th>
</tr>
</thead>
<tbody>
<tr>
<td>for(i = 0; y! = 10; i++)</td>
<td>y = 10</td>
</tr>
<tr>
<td>while (ch! = 'c')</td>
<td>ch = 'c'</td>
</tr>
<tr>
<td>while (x! = 10)</td>
<td>x = 10</td>
</tr>
<tr>
<td>while (a! = b)</td>
<td>a = b</td>
</tr>
<tr>
<td>while (i%2 != 0)</td>
<td>i mod 2 equal 0.</td>
</tr>
<tr>
<td>while ((a = b</td>
<td>c)</td>
</tr>
<tr>
<td>while (x = function())</td>
<td>The value from function() return not equal x</td>
</tr>
</tbody>
</table>

The result of the experiments from every test cases and every testing software has shown that the proposed solution can detect the infinite loop when invalid condition occurs during the run-time process. Moreover, there is no side effect from software termination according to the TM process. Therefore, the users can be ensured that their system is secured and trustable.

Figure 4 shows the testing results for 15 software. The results have shown that implementing the proposed architecture can completely protect the infinite loop from
A trustable software with a dynamic loop control mechanism

unexpected input or computing values. Therefore, the delivered software is trustable and secure for all types of organisations.

Figure 4  The graphs comparing the number of times before using system and after using the system

![Graph comparing number of times before and after system use](image)

6 Discussions

Since the unlimited loop can cause critical damages to organisations, the warning message must be sent to users before these false occurs. This paper proposes an architecture to create a trustable system. Comparing to other protection methods, the verification and validation will be performed during the development process while the dynamic loop still have a chance to create error in the run-time of real situation according to the human’s false. Therefore, those verification and validation methods cannot completely guarantee that the critical damage will not arise after the deployment.

Consider the run-time protection methods proposed by various researchers. The input dataset is the most important factor of the protection mechanism of the protection methods since many researchers believe that the run-time errors are relied on these unpredicted values, such as method proposed by Hangal and Lam (2002). However, the input dataset that has been tested during the software testing phase may or may not cover all unpredictable cases. Therefore, some run-time errors still occur and cause a critical problem especially when the error is related to the infinite loop control. Unlike other run-time protection methods, the proposed solution using time interval to control the run-time period is much flexible and practical for any input dataset because the possible run-time interval is applied as a valid time interval of loops in the software. Thus, errors can be detected without knowing the input values during the run-time process.

Similar to the proposed method of this paper, Robert et al. (2008), Mok and Liu (1997), and Gergeleit et al. (1996) have proposed the detection techniques using time constraint for software termination. These techniques focus on the events and states of the
software within a time interval. However, the time intervals defined by these techniques are the static values that cannot be changed although the software situation is changed. So, the proposed technique in this paper is much realistic according that the time interval can be changed based on the changed situation and environment.

Moreover, this method is much simpler than the method proposed by Groce et al. (2006) where the distance metrics must be created. Additionally, the proposed method in this paper is much flexible and can be applied to every system although the system has to change the kernel. Therefore, this proposed method is a lifelong mechanism when comparing with the monitoring method proposed by Sugaya et al. (2009).

Since there are various existing researches in software verification and validation, one technique is using feedback from users to measure the trust of the system (Moghaddam et al., 2009). However, the bias of users can occur and some default may occur from other factors without informing to users. So, the feedback from users may not be accurate as expected. So, comparing with the proposed method in this research, there is no user interference then the trust of the system will be accurate as required.

As the fact that the infinite loop of software can harm system in various aspects, many researchers have focused to solve this critical defect before software are delivered to clients. One technique was presented by Shen et al. (2001) by creating a complete loop checking mechanism, called VAF-checks. However, the VAF-checking technique is still complicate when comparing this method with the proposed mechanism in this paper. Moreover, the complete loop check must be created for any new software while this technique needs not create any new code for any new software.

7 Conclusions

This paper proposes a trustable architecture for software engineering. The main concept of this infinite loop protection is to use the run-time boundary computed from the normal run-time process of the software. Thus, the architecture will consist of a learning subsystem to store the run-time and compute the EMTI, EMTI, where the boundary of the normal run-time process is identified. The controlling subsystem is responsible for controlling the run-time of the software for not exceed the EMTI boundary. This assigned task of the controlling subsystem ensures users and administrator that even though the software is delivered to users and some processing errors occur, the infinite loop situation will be able to be controlled. Finally, the critical damage can be prevented.

Acknowledgements

This paper is a revised and expanded version of a paper entitled ‘A trustable software with a dynamic loop control mechanism’ presented at the 5th International Conference on Future Information Technology (FutureTech2010), Busan, Korea, 20–24 May 2010. The authors would like to thank the Office of the Higher Education Commission, Thailand for supporting by grant fund under the program Strategic Scholarships for Frontier Research Network for the Join PhD Program Thai Doctoral degree for this research.
References


