An Automated Wrapper-based Approach to the Design of Dependable Software

Matthew Leeke
Department of Computer Science
University of Warwick
Coventry, UK, CV4 7AL
matt@dcs.warwick.ac.uk

Arshad Jhumka
Department of Computer Science
University of Warwick
Coventry, UK, CV4 7AL
arshad@dcs.warwick.ac.uk

***Abstract*—The design of dependable software systems invariably comprises two main activities: (i) the design of dependability mechanisms, and (ii) the location of dependability mechanisms. It has been shown that these activities are intrinsically difficult. In this paper we propose an automated wrapper-based methodology to circumvent the problems associated with the design and location of dependability mechanisms. To achieve this we replicate important variables so that they can be used as part of standard, efficient dependability mechanisms. These well-understood mechanisms are then deployed in all relevant locations. To validate the proposed methodology we apply it to three complex software systems, evaluating the dependability enhancement and execution overhead in each case. The results generated demonstrate that the system failure rate of a wrapped software system can be several orders of magnitude lower than that of an unwrapped equivalent.**

***Keywords*-Importance, Metric, Replication, Variable, Wrappers**

I. Introduction

As computer systems become pervasive, our reliance on computer software to provide correct and timely services is ever-increasing. To meet these demands it is important that software be dependable [1]. It has been shown that a dependable software must contain two types of artefact; (i) error detection mechanisms (EDMs) and (ii) error recovery mechanisms (ERMs) [2], where EDMs are commonly known as detectors and ERMs as correctors. A detector is a component that asserts the validity of a predicate during execution, whilst a corrector is a component that enforces a predicate. Examples of detectors include runtime checks and error detection codes. Examples of correctors include exception handlers and retry [3]. During the execution of a dependable software, an EDM at a given location evaluates whether the corresponding predicate holds at that location, i.e., it attempts to detect an erroneous state. When an erroneous state is detected, an ERM will attempt to restore a suitable state by enforcing a predicate, i.e., it attempts to recover from an erroneous state. Using EDMs and ERMs it is possible to address the error propagation problem. A failure to contain the propagation of erroneous state across a software is known to make recovery more difficult [4].

The design of efficient EDMs [5] [6] [7] and ERMs [8] is notoriously difficult. Three key factors associated with this difficulty are (i) the design of the required predicate [6] [9], (ii) the location of that predicate [10], and (iii) bugs introduced by EDMs and ERMs. The problem of EDM and ERM design is exacerbated when software engineers are lacking in experience in software development or dependability mechanisms [7]. One approach to overcoming this difficulty is to reuse standard, efficient mechanisms, such as majority voting [11], in the design dependable software. However, techniques such as replication or N-version programming (NVP) [11] [12] are expensive, as they work at the software level, i.e., the whole software is replicated in some way. It would be ideal to adapt standard, efficient mechanisms to operate at a finer granularity in order to lessen overheads.

In this paper, we propose an automated methodology for the design of dependable software. Our approach is based on variable replication. This contrasts with current state-of-the-art approaches, which operate at a software level. The replication of software can be viewed as the replication of every variable and code component in a software. However, our approach focuses on replicating *important* variables. The proposed methodology works as follows: A lookup table in which variables are ranked according to their importance is generated. Once this is obtained, we duplicate or triplicate a subset of variables based on their importance using software wrappers, i.e., creating *shadow* variables. When an important variable is written to, the value held by the all relevant shadow variables is updated. When an important variable is read, its value is compared to those of its shadow variables, with any discrepancy indicating an erroneous state. Our approach induces a execution overhead ranging from 20%–35%, and a memory overhead ranging from 0.5%–20%.

The advantages of our approach over current state-of-the-art techniques are: (i) we circumvent the need to obtain non-trivial predicates by using standard efficient predicates, viz. majority voting (ii) we circumvent the need to know the optimal location of a given predicate by comparing values on all important variable reads, (iii) the efficiency of the standard mechanisms is known a priori, obviating the need for validation of the dependability mechanisms using fault injection [13], (iv) the overhead is significantly less than would be incurred by a complete software replication, and (v) we reduce the risk of inserting new bugs through detectors and correctors [8] [14].

A. Contributions

In this paper we make the following specific contributions:

 ISBN: 978-1-61208-149-6

- We describe an automated wrapper-based methodology for the design of dependable software, outlining the steps required for its application and providing insight into the use of the metrics on which it is based.
- We experimentally evaluate the effectiveness of the described methodology in the context of three complex software systems, obtaining results which serve to validate the usefulness of the metrics proposed in [15].
- We evaluate the execution overheads of the described methodology, offering insights into the relevant trade-offs between dependability and execution overheads.

II. Related Work

Software wrapper technology has been investigated in many fields, including computer security, software engineering, database systems and software dependability. In the context of computer security, software wrappers have been used to enforce a specific security policies [16] and protect vulnerable assets [17]. It has also been shown that security wrappers deployed within an operating system kernel can be used to meet application specific security requirements [18].

Software wrappers have been widely applied in the integration of legacy systems [19], where they act as connectors which allow independent systems to interact and the reconciliation of functionality [20] [21]. Examples of this can be found in the field of database systems, where software wrappers are used to encapsulate legacy databases [22] [23].

Software wrappers have been extensively investigated in the context of operating system dependability [24] [25], where emphasis is placed on wrapping device drivers and shared libraries [26] [27]. Software wrappers have also been used to address the more general problem of improving dependability in commercial-off-the-shelf software [28], as well as several more specific software dependability issues, such as the problem of non-atomic exception handling [14].

The proposed methodology is related to [29], where wrappers were used to detect contract violations in component-based systems. In contrast, the proposed methodology combines software wrappers that implement standard predicates and variable replication to enhance dependability. The variable-centric approach, facilitated by the metrics developed in [15], also differentiates the proposed methodology.

III. Models

In this section we present the models assumed in this paper.

A. Software model

A software system S is considered to be a set of interconnected modules $M_1 \dots M_n$. A module M_k contains a set of non-composite variables V_k, which have a domain of values, and a sequence of actions $A_{k1} \dots A_{ki}$. Each action in $A_{k1} \dots A_{ki}$ may read or write to a subset of V_k. Software is assumed to be grey-box, permitting source code access, but assuming no knowledge of functionality or structure.

B. Fault model

A fault model has been shown to contain two parts: (i) a local part, and (ii) a global part [30]. The local fault model, called the *impact model*, states the type of faults likely to occur in the system, while the global model, called the *rely specification*, states the extent to which the local fault model can occur. The rely specification constrains the occurrence of the local model so that dependability can be imparted. For example, a rely specification will state that "at most f of n nodes can crash", or "faults can occur only finitely often". Infinite fault occurrences can only be tolerated by infinite redundancy, which is impossible.

In this paper we assume a *transient* data value fault model [31]. Here, the local fault model is the transient data value failure, i.e., a variable whose value is corrupted, and that this corruption may ever occur again. The global fault model is that we assume that any variable can be affected by transient faults. The transient fault model is used to model hardware faults in which bit flips occur in memory areas that causes instantaneous changes to values held in memory. A transient data value fault model is often assumed during dependability analysis because it can be used to mimic more severe fault models [31], thus making it a good base fault model.

IV. Methodology

The proposed methodology is based on the premise that the replication of important variables can yield significant reduction in failures without incurring significant execution overheads. The importance of variables is based on their implication in error propagation. The methodology is a three step process. First, a table ranking variables according to their importance for a given module is generated. Next, all read and write operations to important variables, as defined by a threshold value, are identified. Finally, such operations are wrapped using specifically designed software wrappers. An overview of the methodology is shown in Figure 1. Sections IV-A-IV-C provide a description of these steps.

A. Step 1: Establishing Variable Importance

The first step is to evaluate the relative importance of each variable within the software module to be wrapped. To achieve this we use the metric suite in [15] to measure importance. The importance metric is a function of two sub-metrics, spatial and temporal impact, and system failure rate.
Spatial Impact: Given a software whose functionality is logically distributed over a set of modules, the spatial impact of variable v in module M for a run r, denoted as $\sigma^r_{v,M}$, is the number of modules corrupted in r. Thus, the spatial impact of a variable v of module M, denoted $\sigma_{v,M}$, is:

$$\sigma_{v,M} = max\{\sigma^r_{v,M}\}, \forall r \quad (1)$$

Thus, $\sigma_{v,M}$ captures the diameter of the affected area when variable v in module M is corrupted. The higher $\sigma_{v,M}$ is, the more difficult it is to recover from the corruption. As the metric captures the diameter of the area affected by the propagation of errors, low values are desirable.

 ISBN: 978-1-61208-149-6

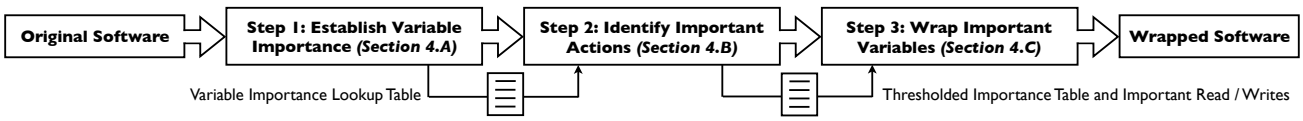

Figure 1. Methodology overview

Temporal Impact: Given a software whose functionality is logically distributed over a set of modules, the temporal impact of variable v in module M for a run r, denoted as $\tau^r_{v,M}$, is the number of time units over which at least one module remains corrupted in r. Thus, the temporal impact of a variable v of module M, denoted $\tau_{v,M}$ is:

$$\tau_{v,M} = max\{\tau^r_{v,M}\}, \forall r \tag{2}$$

Thus, $\tau_{v,M}$ captures the period that program state remains affected when variable v in module M is corrupted. The higher $\tau_{v,M}$ is, the likelier a failure is to occur. As the metric captures the persistence of errors, low values are desirable.
Importance Metric: The importance metric, as instantiated in [15], is defined for variable v in module M with variable specific system failure rate f as:

$$I_{v,M} = \frac{1}{(1-f)^2}\left(\frac{\sigma_{v,M}}{\sigma_{max}} + \frac{\tau_{v,M}}{\tau_{max}}\right)^1 \tag{3}$$

The importance of all variables in a module can be evaluated using Equations 1-3. In [15] fault injection was used to estimate variable importance, though the metric can be evaluated using alternative approaches. Note that the variable ranking generated for each module is relative to that module, which means that these values should not be compared on an inter-module basis. Once this first step of the proposed methodology has been completed, a lookup table relating any given variable to its importance value can be generated.

B. Step 2: Identifying Important Actions

The next step is to identify all read and write actions on important variables. As the replication of a whole software, or indeed every variable, incurs a large overhead, we select a subset of the most important variables for replication using thresholds. We set two thresholds to govern the number of duplicated and triplicated variables; λ_d and λ_t respectively. Thresholds may be defined with respect to importance values, though in many situations, it is reasonable to define thresholds as a proportion of the variables in a module. For example, to triplicate the top 10% and duplicate the top 15% of variables, we would set $\lambda_t = 0.10$ and $\lambda_d = 0.15$. The use of two thresholds is motivated by the desire to reduce replication overhead and provide flexibility in the application of the proposed methodology.

Once threshold values have been set, the variables to be wrapped can be identified. However, before wrapping, each possible read and write location on an important variable must be identified. This can be achieved by several means, including system call monitoring and memory management techniques. The only requirement is that all possible read and write actions on important variables must be identified. In this paper, source code analysis is used to identify important read and write actions, as detailed in Section VI. Completing this step will mean that variables to be wrapped have been identified and a mechanism has been used, or is in place, to identify read and write actions on those variables.

Algorithm 1 Write-Wrapper: Writing a variable v

```
v := f(...)
if (rank(v) ≥ λ_t) then
   create(v');
   create(v'');
   v, v', v'' := f(...);
else if (rank(v) ≥ λ_d) then
   create(v');
   v, v' := f(...);
end if
```

Algorithm 2 ReadWrapper: Reading a variable v

```
y := g(v, ...);
if (rank(v) ≥ λ_t) then
   y := g(majority(v, v', v''), ...);
else if (rank(v) ≥ λ_d) then
   y := g(random(v, v'), ...);
end if
```

C. Step 3: Wrapping Important Variables

Two types of software wrapper are employed by the methodology; *write*-wrappers and *read*-wrappers. Pseudocode for these wrappers is shown in Algorithms 1 and 2.
Write-Wrapper: This software wrapper is invoked when an important variable is written. When a variable v is assigned a value $f(\ldots)$, where f is some function, in the unwrapped module, the ranking of the variable is checked. If the rank of v is in the top λ_t, then two shadow variables, v' and v'', are created. Alternatively, if the rank of v is between λ_t and λ_d, then a shadow variable v' is created. Then, v and all of its shadow variables are updated with $f(\ldots)$.
Read-Wrapper: This software wrapper is invoked when an important variable is read. When a variable y is updated with a function $g(v, \ldots)$ in the unwrapped module, where g is a function and variable v is to be read, the rank of v is checked against λ_t. If the rank of v is greater than λ_t then function g uses the majority of the v, v', v''. If the rank of v is between λ_t and λ_d, then g uses v or v'.

 ISBN: 978-1-61208-149-6

V. Experimental Setup

In this section we detail the experimental setup used in the estimation of the importance metric for each target system.

A. Target Systems

7-Zip (7Z): The 7-Zip utility is a high-compression archiver supporting a variety of encryption formats [32]. The system is widely-used and has been developed by many different software engineers. Most project source code is available under the GNU Lesser General Public License.
Flightgear (FG): The FlightGear Flight Simulator project is an open-source project that aims to develop an extensible yet highly sophisticated flight simulator [33]. The system is modular, contains over 220,000 lines of code and simulates a situation where dependability is critical. All project source code are available under the GNU General Public License.
Mp3Gain (MG): The Mp3Gain analyser is an open-source volume normalisation software for mp3 files [34]. The system is modular, widely-used and has been predominantly developed by a single software engineer. All project source code are available under the GNU General Public License.

B. Test Cases

7Z: An archiving procedure was executed in all test cases. A set of 25 files were input to the procedure, each of which was compressed to form an archive and then decompressed in order to recover the original content. The temporal impact of faults was measured with respect to the number of files processed. For example, if a fault were injected during the processing of file 15 and persisted until the end of a test case, its temporal impact would be 10. To create a varied system load, the experiments associated with each instrumented variable were repeated for 25 distinct test cases, where each test case involved a distinct set of 25 input files.
FG: A takeoff procedure was executed in all test cases. The procedure executed for 2700 iterations of the main simulation loop, where the first 500 iterations correspond to an initialisation period and the remaining 2200 iterations correspond to pre-injection and post-injection periods. A control module was used to provide a consistent input vector at each iteration of the simulation. To create a varied and representative system load, the experiments associated with each instrumented variable were repeated for 9 distinct test cases; 3 aircraft masses and 3 wind speeds uniformly distributed across 1300-2100lbs and 0-60kph respectively.
MG: A volume-level normalisation procedure was executed in all test cases. The procedure took a set of 25 mp3 files of varying sizes as input and normalised the volume across each file. The temporal impact of injected faults was measured with respect to the number of files processed. To create a varied system load, the experiments associated with each instrumented variable were repeated for 3 distinct test cases, where each test case used a distinct set of 25 input files.

C. System Instrumentation, Fault Injection and Logging

Instrumented modules in each target system were chosen randomly from modules used in the execution of the aforementioned test cases. A summary of instrumented modules is given in Table I. The number of variables instrumented for each module accounted for no less than 90% of the total number of variables in that module. All code locations where an instrumented variable could be read were instrumented for fault injection. Those variables and locations not instrumented were associated with execution paths that would not be executed under normal circumstances, e.g., test routines.

Table I
Summary of Instrumented Software Modules

Module	Target System	Module Name
7Z1	7-Zip	LZMADecode
7Z2	7-Zip	7zInput
7Z3	7-Zip	7zFileHandle
FG1	FightGear	FGInter
FG2	FightGear	FGPropulsion
FG3	FightGear	FGLGear
MG1	Mp3Gain	NLaunch
MG2	Mp3Gain	GainAnalysis
MG3	Mp3Gain	Decode

Fault injection was used to determine the spatial and temporal impact associated with each software module [15]. The Propagation Analysis Environment was used for fault injection and logging [35]. A *golden run* was created for each test case, where a golden run is a reproducible fault-free run of the system for a given test case, capturing information about the state of the system during execution. Bit flip faults were injected at each bit-position for all instrumented variables. Each injected run entailed a single bit-flip in a variable at one of these positions, i.e. no multiple injections. For FG each single bit-flip experiment was performed at 3 injection times uniformly distributed across the 2200 simulation loop iterations that followed system initialisation, i.e. 600, 1200 and 1800 control loop iterations after initialisation. For 7Z and M3, each single bit-flip experiment was performed at 25 distinct injection times uniformly distributed across the 25 time units of each test case. The state of all modules used in the execution of all test cases was monitored during each fault injection experiment. The data logged during fault injection experiments was then compared with the corresponding golden run, with any deviations being deemed erroneous and thus contributing to variable importance.

D. Failure Specification

7Z: A test case execution was considered a failure if the set of archive files and recovered content files were different from those generated by the corresponding golden run.
FG: A failure specification was established using of golden run observation and relevant aviation information. A failed execution was considered to fall into at least one of three categories; speed failure, distance failure and angle failure. A run was considered a speed failure if the aircraft failed to reach a safe takeoff speed after first passing through critical speed and velocity of rotation. A run was considered a distance failure if the takeoff distance exceeds that specified by the aircraft manufacturer, where the distance is increased by 10 meters for every additional 200lbs over the aircraft

 ISBN: 978-1-61208-149-6

Table II
IMPORTANCE RANKING FOR 7Z1 VARIABLES

Rank	Variable	Importance	Failure Rate
1	processedPos	0.012869318	0.009893411
2	remainLen	0.010028409	0.009865020
3	distance	0.010085227	0.004867079
4	posState	0.008380682	0.004858712
5	ttt	0.006903409	0.004851485

base-weight. A run was considered an angle failure if a pitch rate of 4.5 degrees is exceeded before the aircraft is clear of the runway or the aircraft stalls during climb out.
MG: A test case execution was considered a failure if the set of normalised output files were different from those generated by the corresponding golden run.

VI. RESULTS

The importance metric can be evaluated using many different approaches, including static analysis and the evaluation of data-flow. In this paper, as in [15], importance is measured using fault injection. Fault injection is a dependability validation approach, whereby the response of a system to the insertion of faults is analysed with respect to a given oracle. Fault injection is typically used to assess the coverage and latency of error detection and correction mechanisms.

Using the approach outlined in Section V, the spatial and temporal impact of each variable was estimated. This information, and the failure rate for fault injections on each variable, was used to evaluate the importance of all variables according to Equation 3. Tables II-X show the importance ranking of all, subsequently identified, important variables for each target modules. For each variable, the *Importance* column gives the value of the importance metric, whilst *Failure Rate* is the proportion of fault injected execution that caused a system failure. Note that failure rate is assessed on a per variable basis. For example, if a variable is subject to 100 fault injected executions and 25 of these result in a system failure, then the it has a failure rate of 0.25.

The entries in Tables II-X give importance values for variable identifies as important in each module. To perform the thresholding required for this identification, we set $\lambda_d = 0.15$ and $\lambda_t = 0.10$. This meant that, for each software module, the top 15% of variables were to be wrapped, with the top 10% being triplicated and the next 5% being duplicated. For example, Table II shows 15% of the 36 variables in module 7Z, where the top three variables were triplicated and the rest were duplicated. The chosen threshold values were selected in order ensure that at least one variable in each module was wrapped, though the choice of λ_d and λ_t will typically be situation specific.

Once the importance table for each module had been thresholded, source code analysis was used to identify read and write actions on important variables. The implementation of our source code analyser was based on the premise that writes and reads to important variable are the only operation types that are deemed to be important actions. When adopting a source code analysis approach it must be recognised that any analysis tool must work under the assumption that any unidentifiable operation type could be an action relating to any important variable. This conservative stance ensures that the coverage of the process is maximised, though unnecessary overheads may be incurred.

Following the identification of read and write actions on

Table III
IMPORTANCE RANKING FOR 7Z2 VARIABLES

Rank	Variable	Importance	Failure Rate
1	numberStreams	0.757575163	0.013881579
2	highPart	0.699089580	0.015526316
3	unpack	0.453218118	0.010994318
4	sizeIndex	0.379870331	0.002755682
5	i_unpack	0.248907060	0.002698864
6	attribute	0.141369792	0.018011364
7	numInStreams	0.099065565	0.002443182
8	numSubstream	0.099059922	0.002386364

Table IV
IMPORTANCE RANKING FOR 7Z3 VARIABLES

Rank	Variable	Importance	Failure Rate
1	seekInStreamS	0.009250000	0.382360363

Table V
IMPORTANCE RANKING FOR FG1 VARIABLES

Rank	Variable	Importance	Failure Rate
1	vTrueKts	0.056881	0.003472
2	runAltitude	0.039179	0.002778
3	vGroundSpeed	0.035471	0.208333
4	alpha	0.033359	0.004629

Table VI
IMPORTANCE RANKING FOR FG2 VARIABLES

Rank	Variable	Importance	Failure Rate
1	currentThrust	1.047348000	0.010417000
2	hasInitEngines	1.016663000	0.003472000
3	numTanks	1.012560000	0.004630000
4	totalQuanFuel	1.011618000	0.004167000
5	firsttime	1.009914000	0.001736000
6	dt	0.506376000	0.005208000

Table VII
IMPORTANCE RANKING FOR FG3 VARIABLES

Rank	Variable	Importance	Failure Rate
1	compressLen	0.730128000	0.013889000
2	groundSpeed	0.433243000	0.001984000
3	steerAngle	0.053254000	0.011111000

Table VIII
IMPORTANCE RANKING FOR MG1 VARIABLES

Rank	Variable	Importance	Failure Rate
1	selfWrite	0.283413927	0.028650000
2	bitridx	0.278821206	0.012650000
3	whiChannel	0.277626178	0.008400000
4	gainA	0.160324478	0.016700000
5	curFrame	0.160096536	0.015300000
6	inf	0.160035590	0.014925000
7	cuFile	0.099405049	0.005850000

 ISBN: 978-1-61208-149-6

Table IX
IMPORTANCE RANKING FOR MG2 VARIABLES

Rank	Variable	Importance	Failure Rate
1	sampleWin	1.337694959	0.194400000
2	batchSample	0.988385859	0.031100000
3	curSamples	0.925373931	0.008350000
4	first	0.923418424	0.006250000

Table X
IMPORTANCE RANKING FOR MG3 VARIABLES

Rank	Variable	Importance	Failure Rate
1	maxAmpOnly	1.131021387	0.011825000
2	dSmp	0.683939300	0.009200000
3	winCont	0.678189611	0.000800000

important variables, the software wrappers described in Section IV-C were deployed. As the locations for read-wrapper and write-wrapper deployment were necessarily consistent with the code locations of important read and write actions respectively, information generated during source code analysis was used to drive wrapper deployment.

Figures 2 and 3 show examples of read-wrapper and write-wrapper deployments. The first line in each figure shows the original program statement before wrapping. The second line in each figure illustrates the use of wrappers. In Figure 2 the *dt* variable is being read-wrapped, whilst Figure 3 shows the *currentThrust* variable being write-wrapped. Observe that, in both cases, it is necessary to provide the wrapping functions with identifiers for the variable and location. This information is generated, maintained and known only to the wrapping software following the identification of important read and write actions, which means that it has no discernible impact on the execution of the target system.

To validate the effectiveness of the proposed methodology, the fault injection experiments described in Section V were repeated on wrapped target systems. Only one module in any target system has its important variables wrapped at any time. Table XI summarises the impact that the proposed methodology had on the dependability of all target modules. The *Unwrapped Failure Rate* column gives the original system failure rate with respect to all fault injection experiments, i.e., the proportion of failures of the unwrapped system when fault injection across all variables in the given module are considered. The *Wrapped Failure Rate* column then gives the same statistic for wrapped modules.

Observe from Table XI that the system failure rate of each module decreased in all cases, thus demonstrating the effectiveness the methodology. Further, the decrease in system failure rate of many modules is greater than combined failure rates of the wrapped variables in those modules. For example, module MG3 had an unwrapped failure rate of 0.002780830, which corresponded to 39361 failures. The same module had a wrapped failure rate of 0.000006105, corresponding to 86 failures. This improvement can not be accounted for by the 1142 failures incurred by the three wrapped variables, thus there is evidence that the error propagation problem has been addressed. This observation is

```
/* tankUPD = calc + (dt * rate); */
tankUPD = calc + (readWrapper(VARID_12, LOCID_4, dt) * rate);
```

Figure 2. Read wrapper example deployment

```
/* currentThrust = Engines[i]->GetThrust();*/
currentThrust = writeWrapper(VARID_17, LOCID_8, Engines[i]->GetThrust());
```

Figure 3. Write wrapper example deployment

Table XI
SYSTEM FAILURE RATES ASSOCIATED WITH ALL FAULT INJECTED EXECUTIONS OF INSTRUMENTED MODULES

Module	Unwrapped Failure Rate	Wrapped Failure Rate
7Z1	0.002407940	0.000017397
7Z2	0.007082023	0.000141946
7Z3	0.000856604	0.000030189
FG1	0.004582688	0.000045475
FG2	0.002481621	0.000002047
FG3	0.001471873	0.000135395
MG1	0.004983750	0.000012083
MG2	0.007888044	0.000013426
MG3	0.002780792	0.000006076

Table XII
PEAK INCREASE IN EXECUTION TIME AND MEMORY USAGE INCURRED BY WRAPPERS (SHOWN AS % INCREASES FOR EACH MODULE)

Module	Execution Time (Peak % Increase)	Memory Usage (Peak % Increase)
7Z1	26.048%	07.55%
7Z2	31.470%	18.16%
7Z3	20.359%	00.94%
FG1	30.660%	20.63%
FG2	35.829%	03.32%
FG3	23.529%	02.03%
MG1	25.983%	05.22%
MG2	28.090%	04.93%
MG3	23.174%	00.58%

particularly relevant to limiting the propagation of erroneous states that originate "upstream" of a target module.

Table XII summarises the overhead of the proposed methodology on all target modules. The *Execution Time* column gives the peak percentage increase in runtime when comparing executions of the wrapped and unwrapped target modules. The *Memory Usage* column gives the peak percentage increase in memory consumption when comparing executions of the wrapped and unwrapped target modules. All overheads were measured by monitoring target modules in isolation using the Microsoft Visual Studio Profiler.

Observe from Table XII that the execution overhead of wrapped modules varies between 20% and 35%. The worst case absolute increase in the execution time of a module was observed for module 7Z2, which increased by 31.470% to approximately 28μs. There is a coarse correlation between the increase in execution time and the number of variables in each module, though the frequency with which each variable

 ISBN: 978-1-61208-149-6

is used is likely to impact this overhead more directly. The increases in memory consumption are more varied than increases in execution time, with the maximum and minimum increases being 20.63% and 0.94% respectively. Note that the memory usage increases shown are the peak observed increases for each module. This means that the increase is unlikely to be sustained beyond the execution of a module and the relative scale of an increase may be small. For example, the 20.63% increase in memory consumption shown for FG1 module corresponds to an additional overhead of less than 4KB.

VII. Discussion

Inserting detection and correction mechanisms directly into a software system is likely to result in a low overhead, due to the fact that only a small number of variables and code segments must be added or replicated. However, as argued earlier, this approach necessitates the design of non-trivial predicates, which is known to be difficult [7]. Also, it is known that the design of correctors often introduces additional bugs into software [14]. The proposed methodology circumvents these problems by (re)using standard efficient detectors and correctors, though this comes at the cost of greater overheads. We see this problem as a tradeoff; inserting mechanisms directly is more difficult and error prone but imposes less overhead, whilst our approach can reuse simpler mechanisms at the expense of greater overheads.

The performance overheads of the proposed methodology will vary according to the extent of variable wrapping performed, i.e., according to λ_d and λ_t. Overhead comparison with similar approaches are desirable, but generally invalid due to difference in the extent, intention and focus of the wrapping mechanisms. For example, the results presented in this paper demonstrate that with $\lambda_d = 0.15$ and $\lambda_t = 0.10$, for a single module measured in isolation, our approach introduces a additional runtime overhead ranging from 20%-35% and a memory overhead ranging from 0.5%-20%. In contrast, the approach developed in [14] had a memory overhead for the masking of a fixed-duration function (5μs) of over 1200%. However, the component / object focus of this approach, as opposed to the novel variable-centric focus developed in this paper, invalidates this comparison.

Given that the software wrappers operate by updating replicated variables during writes and choosing a majority value during reads, our approach will work with variables of different types whenever the notion of equality exists or can be defined for that type. This is well-defined for integer, real and boolean types, which were the ones mostly encountered in the case studies presented, but there is no reason why the notion of equality can not be defined for composite types.

To prevent bias, the target modules in this paper were selected randomly. In reality, module selection could be based on expert knowledge, an understanding of system structure and dependability properties. For example, in systems where a given module is known to act as a "hub", it would be come a candidate for wrapping. Dependability frameworks can also be used to inform module selection [36].

The main limitation of the proposed methodology, as it has been applied in this paper, is the need for source code access. Although no attempt has been made to constrain the means by which methodology steps can be met, it may be difficult to devise an appropriate combination of means when source code is not available. For example, the identification of read and write actions on important variables was performed using source code analysis. In situations where source code is not available this is not possible, meaning that an alternative approach must be employed. However, it should be remembered that the intention of the methodology is to aid in the design of dependable software during its development, when source code is normally available.

VIII. Conclusion

In this paper we developed an automated wrapped-based approach for the design of dependable software. The novelty of the approach is in the reuse of standard efficient dependability mechanisms at the level of variables, which has been enabled by the use of software wrappers that have been built on a dynamic error propagation metric. The use of wrappers is justified by the fact that we do not require knowledge of system implementation in order to apply the methodology. The propose methodology was validated through in-depth studies of several complex software systems, each drawn from a different application domain. This application of the methodology yielded promising results, with all treated modules exhibiting significant dependability improvements.

References

[1] J.-C. Laprie, *Dependability: Basic Concepts and Terminology*. Springer-Verlag, December 1992.

[2] A. Arora and S. S. Kulkarni, "Detectors and correctors: A theory of fault-tolerance components," in *Proceedings of the 18th IEEE International Conference on Distributed Computing Systems*, May 1998, pp. 436–443.

[3] Y. M. Wang, Y. Huang, and W. K. Fuchs, "Progressive retry for software error recovery in distributed systems," in *Proceedings of the 23rd International Symposium on Fault Tolerant Computing*, June 1993, pp. 138–144.

[4] A. Arora and M. Gouda, "Distributed reset," *IEEE Transactions on Computers*, vol. 43, no. 9, pp. 1026–1038, September 1994.

[5] A. Arora and S. Kulkarni, "Designing masking fault-tolerance via nonmasking fault-tolerance," in *Proceedings of the 14th IEEE Symposium on Reliable Distributed Systems*, June 1995, pp. 435–450.

[6] A. Jhumka, F. Freiling, C. Fetzer, and N. Suri, "An approach to synthesise safe systems," *International Journal of Security and Networks*, vol. 1, no. 1, pp. 62–74, September 2006.

[7] N. G. Leveson, S. S. Cha, J. C. Knight, and T. J. Shimeall, "The use of self checks and voting in software error detection: An empirical study," *IEEE Transactions on Software Engineering*, vol. 16, no. 4, pp. 432–443, April 1990.

 ISBN: 978-1-61208-149-6

[8] H. Shah, C. Gorg, and M. J. Harrold, "Understanding exception handling: Viewpoints of novices and experts," *IEEE Transaction on Software Engineering*, vol. 36, no. 2, pp. 150–161, March 2010.

[9] A. Arora and S. S. Kulkarni, "Component based design of multitolerant systems," *IEEE Transactions on Software Engineering*, vol. 24, no. 1, pp. 63–78, January 1998.

[10] M. Hiller, A. Jhumka, and N. Suri, "An approach for analysing the propagation of data errors in software," in *Proceedings of the 31st IEEE/IFIP International Conference on Dependable Systems and Networks*, July 2001, pp. 161–172.

[11] A. Avizienis, "The n-version approach to fault-tolerant software," *IEEE Transaction on Software Engineering*, vol. 11, no. 12, pp. 1491–1501, December 1985.

[12] A. Avizienis and L. Chen, "On the implementation of n-version programming for software fault tolerance during execution," in *Proceedings of the 1st IEEE-CS International Computer Software Applications Conference*, November 1977, pp. 149–155.

[13] M. Hsueh, T. K. Tsai, and R. K. Iyer, "Fault injection techniques and tools," *IEEE Computer*, vol. 30, no. 4, pp. 75–82, April 1997.

[14] C. Fetzer, P. Felber, and K. Hogstedt, "Automatic detection and masking of nonatomic exception handling," *IEEE Transactions on Software Engineering*, vol. 30, no. 8, pp. 547–560, August 2004.

[15] M. Leeke and A. Jhumka, "Towards understanding the importance of variables in dependable software," in *Proceedings of the 8th European Dependable Computing Conference*, April 2010, pp. 85–94.

[16] P. Sewell and J. Vitek, "Secure composition of untrusted code: Wrappers and causality types," in *Proceedings of the 13th IEEE Computer Security Foundations Workshop*, July 2000, pp. 269–284.

[17] S. Cheung and K. N. Levitt, "A formal-specification based approch for protecting the domain name system," in *Proceedings of the 30th IEEE/IFIP International Conference on Dependable Systems and Networks*, June 2000, pp. 641–651.

[18] T. Mitchem, R. Lu, and R. O'Brien, "Using kernel hypervisors to secure applications," in *Proceedings of the 13th Annual Confernece on Computer Security Applications*, December 1997, pp. 175–181.

[19] E. Wohlstadter, S. Jackson, and P. Devanbu, "Generating wrappers for command line programs: The cal-aggie wrap-o-matic project," in *Proceedings of the 23rd ACM/IEEE International Conference on Software Engineering*, May 2001, pp. 243–252.

[20] A. C. Marosi, Z. Balaton, and P. Kacsuk, "Genwrapper: A generic wrapper for running legacy applications on desktop grids," in *Proceedings of the 23rd International Symposium on Parallel and Distributed Computing*, May 2009, pp. 1–6.

[21] B. Spitznagel and D. Garlan, "A compositional formalization of connector wrappers," in *Proceedings of the 25th ACM/IEEE International Conference on Software Engineering*, May 2003, pp. 374–384.

[22] A. Cleve, "Automating program conversion in database reengineering: A wrapper-based approach," in *Proceedings of the 10th European Conference on Software Maintenance and Reengineering*, March 2006, pp. 323–326.

[23] P. Thiran, J. Hainaut, and G. Houben, "Database wrappers development: Towards automatic generation," in *Proceedings of the 9th European Conference on Software Maintenance and Reengineering*, March 2005, pp. 207–216.

[24] A. K. Ghosh, M. Schmid, and F. Hill, "Wrapping windows nt software for robustness," in *Proceedings of the 29th Annual Symposium on Fault Tolerant Computing*, June 1999, pp. 344–347.

[25] A. S. Tanenbaum, J. N. Herder, and H. Bos, "Can we make operating systems reliable and secure?" *Computer*, vol. 39, no. 5, pp. 44–51, May 2006.

[26] C. Fetzer and Z. Xiao, "An automated approach to increasing the robustness of c libraries," in *Proceedings of the 32nd IEEE/IFIP International Conference on Dependable Systems and Networks*, December 2002, pp. 155–164.

[27] M. Susskraut and C. Fetzer, "Robustness and security hardening of costs software libraries," in *Proceedings of the 37th IEEE/IFIP International Conference on Dependable Systems and Networks*, June 2007, pp. 61–71.

[28] F. Salles, M. Rodriguez, J.-C. Fabre, and J. Arlat, "Metakernels and fault containment wrappers," in *Proceedings of the 29th International Symposium on Fault-Tolerant Computing*, November 1999, pp. 22–29.

[29] S. H. Edwards, M. Sitaraman, and B. W. Weide, "Contract-checking wrappers for c++ classes," *IEEE Transactions on Software Engineering*, vol. 30, no. 11, pp. 794–810, November 2004.

[30] H. Volzer, "Verifying fault tolerance of distributed algorithms formally - an example," in *Proceedings of the 1st International Conference on the Application of Concurreny to System Design*, March 1998, pp. 187–197.

[31] D. Powell, "Failure model assumptions and assumption coverage," in *Proceedings of the 22nd International Symposium on Fault-Tolerant Computing*, July 1992, pp. 386–395.

[32] 7-Zip, "http://www.7-zip.org/," February 2011.

[33] FlightGear, "http://www.flightgear.org/," February 2011.

[34] MP3Gain, "http://mp3gain.sourceforge.net/," February 2011.

[35] M. Hiller, A. Jhumka, and N. Suri, "Propane: An environment for examining the propagation of errors in software," in *Proceedings of the 11th ACM SIGSOFT International Symposium on Software Testing and Analysis*, July 2002, pp. 81–85.

[36] ——, "Epic: profiling the propagation and effect of data errors in software," *IEEE Transactions on Computers*, vol. 53, no. 3, pp. 512–530, May 2004.

 ISBN: 978-1-61208-149-6