Nested genetic algorithm for highly reliable and efficient embedded system design

Israr, Adeel; Kaleem, Mohammad; Nazir, Sajid; Mirza, Hamid Turab; Huss, Sorin Alexander

Published in:
Design Automation for Embedded Systems

DOI:
10.1007/s10617-020-09234-6

Publication date:
2020

Document Version
Author accepted manuscript

Link to publication in ResearchOnline

Citation for published version (Harvard):
Israr, A, Kaleem, M, Nazir, S, Mirza, HT & Huss, SA 2020, 'Nested genetic algorithm for highly reliable and efficient embedded system design', *Design Automation for Embedded Systems*, vol. 24, pp. 185-221.
https://doi.org/10.1007/s10617-020-09234-6

Noname manuscript No.
(will be inserted by the editor)

Nested Genetic Algorithm for Highly Reliable and Efficient Embedded System Design

Adeel Israr · Muhammad Kaleem · Sajid Nazir · Hamid Turab Mirza · Sorin Alexander Huss

Received: date / Accepted: date

Abstract Modern embedded systems must have high reliability and performance. They should be able to tolerate both hard as well as soft errors occurring in the resources constituting the system. Reliability must be part of the system design and the system must consist of non expensive off-the-shelf resources. A system-level design process of reliable system demands efficient reliability evaluation of the explored design alternatives (DA). This work presents a new approach to accelerate the calculation of reliability and execution time of the system and thereby suggests the design space exploration for a reliable system. A new data structure denoted as System Error Decision Diagram (SEDD) is proposed, which is based on both binary decision diagrams to model hard errors and zero-suppressed decision diagrams to model soft errors. The construction of the SEDD diagram and the calculation of reliability and execution time are explained in an algorithmic way. SEDD is found to be better in terms of memory requirements and construction time compared to other models available in the literature. Using SEDD and the corresponding algorithms, a nested genetic algorithm is constructed that designs system for lifetime reliability and execution time. The result of the design space exploration algorithm is a set of Pareto DAs. A so-called 'human designer' is thus able to select one of the best alternative that represents the given system requirements. The nested genetic algorithm and its benefits are illustrated using a real-life embedded application from automotive domain.

Keywords Nested Genetic Algorithm · Reliability · Execution Time · Decision Diagram

Adeel Israr
Department of Electrical and Computer Engineering
COMSATS University Islamabad, Islamabad Campus, Pakistan E-mail: adeel.israr@comsats.edu.pk

Muhammad Kaleem
Department of Electrical and Computer Engineering
COMSATS University Islamabad, Islamabad Campus, Pakistan E-mail: mkaleem@comsats.edu.pk

Sajid Nazir
Glasgow Caledonian University, UK E-mail: sajid.nazir@gcu.ac.uk

Hamid Turab Mirza
E-mail: drturab@cuilahore.edu.pk COMSATS University Islamabad, Lahore Campus, Pakistan

Sorin Alexander Huss
E-mail: huss@iss.tu-darmstadt.de Technische Universität Darmstadt, Germany

1 Introduction

Exponentially shrinking transistor size has resulted in a tremendous increase of the computational and memory resources available, making it impossible for the system designer to work at register transfer level (RTL) [1]. At present, the researchers focus on the system-level design, which addresses the synthesis of complete systems based on coarse-grain specifications. This focus is driven by the stringent time-to market requirements. To keep the pace with the market, a solution provider should be able to produce the solution to the given problem before his/her competitor. Therefore, the emphasis is given to system design methodologies that can minimize the design time and accuracy.

From the very first computer systems, the emphasis has been on the correctness of the result, in addition to the cost of the system, and the execution time of the application. With the emergence of embedded systems, the requirement for the correct results has become more stringent because these systems are being deployed to the applications in which incorrect results may translate into loss of life or property. In case of reliability-critical systems, the dedicated and possibly multiple replicated processors can be used to execute the critical tasks. Even for the common applications used everyday, a minimum level of reliability should be guaranteed. Thus, the reliability has to be engineered in the same way as the functionality of an embedded system during the whole life cycle of a product. Most of the existing approaches for the design of a reliable system initially design the functionality of the desired system then they analyze the reliability figures the design. After which they replicate and/or exchange critical parts of the embedded system in order to meet given reliability constraints. Addressing reliability in the later stages of the design process may lead to a complete redesign of the system. Therefore, the reliability of the system must be considered simultaneously along with other attributes.

Early computers were prone to hard errors due to overheating of components. Over the decades, the rate of hard errors occurring in digital devices has greatly reduced due to the use of VLSI devices. The increased level of integration has given rise to soft errors occurring in digital devices Feng et al. [2]. They observed that the soft error rate increases almost exponentially with each new technology generation. The scaling of the feature size has also increased the static power dissipation of the CMOS based VLSI device, thereby causing again an increase in the hard error rate, as reported by Srinivasan et al. [3]. Therefore, a system cannot be called reliable if it ignores hard errors.

It is hence required that embedded systems used in critical systems continue their operation even in the presence of these perturbations. In spite of the disadvantages of decreasing transistor size mentioned, the increased number of digital resources on the chip (e.g., 1000 core processor built by Scottish researchers [4]) could be used to provide the needed redundancy to continue operation of the system in the presence of these errors. However, the digital processing resource should be assumed to be fail non-silent. As observed in [5] the fail-silent resources cost twice as much, at minimum, as compared to the fail non-silent resources.

For these reasons, the reliability model for a system should embody both hard and soft errors. The new data structures and the accompanying algorithms must be constructed to help evaluate the reliability of the system considering both hard and soft errors occurring in systems. In addition, any embedded system design process should consider such a reliability model to ensure a minimum level of assurance against these errors. The execution time of the applications executing on digital processing resources must be one of the main considerations of the system design. One of the important steps in embedded system design is to evaluate the time required for the execution of the task graph on the given architecture, for instance, the input from a sensor (or a sensor network) has to be processed before the arrival of the next input.

An application implemented in an embedded system can be modeled by flow graphs consisting of a number of tasks, with the edges in the graph representing the dependence between the tasks. The number of the digital resources allocated for an embedded system can be greater than the minimum number of resource required for the functionality of the application being executed on the resources. Such a system can exist in a number of states depending upon whether any of its resource has experienced hard errors. When none of the resources allocated to the embedded system are affected by hard errors, then each task in the task flow graph should be mapped to the resource which has minimum possibility of being affected by a soft error while executing it. The reliability of the system in that state (when all the resources are without hard error) depends upon the individual probabilities of every task being executed on their respective resources without soft errors. If there is a hard error in a digital resource that was originally allocated to one of the task then the system must be reconfigured following the rule of mapping the task in the task flow graph to the resource from the set of remaining resources that has the minimum possibility of being affected by a soft error. Again, the reliability of the system in that state (when one of the previously mapped resources has a hard error) depends upon the individual probabilities of every task being executed on its respective resources without soft errors. The probability of the system being in any state depends upon the probability of the individual resources constituting the system being affected by hard errors. In this manner, the lifetime system reliability can be defined as the reliability of system averaged over various states of the system, where the minimum functionality of the application can be defined.

A system exists in two types of states. One type of state dictates when the system cannot perform the required functionality, due to the hard errors in the resources. Whereas in second type of state, it can be reconfigured to provide the desired functionality, in spite of the hard errors which may have affected some of the resources. When the system is in the second type of state then the tasks should be allocated to those resources which can execute them efficiently. The values of the execution time of the system in these types of states can be calculated. The probability of the system being in any state depends upon the probability of the individual resources constituting the system being affected by hard errors. The lifetime system execution time is therefore the execution time averaged over the life time of the system.

Fig. 1 shows the advocated abstract system-level design process describing a procedure of constructing the reliable systems while considering the lifetime system execution time. To initialize the design process, it requires the specification of the system represented as the system specification graph (see Sec. 3) as the input. Utilizing the specification graph, a data structure named as the system error decision diagram is constructed. This tool is then used to efficiently evaluate lifetime reliability and execution time of the desired system.

The proposed model denoted as *System Error Decision Diagram* (SEDD) is characterized by the following novel features;

1. The model combines the system specification presented as a task graph, the platform specification presented as a resource graph, and the mapping information into one diagram. It provides information on the system reliability and execution time. This model is constructed only once for the whole design space exploration process.
2. The SEDD is a complex diagram, which considers both hard and soft errors differently. The hard errors are modeled as nodes of a binary decision diagram (BDD) and the soft errors are treated as nodes of a zero-suppressed binary decision diagram (ZBDD).

Given the system specification a system design methodology is constructed to design system optimized for lifetime reliability and execution time.

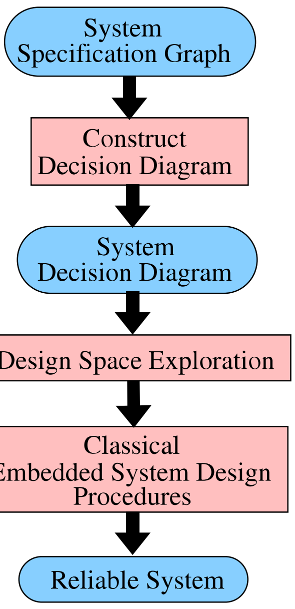

System Specification Graph
Construct Decision Diagram
System Decision Diagram
Design Space Exploration
Classical Embedded System Design Procedures
Reliable System

Fig. 1: **Abstract Design Flow for Reliable System Design**

A nested genetic algorithm has been constructed to search for the Pareto-optimal DAs in the multi-objective search space. For the genetic algorithm presented in our work, unlike the ones presented in the published research, a new system model is not needed to be constructed for the evaluation of each individual of the genetic algorithm. Lifetime reliability and execution time value is calculated for the system design represented by the individual. To calculate the lifetime values, the reliability and execution time values for each system state is calculated. In this work, we present a novel and efficient means to evaluate the reliability value for the system in a particular state. However, to calculate the best execution time of the system in the given state, a separate genetic algorithm is implemented. Similarly, the Pareto-set maintained through the genetic algorithm is used to calculate the score value of the individual. At the end of the search process, a human designer is presented with these DAs. Therefore, one can select the DA that best suits the system requirements.

The following portion of the paper is organized as follows. Sec. 2 discusses the related work. In Sec. 3, we introduce the system model and various terms used throughout the paper. Sec. 4 describes the novel tool used to model lifetime reliability and execution time of the system. In addition, the process of its construction is also discussed in this section. In Sec. 5, we present the outer most genetic algorithm for the nested genetic algorithm. The procedure to calculate the reliability and execution time of the system for a particular state of the system is given in Sec. 6. In Sec. 7 , we test the effectiveness of our approach using an example from automotive domain. Finally, Sec. 8 concludes the paper.

2 Related Work

Majority of the work done in the field of reliable system design considers either hard or soft error occurring in the system. Huang and Xu [6] have presented a simulation framework for evaluating the hard reliability of a processor-based system-on-chip for a given task to resource allocation. Whereas, in [7] Huang et al. have proposed an analytical model to estimate the hard reliability of the Multiprocessor System-on-Chip (MPSoC). Based on their models, they have presented a novel reliability-aware task allocation and a scheduling algorithm that takes the aging of the device into account. This technique is based on the simulated annealing optimization method.

The reliability should be considered as the integral part of the system design framework, along with other optimization parameters. If this is not the case, then it results in techniques like [8]. In [8], Karl et al. discuss the three main causes of the hard errors in the modern digital devices, i.e., oxide breakdown, electromigration, and thermal cycling. They discuss the dependence of the input voltage to above factors and propose dynamic voltage scaling as a part of dynamic reliability during the life time of the system. They propose to trade reliability for performance when needed (especially in case of the real time systems). This technique considers reliability only as an add-on and does not totally cover the errors that can occur in the tasks/resources constituting the system. On the other hand, considering reliability not as a part of design framework may result in techniques that sacrifice other design parameters in the favor of reliability. In [9], Zhu et al. proposed a two-step approach for the reliable MPSOC synthesis. The main concern of these authors is the hard errors occurring in the system due to thermal cycling, time dependent dielectric breakdown, and electro- and stress migration. Mean-time-to-failure is the metric used to quantify the dependability of the system being designed. Initially, a design alternative is selected using simulated annealing heuristics considering area and performance as the main objectives. In the second step, the processing is migrated from the resources with least time to failure to the redundant resources and components. This decreases the likelihood of failure of the said resource due to thermal stress, thereby improving the mean time to failure of the entire system. The problem with this solution is that the performance and area of the system resulting after the task migration might not meet the requirements, forcing the system to go through the design process again. In [10], Srinivasan et al. have discussed two techniques to improve the hard reliability of the system. Structural Duplication (SD): multiple spare structures are added to the processing resources and are gated-off when not in use, to decrease the probability of the hard errors in them. They are utilized only in case of error in the original structures. Graceful Degradation (GDR): There are many redundant structures already presented in the off-the-shelf processing resources. Exploiting the parallelism inherent in the task results in utilization of all of these structures, thereby enhancing the performance of the task. The graceful degradation allows the resource to execute on lower performance. The authors found that using SD leads to greater reliability as compared to GDR. However, this increases the overall cost of the system. In [11], Lee et al. have proposed a static task migration of the task graph from a resource with hard error to a healthy one. The migration schedule, calculated for various faults configurations, helps in maintaining the throughput of the streaming application in case of faults. In [12], Meyer et al. have presented a cost effective slack allocation approach for the improvement in the hard reliability of MPSoCs. The "slack" is defined by the authors as the under-utilization of resources (both memory and processing). They have proposed a design space exploration approach, resulting in the generation of a set of Pareto optimal alternatives representing various combination of slack utilization in the event of a hard error, with the hard reliability and cost as optimization parameters.

In [13], Bolchini et al. have presented a system level design while considering reliability as one of the critical design criteria in the embedded system HW/SW co-design flow. The co-design framework assesses the reliability properties. The system is implemented on dedicated architecture presented in [14]. This architecture features primary and auxiliary processing resources. A totally self-checking "system-reconfigurator", used as a part of the architecture, which can reconfigure the system in case of a fault in the primary processing resource. However, this architecture does not support online masking of soft errors.

Similar to the techniques that consider the hard errors while designing the system, there are techniques that address soft errors occurring in the system. However, many of these techniques consider reliability to be handled later in the design process resulting in either a partial fault coverage, or sacrificing other design parameters in the favor of reliability. Xie et al. in [15,16] consider soft errors occurring in the resources of the designed system, while designing at a high level of abstraction. These authors present a two-step approach of the system design with high performance as the main objective. Initially, a simple list-scheduling algorithm is used to find the schedule of the task graph on the resources. Then, the selected tasks are duplicated using the idle time in the schedule given in the first step, in such a way that the overall execution time from the first step is not changed. In this design methodology the performance is given a higher priority to the reliability. Not all of the tasks in the task graph are duplicated, to cover for the soft errors. However, a duplicated execution of the task does not actually increase the reliability of the said task. The proposed technique by Izosimov et al. [17] suggests a tradeoff between the performance and the reliability. The faults are detected by running a small watchdog routine after every task. Upon the detection of the error, the said task is re-executed. The authors assume "k" faults occurring in the system during the execution of the task graph on the system resources. In [18] they consider soft errors occurring in the resources while executing the tasks with a mixture of hard and soft real time requirements. The authors have proposed a quasi-static scheduling technique, where initially a static schedule is executed in case of no faults occurring in the system. Each task is followed by a fault detection routine. Upon detection of the fault, the task is re-executed. For the re-execution, the tasks with hard real time requirements are preferred. The resource allocation is done so that at least "k" faults are tolerated while making sure that the hard real time tasks meet their deadlines. In case more faults occur, then the tasks with soft real time requirements are dropped in favor of those tasks with hard real time requirements.

In [19], Izosimov et al. have proposed a high level system design technique aiming at the mitigation of the soft errors in the system using radiation hardened resources and task re-execution. A set of Pareto design alternatives, that meet the deadline requirement of the hard real time system, is generated using a Tabu search meta-heuristic. Each alternative represents various degrees of the used hardened resources and task re-execution, with reliability and overall system cost as the optimization parameters. In [20], these authors have proposed a high-level system design approach considering the soft error occurring in the system. Tabu search is conducted to find the right mix of task re-execution and task replication (on multiple resources) so that the reliability is maximized while minimizing the system cost. In [21], Conner et al. have proposed a hybrid genetic/heuristic based HW/SW co-synthesis framework for partial coverage of soft errors. The resulting schedule of the task graph on the given set of resources is shown to cover 22% of the soft errors without incurring any extra cost, in terms of resources and performance of the task graph. In [22], Calvert et al. have proposed an integrated framework for finding a set of components best suited for the system requirements. The selection process is assisted by a simulated annealing algorithm and a greedy approach designed to find a system with the best reliability, cost, and delivery time.

The authors like Srinivasan et al. [23] have proposed that hard and soft reliability of the system should be considered simultaneously. Zhang and Orshansky [24] have suggested that in spite of the "burn-in" process, the hard errors can still occur in the system and therefore should be considered during the design process of the system. In [25], Grüttner et al. have presented the idea of the dependable state machine for the design of embedded systems at a high-level of abstraction. The fault detection is achieved by monitoring the output of the tasks. Upon detection of a fault the process of rescheduling of affected task is initiated. In [26], Jhumka et al. have developed the concept of using dependability as an optimization criteria in the system level design process of embedded systems. The multiple instances of the tasks in the task graph are executed in parallel to make the task execution more reliable, thereby causing an increase in the dependability of the complete system. A set of Pareto optimal design alternative is generated using genetic algorithms, with dependability and cost as optimization objectives. However, they do not describe any method of error detection, thus implying the use of fail-silent resources. In [27], Glaß et al. have reported a scheme for reliable system design at the system-level considering both hard and soft errors. A process in the process graph is bound to multiple resources, which execute each a copy of this process simultaneously. In the case of an error (hard or soft), the affected resource produces an error signal indicating the corresponding process to ignore the result produced by this resource. One disadvantage of this approach is the need of fail-silent resources, which are largely costlier than off-the-shelf resources. Additionally, the proposed approach evaluates reliability using a binary decision diagram, which demands higher node numbers and considerably longer calculation times for reliability evaluation. A set of Pareto optimal design alternative is generated using genetic algorithms, with the mean-time-to-failure, the area and the latency as the optimization objectives.

The classical reliability calculation methods like fault-trees, reliability block diagrams, and binary decision diagrams, require a large amount of memory. If the specification for the system to be designed is large, then the structures required for the calculation of the reliability cannot be realized. Following approaches support the construction of reliable embedded systems in spite of this handicap. In [28], Glaß et al. consider only the hard errors occurring in the embedded system, especially in its electronic control units. In their paper, the hard errors are assumed to occur due to the constant vibrations experienced by the system because of its relative position to the engine or other moving parts of a vehicle. They utilize the binary decision diagrams in calculating the reliability of the system while considering the hard errors occurring in the resources of the system. The BDD construction process described by these authors requires the use of temporary variables. These variables are eliminated resulting in a BDD consisting of the nodes with resources of the system as the variables. These temporary variables result in the memory overrun especially in the case of large specification. To overcome this problem, the authors have proposed to partition the specification, thereby constructing small BDDs for each partition. These small BDDs are then combined to make the final BDD for the entire specification. This BDD is then used in calculating the reliability value for the design alterative generated by an ILP solver.
In [29], however same authors have pointed out two disadvantages of their previous approach detailed in [27]. First, for many specification graphs the size of the associated BDDs is often up to several orders of magnitude larger than the available memory. Secondly, many additional temporary nodes are required during the construction of the final data structure, thus further exacerbating the memory problem. They have presented in [29] two methods to address these disadvantages. The first one overcomes the second disadvantage only, whereas the second method results in just an approximate calculation of the reliability values.

One of the important steps in the embedded system design is to evaluate the time required for the execution of the task graph on the given set of processing resources. The input from a sensor (or a sensor network) has to be processed before the arrival of the next input. The problem of scheduling of a task graph has been studied for quite some time [30–32]. The number of possible schedules, in the worst-case situation, grows as the factorial of the number of the tasks and finding an optimal schedule among them is a non-polynomial hard problem [33–35]. These reported research work are mostly concerned with the problem of scheduling and evaluation of the execution time of the task graph on a system based on homogeneous processors. The use of decision diagrams has also intrigued the researchers to help solve this problem. In [34] the authors have tried to solve the problem of scheduling a task graph on a set of homogeneous processors by constructing a binary decision diagram and by using the breath firth search algorithm for finding an optimal schedule. However, their progress was halted by the problem of variables ordering of the BDD. The complexity of scheduling a task graph on a set of heterogeneous processors is more than that of homogeneous processors. In [36] and [37], Topcuoglu et al. proposed two algorithms to schedule tasks in a heterogeneous environment. Both of them use the critical path to prioritize tasks first and then schedule them. In [38] and [39], scheduling algorithms based on dynamic critical paths is studied. In [40], a hybrid scheduling method is proposed, which consists of mapping before execution and a remapping based on run-time values during execution.

A few researchers have tried to study both the execution time of the overall system and its reliability. In [41], Girault et al. have considered the soft errors occurring in the system. The system is considered as a task graph implemented on a set of heterogeneous resources. The reliability is achieved by active replication of the tasks and is evaluated using the reliability block diagrams. The execution time of the system is then calculated using a list-scheduling algorithm. Given that, the authors set up a bi-objective algorithm to generate the Pareto optimal designs. Xie et al. [16], exploit the presence of time slots in the schedule of the task graph produced by the scheduling process. These time slots are then utilized to execute replicas of the tasks executed forehand by the system architecture, thereby improving the reliability of the system as compared to the *base system*. The work done by Jhumka et al. [26] is one of the major works that explores jointly the execution time and the dependability as the optimization parameters for a design space exploration algorithm. The multiple instances of the tasks in the task graph are scheduled to be executed in parallel thereby increasing the dependability of the overall system. However, they do not describe any method for the required error detection implying the need of so-called *fail silent* resources. In the paper by Glaß et al. [27], the authors use BDD to calculate the reliability value for the task graph implemented on a set of connected resources. The output of the presented technique is a set of Pareto optimal design alternative with the reliability, the cost, and the latency as the optimization parameters. Even though the authors do not actually state the method of calculating the latency in the paper, the latency is displayed as the label of one of the axis in the graph depicting the Pareto optimal alternatives.

Our research work considers both hard and soft error occurring in the system. It also considers fail non-silent resources as this approach is significantly cheaper as compared to the other alternatives. Here, we are using system error decision diagram to model the overall reliability. Our approach is based on a combination of binary decision diagram (BDD) and Zero-suppressed binary decision diagram (ZBDD). It has been found to require less number of nodes [5] (and therefore less processing time) than using pure BDD solution.

A system design space exploration considering the reliability and the execution time as the integral optimization parameters is described in Sections 5. The reliability of the system is expressed as a function of both hard and soft errors occurring in the system designed using fail non-silent resources only.

3 System Model

This section contains the definitions of different terms needed to understand the system error decision diagram (SEDD), which is introduced in Sec. 4.

Table 1: Task Instance Sets and System Instance Sets for example SSG

RSV	TIS(X)	TIS(Y)	TIS(Z)	valid and invalid system instances	SIS
$r_1 r_2 r_3$	$\{X_1, X_2\}$	$\{Y_1, Y_2\}$	$\{Z_3\}$	$\{(X_1, Y_1, Z_3), (X_1, Y_2, Z_3), (X_2, Y_1, Z_3), (X_2, Y_2, Z_3)\}$	$\{(X_1, Y_1, Z_3), (X_1, Y_2, Z_3), (X_2, Y_2, Z_3)\}$
$r_1 r_2 \overline{r}_3$	$\{X_1, X_2\}$	$\{Y_1, Y_2\}$	$\varnothing$	$\varnothing$	$\varnothing$
$r_1 \overline{r}_2 r_3$	$\{X_1\}$	$\{Y_1\}$	$\{Z_3\}$	$\{(X_1, Y_1, Z_3)\}$	$\{(X_1, Y_1, Z_3)\}$
$r_1 \overline{r}_2 \overline{r}_3$	$\{X_1\}$	$\{Y_1\}$	$\varnothing$	$\varnothing$	$\varnothing$
$\overline{r}_1 r_2 r_3$	$\{X_2\}$	$\{Y_2\}$	$\{Z_3\}$	$\{(X_2, Y_2, Z_3)\}$	$\{(X_2, Y_2, Z_3)\}$
$\overline{r}_1 r_2 \overline{r}_3$	$\{X_2\}$	$\{Y_2\}$	$\varnothing$	$\varnothing$	$\varnothing$
$\overline{r}_1 \overline{r}_2 r_3$	$\varnothing$	$\varnothing$	$\{Z_3\}$	$\varnothing$	$\varnothing$
$\overline{r}_1 \overline{r}_2 \overline{r}_3$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$

3.1 **System Specification Graph**

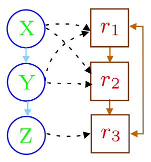


Fig. 2: System Specification Graph Example

Fig. 2 depicts a system specification graph (SSG). It is a combined graph, which consists of three sub-structures. A **task graph** is an acyclic directed graph $G_t = (V_t, E_t)$, which describes the system functionality. The vertices $t \in V_t$ of the graph G_t represent the tasks and the edges $e_t \in E_t$ represent data dependencies. The SSG also consists of a **resource graph** which is a directed graph $G_r = (V_r, E_r)$, that describes the processing resource architecture. The vertices $r \in V_r$ represent the processing resources and the edges $e_r \in E_r$ represent the interconnections between the resources. Every resource r in the resource graph are attributed with specific reliability $R_{sp}(r)$, which represents the effects of hard errors on the resource (r). In this study, the faults in the communication links are not explicitly considered. If there is fault in the communication link then the resources connected with that link are permanently inaccessible and therefore are considered to be suffering from hard errors. Further, the reliability of a resource node is a function of the reliability of its processing unit and its communication link. The task graph and the resource graph are connected by a set of **mapping edges** $m = (t, r) \in E_m \subseteq V_t \times V_r$, which represents all mapping possibilities of tasks onto resources. Every mapping edge is attributed with: Specific time $T_{sp}(m)$, which represents the worst-case execution time of the task t on the resource r, and Specific reliability $R_{sp}(m)$, which represents the soft error occurring during the execution of the task t on the resource r. Table 2 gives the soft reliability and the specific execution time values for the mapping edges of the system specification graph of Fig. 2.

3.2 **Task Instance**

A task instance (TI) is the logical representation of a task on the resource graph. Following are the two types of TIs studied in this work:

Table 2: **Reliability and time attributes of Mapping Edges in the SSG of Fig. 2**

Mapping Edges (m)	$R_{sp}(m)$	$T_{sp}(m)$ [s]
(X, r_1)	0.91	0.8
(X, r_2)	0.99	0.79
(Y, r_1)	0.99	0.81
(Y, r_2)	0.91	0.89
(Z, r_3)	0.95	0.92

3.2.1 Singular Modular Redundancy (SMR)

A SMR TI is a tuple (t, r), where t represents a task and r represents a resource, which can execute t. The resource r is both the input as well as the output resource for this TI. This means that the resource r transmits the data, produced by executing the task t, to the input resources of the TIs for the tasks that are data dependent on the task t. Also, the resource r receives the output of the TIs of the tasks that generates data for the task t. A SMR TI is represented for brevity by the name of the task indexed by the number of the resource. X_1, X_2, Y_1, Y_2, Z_3 are the SMR TIs available in the SSG of Fig. 2. Pictorially, these TIs are represented as the nodes of lower half of the decision diagram (see Sec. 4 depicted in Fig. 7(d)). The name of the task is depicted in green color, whereas the resource, which is both input as well as the output resource for this TI, is depicted in red color. The specific reliability for a SMR TI (t, r) is equal to the specific reliability assigned to the corresponding mapping edge (t, r). This attribute represents the soft error occurring during the execution of that task t on the resource r. The specific time attribute for the SMR TI (t, r) is equal to the specific time of the constituent mapping edge.

3.2.2 Triple Modular Redundancy (TMR)

A TMR TI is a 4-tuple (t, r_1, r_2, r_3), where t represents a task, and r_1, r_2, and r_3 represent the resources, which can execute t. The task t is executed on all the three resources at the same time, but the voting takes place at the resource r_1. A 4-tuple (t, r_1, r_2, r_3) is a TMR TI for the task t, if it fulfills the following requirements:

1. $m_1 = (t, r_1), m_2 = (t, r_2), m_3 = (t, r_3) \in E_m$
2. $\{(r_2, r_1), (r_3, r_1)\} \subseteq E_r \vee \{(r_3, r_2), (r_2, r_1)\} \subseteq E_r$

All the resources r_1, r_2, and r_3 are the input resources, whereas only r_1, i.e., the voting resource, is the output for this TI. A TMR TI is represented by, for brevity, the name of the task indexed by the number of the resources. For simplicity and place reasons, the above definitions and examples assume a single modular redundancy (SMR). A TMR TI is not possible for the SSG in Fig. 2, as none of the requirements for TMR are fulfilled. Our solution, however, enables the construction of TMR TIs. For example the SSG as depicted in Fig. 3(a). Note that only TIs created using this TMR are able to correct soft errors. Fig. 3(b) depicts the pictorial representation of one of the TI defined for the system specification graph of Fig. 3(a). The task X is depicted in green color, whereas the resource r_2, which is the output resource for this TI, is depicted in red color. The resources r_1 and r_3, which are only input resources, are depicted in blue color.

A TMR TI produces correct result if at most one of its resources does not suffer from soft error. The specific reliability for a TMR TI (t, r_1, r_2, r_3) is calculated using the specific reliability assigned to

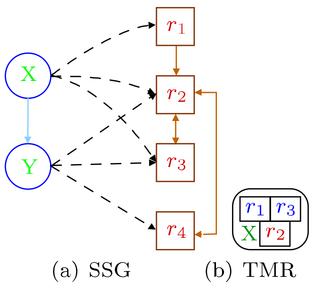


Fig. 3: **TIs for X of SSG in Fig. 2**

the corresponding mapping edges (t, r_1), (t, r_2), and (t, r_3) and is given in Eq. 1. The first line in the formula represent that there is no soft error in any of the resources. Whereas the second, third and fourth lines in the formula model the effect of soft error in first, second or the third resources of the TMR TI respectively.

$$R_{sp}(ti) = \begin{array}{ll} R_{sp}[t,r_1] \times R_{sp}[t,r_2] \times R_{sp}[t,r_3] & + \\ (1 - R_{sp}[t,r_1]) \times R_{sp}[t,r_2] \times R_{sp}[t,r_3] & + \\ R_{sp}[t,r_1] \times (1 - R_{sp}[t,r_2]) \times R_{sp}[t,r_3] & + \\ R_{sp}[t,r_1] \times R_{sp}[t,r_2] \times (1 - R_{sp}[t,r_3]) & \end{array} \tag{1}$$

The specific time attribute for the TMR TI (t, r_1, r_2, r_3) is expressed as the maximum of the specific time attribute of the three constituent mapping edges. For the general TMR TI $t_{[1,2,3]}$ the specific time attribute is expressed as:

$$T_{sp}(t_{[1,2,3]}) = \max[T_{sp}(t, r_1), T_{sp}(t, r_2), T_{sp}(t, r_3)]$$

3.3 Resource State Vector

Given a resource graph with m resources, a resource state vector (RSV) is a vector of literals $r_1r_2...r_m$, where r_i represents a resource of that resource graph. A literal with a bar indicates a hard error in that resource. First column of Table 1 depicts the RSVs for the SSG in Fig. 2. The vector $r_1r_2r_3$, for instance, indicates that none of the resource has a hard error. In contrast, the vector $\overline{r}_1\overline{r}_2\overline{r}_3$ represents the case, where all resources have hard errors.

A resource state tree is a graphical representation of all RSVs of a SSG. This tree has the following properties:

1. The RST is a binary tree.
2. The root and each internal node represents a resource.
3. The root and each internal node has two outgoing edges. The left one indicates the corresponding resource has a hard error and a right one indicates that the corresponding resource has no hard error
4. A path from the root node to a terminal node (i.e., a leaf) represents a RSV. This RSV is also written as the label for this terminal node. Fig. 4 depicts the RST for the resources of the SSG given in Fig. 2.

For each task and each RSV a TI set can be defined. This set includes all TIs of that task, which can be executed in the case of the resource state represented by that RSV. Columns 2, 3, and 4 of Table 1 show the task instance sets for all tasks and all RSVs relating to the SSG of Fig. 2. The set $\{X_1\}$ can

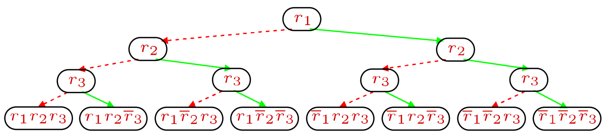

r_1
r_2
r_2
r_3
r_3
r_3
r_3
$r_1r_2r_3$
$r_1r_2\overline{r}_3$
$r_1\overline{r}_2r_3$
$r_1\overline{r}_2\overline{r}_3$
$\overline{r}_1r_2r_3$
$\overline{r}_1r_2\overline{r}_3$
$\overline{r}_1\overline{r}_2r_3$
$\overline{r}_1\overline{r}_2\overline{r}_3$

Fig. 4: Resource State Tree for example SSG

be constructed for task X and RSV $r_1\overline{r}_2r_3$ This TIS indicates that for this RSV, the task X can only be executed on r_1, as r_2 has a hard error. Note that we do not yet consider soft errors.

3.4 System Instance

A system instance(SI) is a logical representation of a task graph on a resource graph. For each resource state vector RSV several valid system instances may be found. A valid SI is defined as a tuple of n TIs, where:

1. n is the number of tasks in the SSG,
2. each task of the SSG is represented by one and only one TI in the tuple, and
3. the selected TIs must ensure data transfer between data-dependent tasks.

An n-tuple, which does not fulfill the last condition is called an *invalid SI*. The concept of invalid system instances is needed as our algorithms produce such tuples inherently. These tuples, however, are removed in the progress of the SEDD construction, see Section 4.2. Column 5 of Table 1 shows all valid and invalid system instances for considered SSG example. The tuple (X_1, Y_2, Z_3), for instance, represents a valid SI. It indicates that the tasks X_1, Y_2, and Z_3 are executed on the resources r_1, r_2, and r_3, respectively. In contrast, the tuple (X_2, Y_1, Z_3) is an invalid SI as there is no communication link from r_2 to r_1, executing task X and Y respectively. Assuming that all the tasks in the task graph are equally important for the system functionality, then the soft reliability for the SI is defined as follows:

$$R(SI) = \prod_{ti_i \in SI} R_{sp}(ti_i), \tag{2}$$

The second column of Table 3 gives the soft reliability values for all the valid system instances of the SSG of Fig. 2. These values are calculated using the values in the Table 2.

3.4.1 Linear System Instance

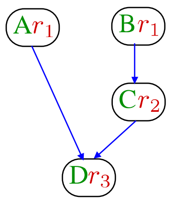
Evaluating the execution time of the SI is straight forward provided the task graph and henceforth the SI are linear (i.e., they do not contain any fork or join structures). The time attribute for such a SI can be calculated by adding the specific time attribute values of the constituent TIs. The third column of Table 3 gives the execution time values for all the valid system instances of the SSG of Fig. 2.

3.4.2 Non-Linear System Instances

The calculation of the time attribute for the graphs containing non-linear structures such as *fork* or *join*, is not as simple as the summation of the specific time attribute values of the constituent TIs. The *order*

Table 3: **Attributes of the system instances for the SSG of Fig. 2**

System Instances	$R(SI)$	$T(SI)$ [s]
(X_1, Y_1, Z_3)	0.855855	2.53
(X_1, Y_2, Z_3)	0.786695	2.61
(X_2, Y_2, Z_3)	0.855855	2.6



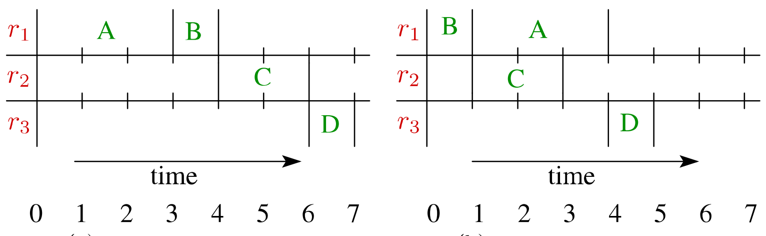
(a) System Instance

TI (ti)	$T_{sp}(ti)$ (units)
A_1	3
B_1	1
C_2	2
D_3	1

(b) Specific Time Attribute

Fig. 5: **System Instance and Specific Time Values of its TIs**

in which the TIs are processed by a scheduling algorithm (given in this section) becomes significant when the task graph is non-linear. A task priority vector (TPV) is a list of the tasks in a task graph that depicts a desired *sequence* in which the TIs representing the respective tasks are to be executed on the resource graph.



(a) $t_A \rightarrow t_B \rightarrow t_C \rightarrow t_D$ (b) $t_B \rightarrow t_A \rightarrow t_C \rightarrow t_D$

Fig. 6: **Execution Schedules for the SI of Fig 5(a)**

Fig. 6 shows the execution schedules for the two task priority vectors for the SI of Fig 5(a). Note that this SI has nothing to do with the SSG in Fig. 2, as all the system instances of that SSG are linear. The time attribute value for the SI with TPV: $t_A \rightarrow t_B \rightarrow t_C \rightarrow t_D$ is 7 units as depicted in Fig. 6(a). If the TPV is changed to $t_B \rightarrow t_A \rightarrow t_C \rightarrow t_D$, then the resulting time attribute value is 5 units as depicted in Fig. 6(b).

The procedure in Algo. 1 evaluates the reliability and execution time attributes of a SI given a task priority vector. The process is initialized by generating *ReadySet* (Step 2 in Algo. 1), initializing it as a set of TIs with no preceding TIs in the graphical representation of the SI. If the SI of Fig 5(a) is considered as an input to this procedure, then the *ReadySet* will be initialized as $\{A_1, B_1\}$. Step 7 selects the TI from the *ReadySet* that corresponds to the task with highest priority in the input TPV. For the TPV: $t_C \rightarrow t_A \rightarrow t_B \rightarrow t_D$ and the *ReadySet* $(\{A_1, B_1\})$ the priority of the tasks for the TIs A_1 and B_1

Algorithm 1: EvlSIRelExTm: Reliability and Execution Time of a SI

```
   Input: $SI$: a SI; $TPV$: the task priority vector
   Output: $RelSI$: Reliability value of SI ($SI$); $ExeTimeSI$: time attribute for the SI ($SI$)
 1 begin
 2   Initialize $ReadySet$ using the SI graph
 3   Initialize $TIFinTimeVec$ to zeros
 4   Initialize $ResAvailTimeVec$ to zeros
 5   $RelSI \Leftarrow 1.0$
 6   while $ReadSet \neq \varnothing$ do
 7     $ReadyTI \Leftarrow$ ti from $ReadySet$ with highest priority (using $TPV$)
 8     $PredSet \Leftarrow ReadyTI_{Predecessor\ ti\ Set}$
 9     foreach $ti$ in $PredSet$ do $StartTime \Leftarrow TIFinTimeVec[ti_{taskID}]$
10
11     foreach $r$ resource of $ReadyTI$ do
12       $StartTime \Leftarrow$MAX$[StartTime, ResAvailVec[r_{ID}]]$
13     end
14     $FinTime \Leftarrow StartTime + T_{sp}(ReadyTI)$
15     $TIFinTimeVec[ReadyTI_{taskID}] \Leftarrow FinTime$
16     foreach $r$ resource of $ReadyTI$ do $ResAvailVec[r_{ID}] \Leftarrow FinTime$
17
18     $RelSI \Leftarrow RelSI \times R_{sp}(ReadyTI)$
19   end
20   foreach $ti$ in $ReadyTI_{Dependant\ Task\ Instance\ Set}$ do
21     if All Preceding TIs of ti have finished execution then
22       Insert $ti$ into the $ReadySet$
23     end
24   end
25   Remove $ReadyTI$ from the $ReadySet$
26   $ExeTimeSI \Leftarrow 0.0$
27   for $i$ 1to $SI_{size}$ do $ExeTimeSI \Leftarrow$MAX$[ExeTimeSI, TIFinTime[i]]$
28
29 end
```

are 2 and 3, respectively. Therefore, TI A_1 is selected in this step. This step results in the selection of the correct TI even though the TPV does not follow the task dependencies of the task graph, which is not the case of the TPVs depicted in Fig. 6. Such a task priority vector may be generated as the result of the mutation and crossover operators of the genetic algorithm (Section 13 and 19). Steps 8 to 13 evaluate the *start time* of the selected TI. It is the maximum of the *finish time* of its preceding TIs (Steps 8 to 10) and the maximum of the *available time* of the resource(s) (Steps 11 to 13) it is executed on. The evaluation of the finish time for the selected TI is done in Step 14. For evaluating the reliability of the SI, the specific reliability attribute of the TI is multiplied and equated to a variable which is initialized to 1.0 (Step 18) before the start of the loop. The time attribute for the input SI is then evaluated by finding the maximum of the finish times of all TIs (Step 27).

3.4.3 System Instance Set

The set of all valid system instances belonging to a RSV is denoted as a system instance set (SIS). Column 6 of Table 1 shows the system instance sets for all RSVs of the SSG in Fig. 2. A SI is selected amongst all SIS to represent the system when it is in a state defined by the RSV. The chosen SI is the one with best reliability and execution time value from system instances in the SIS. The procedure to deterring the best SI is explained in Sec. 6.

4 System Error Decision Diagram

In this section system error decision diagram is introduced as an efficient means to model the lifetime reliability and execution time of the desired system.

4.1 **Decision Diagram**

A system error decision diagram (SEDD) is a decision diagram which describes the conditions for a correct operation of the entire system considering both hard and soft errors. The SEDD has three types of nodes:

1. *Resource nodes*, which settle on the upper portion of the SEDD. The outgoing right edge of the node (green and continuous) indicates that the corresponding resource is free from hard errors. In contrast, the left edge (red and dotted) represents a hard error. Each path in this portion represents one or more RSVs.
2. *Task instance nodes*, which settle on the lower portion of the SEDD, model the soft errors. Together, these nodes represent the system instance sets for the RSVs depicted in the upper portion of the decision diagram. A task instance node and the child following its outgoing right edge (green and continuous) belongs to the same system instance. Whereas, the child following the outgoing left edge (red and dotted) does not belong to the same SIS.
3. Two *terminal nodes*: a right one labeled **1**, which represents a functional system and a left one labeled **0**, which represents a failed system due to hard and/or soft errors.

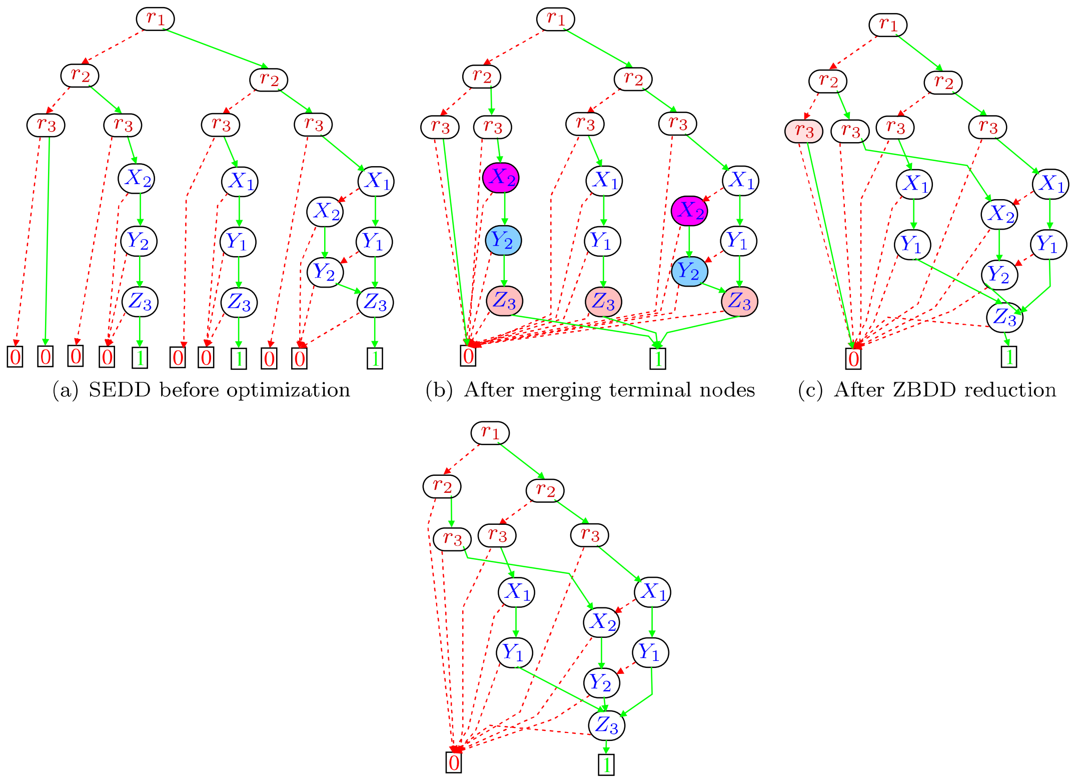
The resource nodes are represented at the upper portion of SEDD whereas the TI nodes are represented as the nodes of its lower half. This is because the SEDD calculates the lifetime reliability of the system which is an average reliability value of the system calculated over the lifetime of the system. When being in a particular state (defined as RSV Section 3.3) the system is represented by a particular system instance whose reliability value is calculated using reliability values of its constituent task instances. The system may be represented by a different system instances if it is in a different state. When calculating the average value, the frequencies are always applied at the end. While traversing the SEDD graph, the effect of the top resource nodes is considered later as they are used for frequency averaging.

Fig. 7(d) presents the SEDD for the SSG of Fig. 2. Fig. 7(a) to Fig. 7(c) depict the SEDD in preliminary stages as it will be illustrated in the next section. Note that the reliability attributes are not shown in these figures for clarity. A decision diagram consisting of only task instance nodes and the root node can represent a SIS and is referred to as a task instance decision diagram (TIDD).

4.2 **SEDD Construction**

The procedure of constructing a system error decision diagram from a system specification graph is given in Algorithm 2. The different steps of the algorithm are described in abstract form without implementation aspects for clearness. In the following some of the steps of Algorithm 2 are illustrated. For this purpose the SSG of Fig. 2 and the relating examples given in the last section are utilized.

Step 1, Step 2, and Step 3.1 are straightforward and can be understood referring to Sec. 3.2.1 and Sec. 3.3. In Step 3.2 we apply the Cartesian product from the set theory to construct all possible system instances which are ordered tuples, whose first element stems from of the first set (in column 2), second element from the second set (in column 3), etc. Column 5 of Table 1 shows the result of this operation. According to the resource graph, some system instances may be invalid. Step 3.3, therefore, removes these invalid system instances. The resulting sets are the system instance sets, see column 6 of Table 1. Step 3.4 constructs a decision diagram, which describes the system behavior in the case of soft errors for each resource state vector RSV. Although binary decision diagrams (BDDs) can be used for this purpose, we applied the zero-suppressed binary decision diagrams (ZBDD). Fig. 7(a) shows the diagram resulting



(a) SEDD before optimization (b) After merging terminal nodes (c) After ZBDD reduction

(d) After BDD reduction

Fig. 7: Constructing the SEDD for the example SSG

Algorithm 2: Construct SEDD

```
    Input: System Specification Graph SSG
    Output: System Error Decision Diagram SEDD
  1 Create all task instances TIs of the SSG;
  2 Create the resource state tree RST for the SSG;
  3 foreach RSV do
3.1     foreach task in the task graph do create the task instance set TIS;
3.2     Apply product operator to TI sets of all task to construct all possible system instances;
3.3     Construct the SIS by removing invalid system instances resulting from step 3.2;
3.4     Construct a ZBDD for the SIS resulting from step 3.3 [42];
3.5     Replace the terminal node of the RST by the constructed ZBDD;
  4 Optimize the resulting diagram:
4.1   Merge all terminal nodes into two terminal nodes;
4.2   Remove all redundant task instance nodes using ZBDD reduction rules [42];
4.3   Remove all redundant resource nodes using BDD reduction rules [43];
  5 return SEDD;
```

from applying Step 3.4 to all RSVs and appending the resulting ZBDDs to the RST according to Step 3.5. ZBDDs are appropriate for applications with sparse nodes, as in the case of task instance nodes. In the case RSV $= r_1r_2r_3$, the SIS contains 3 system instances as shown in the column 6 of Table 1. Representing this SIS as a BDD would require a total of 9 nodes whereas it requires 5 nodes to represent it as ZBDD shown in Fig. 7(a). Please note that the reduced number of nodes of SEDD is the first advantage of our approach towards fast reliability evaluation. Steps 4.1, 4.2, and 4.3 are then executed to get the optimized SEDD shown in Fig. 7(d). In Step 4.2 all nodes with the same task instance and with identical successor nodes are merged, which follows optimization rules of ZBDDs [42]. In Fig. 7(b) such nodes are marked with the same color. Fig. 7(c) shows the diagram after merging these nodes. In Step 4.3 optimization rules of BDD are applied. In Fig. 7(c) the left resource node r_3 can be omitted as both of its outgoing edges end at the same terminal node 0. Fig. 7(d) shows the final SEDD.

5 Design Space Exploration

This section describes the procedure to discover a system design optimized in terms of lifetime reliability and execution time. It is discussed in [33–35] that the number of possible solutions increases exponentially with the increase in the size of the problem. The Genetic algorithms are well suited for the problems with a large search space. In addition, it is better suited for multiobjective optimization as compared to various other meta-heuristics approaches, as it is based on the process of natural evolution where multiple objectives are optimized, simultaneously. Here we implement multi-objectives genetic algorithm using GALib [44] framework. In a genetic algorithm (e.g. Fig. 11) a population of designed solutions (refereed as individuals) are transformed using the processes of mutation and crossover. This results in the production of a new population. Only those individuals that are close to the desired goals are kept for the next iteration. The process is continued until the system design with the desired optimization goals is achieved.

5.1 **Individuals of Population**

The nature of the information in the individual should be such that when it gets manipulated by the genetic operators, it must represent a valid system. The system specification graph consists of three portions. Only the set of the task to resource mappings/bindings can be modified, i.e., without affecting the functionality/hardware of the target system. A mapping edges set (MES) along with the SSG represents a system design with a certain reliability and execution time values. A different MES combined with the SSG will represents a different system design with its corresponding reliability and execution time values. *For these reasons the individual in the genetic algorithm is encoded with the set of mapping edges.* The MES, of the individual, combined with the SSG should represent atleast one full TMR SI. This will ensure that the resulting system will be capable of tolerating soft errors at least when there is no hard error in the system. This is referred as “One TMR Rule” in the sequel.

5.2 **Genetic Operators**

In this section the initialization, mutation, and crossover operators for the genetic algorithm are described. The main input to these operators is the TIDD representation for the SIS of all one RSV of the main SEDD. $\{A_1, A_2, A_3, B_1, B_3, B_4\}$ and the TIDD depicted in Fig 8 are used as an example to explain the algorithms for these operators. This TIDD cannot possibility result in a MES which adheres “One TMR Rule”. It is used here due to place reasons. The MES generated by the operators given in Algo. 3, Algo. 4, and Algo. 5, however, adheres to the “One TMR Rule”. Therefore, the steps in the algorithm dealing with TMR will be ignored, when these genetic operators are described, using the TIDD of Fig 8.

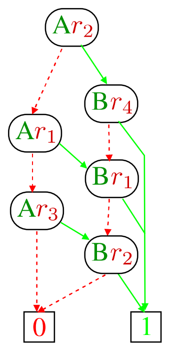


Fig. 8: **Example TIDD**

5.2.1 Initialization Operator

Algorithm 3: DAInitRecur: Initialization Operator

```
   Input: $N$: A TIDD node
   Global: $MpgSt$: MES representing the individual in the Genetic Algorithm
   Output: Unsigned integer value
1  begin
2    if $N$ is the terminal node then return $Task\_Graph_{Size}$;
3    $TskInd \Leftarrow ti_{taskID}$;
4    if exist Uni-Cache[UNI-INIT,N] then                              /* $N$ Already Visited? */
5      return entry Uni-Cache[UNI-INIT,N];                    /* return saved Value */
6    end
7    $ti \Leftarrow N_{ti}$;$TMR \Leftarrow ti \rightarrow$IsTMR();
8    $TIndex \Leftarrow RIndex \Leftarrow LIndex \Leftarrow Task\_Graph_{Size}$;
9    if $ti_{MES} \not\subset MpgSt$ then                   /* $ti_{MES}$: Set of mapping edges of $ti$ */
10     if GetRandomNum() $< 0.5$ then
11       $MpgSt = MpgSt \cup ti_{MES}$;$TIFeasible \Leftarrow$true
12     end
13   else $TIFeasible \Leftarrow$true;
14   if $TIFeasible$ and $TMR$ then
15     $RIndex \Leftarrow$DAInitRecur($N_R$)$-1$;               /* process $N$'s right edge */
16     if $MnTskInd > TIndex$ and $TIndex = TskInd$ then
17       $MnTskInd = TskInd$
18     end
19   end
20   $LIndex \Leftarrow$DAInitRecur($N_L$);
21   if $LTIndex = TskInd$ then $TIndex \Leftarrow LIndex$;
22   if !$TIFeasbile$ then
23     if $TMR$ and $MnTskInd > TskInd$ and $TIndex \neq TskInd$ then
24       $MpgSt = MpgSt \cup ti_{MES}$;
25       $TIndex \Leftarrow$DAInitRecur($N_R$)$-1$;               /* process $N$'s right edge */
26       if $TIndex = TskInd$ then $MnTskInd \Leftarrow TskInd$;
27     end
28   end
29   return insert Uni-Cache[UNI-INIT,$N$, $TIndex$];                    /* store value */
30 end
```

The procedure given in Algo. 3 is the recursive portion of the initialization operator for the genetic algorithm. This function results in the MES *MpgSt* being initialized so that it adheres to the rule of “One TMR Rule”. The random number generator plays an important role. It helps in handling initialization, mutation, and crossover of the MES. Depending on the value produced by the random number generator (i.e., if the number generated is less than 0.5), the TI is made feasible according to the MES *MpgSt* (Step 10 in Algo. 3). The call to the random number generator is, however, only made if the TI is not already feasible according to *MpgSt* (Step 9). In either of the two cases, the function calls itself for the right edge of the node (Step 15). The initialization operator then calls itself for the left edge of the node (Step 20).

There is a small chance the MES resulting from the initialization operator is not following "One TMR Rule" for some task. In such a situation Steps 23 to 26 force the TI to be feasible according to the MES *MpgSt*. The process of forcing is, however, not done if $MnTskInd$ value is zero or less than or equal to the current value of the task index. When the value for $MnTskInd$ is zero, it implies that due to the recursive process the MES already adheres to "One TMR Rule".

Table 4: **Examples to explain the Initialization Operator**

Random Number Generator	MES
L,L,G,G,G,G, . . .	$\{A_2, B_4\}$
L,G,G,G,G,G, . . .	$\{A_2, B_2\}$
L,L,G,G,L,G, . . .	$\{A_1, A_2, B_2\}$

Table 4 depicts different output mapping edges sets generated by the initialization operator, given the different output values of the random number generator, for the TIDD of Fig 8. The letters G and L represent that the value produced by the random number generator is greater/less than 0.5, respectively (see Step 10 in Algo. 3). The example in the first row of the table depicts the possibility of the MES being initialized by *only* the mapping edges representing the first SI in the top down direction. The second row depicts the possibility of the mapping edges of a TIs being *forcibly* selected, if there is no mapping edges corresponding to the task represented by the TI being processed. The third row in the table depicts the possibility of mapping edge B_2 being selected, without the *forcing* procedure being invoked.

5.2.2 Mutation Operator

The procedure given in Algo. 4 is the recursive portion of the mutation operator for the genetic algorithm. The basic structure of the operator is similar to initialization given in Algo. 3.

This function results in a different MES then the original *OldMpgSt* depending upon the probability of mutation (*pmut*) and sequences of numbers generated by the random number generator. The process of mutation is performed according to the value produced by the random number generator (Step 11). When the mutation needs to be performed (Steps 12 to 14), the TI corresponding to the input node is made feasible corresponding to MES *MpgSt*. If it is not already feasible according to the old MES *OldMpgSt*, then the TI is made feasible according to the *not MES NMpgSt*. If it is feasible to the old MES, then the reverse is done given no mutation is required to be performed (Steps 16 to 18).

Consider a TIDD with mixture of TMR and SMR TI nodes. Assume that a node in this TIDD is a TMR node with an SMR node for that same task on its left, such that MES of SMR node is subset of MESS of the TMR node. The random number generator may make the TMR node not feasible according to the MES *MpgSt*, followed by making the SMR node feasible. The second change to the MES is the violation of the initial directives of the random number generator. To avoid such a scenario two MES's are maintained in Algo. 4. If the TMR is made not feasible to MES *MpgSt* then it will be made feasible to MES *NMpgSt*. Thereby eliminating the possibility of making the SMR node feasible to *MpgSt*.

Table 5 depicts different output mapping edges sets produced by the mutation operator given the output of the random number generator and the "Old MES". M represents that the mutation is performed for the node whereas the N represents otherwise. The example given in the first row of the table depicts the possibility of the first SI of the SIS represented in Fig. 8 being selected. The example given in the third row, depicts the possibility of the "forcing" portion of the function being invoked (Steps 30 to 36).

Table 5: **Examples to explain the Mutation Operator**

Old MES	Random Number Generator	MES
$\{A_1, B_1\}$	M, M, N, N, N, N	$\{A_2, B_4\}$
$\{A_1, B_1\}$	M, M, N, N, N, N	$\{A_1, A_2, B_1, B_2\}$
$\{A_1, A_2, B_1\}$	M, M, N, N, N, N	$\{A_3, B_2\}$

Algorithm 4: DAMutRecur: Mutation Operator

```
   Input: $N$: A TIDD node
   Global: $MpgSt$, $NMpgSt$, $OldMpgSt$ and $pmut$.
   Output: Unsigned integer value
 1 begin
 2   if $N$ is the terminal node then return $Task\_Graph_{Size}$;
 3   if exist Uni-Cache[UNI-MUT,N] then                                   /* $N$ Already Visited? */
 4     return entry Uni-Cache[UNI-MUT,N];                      /* return saved Value */
 5   end
 6   $ti \Leftarrow N_{ti}; TMR \Leftarrow ti \rightarrow$IsTMR();
 7   $TIndex \Leftarrow RIndex \Leftarrow LIndex \Leftarrow Task\_Graph_{Size}$;
 8   if $ti_{MES} \subseteq MpgSt$ then $TIFeasible \Leftarrow$true ;
 9   else if $ti_{MES} \subset NMpgSt$ then $TIFeasible \Leftarrow$false ;
10   else
11     if GetRandomNum() $< pmut$ then                                    /* Do Mutation */
12       if $TIFeasible \Leftarrow ti_{MES} \not\subseteq OldMpgSt$ then
13         $NMpgSt = NMpgSt \cup ti_{MES}$
14       else $MpgSt = MpgSt \cup ti_{MES}$;
15     else                                                               /* No Mutation */
16       if $TIFeasible \Leftarrow ti_{MES} \subseteq OldMpgSt$ then
17         $MpgSt = MpgSt \cup ti_{MES}$
18       else $NMpgSt = NMpgSt \cup ti_{MES}$;
19     end
20   end
21   if $TIFeasible$ and $TMR$ then
22     $RIndex \Leftarrow$DAMutRecur($N_R$)$-1$;               /* process $N$'s right edge */
23     if $MnTskInd > ti_{taskID}$ and $TIndex = ti_{taskID}$ then
24       $MnTskInd = ti_{taskID}$
25     end
26   end
27   $LIndex \Leftarrow$DAMutRecur($N_L$);
28   if $LTIndex = ti_{taskID}$ then $TIndex \Leftarrow LIndex$;
29   if $!TIFeasbile$ then
30     if $TMR$ and $MnTskInd > ti_{taskID}$ and $TIndex \neq ti_{taskID}$ then
31       $MpgSt = MpgSt \cup ti_{MES}$;
32       $NMpgSt = NMpgSt \cap ti_{MES}$;
33       $tindex \Leftarrow$DAMutRecur($N_R$)$-1$;               /* process $N$'s right edge */
34       if $TIndex = ti_{taskID}$ then $MnTskInd \Leftarrow ti_{taskID}$;
35     end
36   end
37   return insert Uni-Cache[UNI-MUT,$N$, $TIndex$];                 /* store value */
38 end
```

5.2.3 Crossover Operator

Algo. 5 depicts the recursive portion of the crossover operator for the genetic algorithm. This function combines the mapping edges sets of the two input individuals in such a way, that the resulting MES contains the elements of both input mapping edges sets and adheres to "One TMR Rule". The non-recursive portion of the crossover operator populates the *BoolVec* using the output of the random number generator. The size of this vector is equal to the size of the input task graph. If the entry in the *BoolVec* corresponding to the task of the input node's TI is *true*, then MES *true* is considered to be the primary MES (Step 9), and the second set is considered the secondary MES. The reverse is done if the respective entry is *false*.

Steps 11 to 14 make the TI corresponding to the input node feasible to the MES $MpgSt$ if it is feasible to the selected MES. If any forcing must be done (Steps 25 to 31), then it is performed using the secondary MES.

The first and second rows in Table 6 depict the normal operation of the crossover function. The third row of the table represents the possibility of the "forcing" portion of the algorithm being invoked (Steps 25 to 31). The portions of initialization and mutation functions, that force the TIs to be feasible according to MES $MpgSt$. It requires that the TI being forced should be feasible according to the MES of the *original system specification graph*. The crossover function, on the other hand, requires that the TI being forced should be feasible to the selected *secondary* MES. This is the reason why in the third example mapping edge B_1 is selected instead of B_2, which would have been the case for the initialization and mutation functions.

Table 6: **Examples to explain the Crossover Operator**

$MpgSt1$	$MpgSt2$	$BoolVec$	$MpgSt$
$\{A_2, B_4\}$	$\{A_1, B_1\}$	[T, F]	$\{A_2, B_1\}$
$\{A_2, B_2\}$	$\{A_1, B_1\}$	[F, T]	$\{A_1, B_2\}$
$\{A_2, B_4\}$	$\{A_1, B_1\}$	[F, T]	$\{A_1, B_1\}$

Algorithm 5: DAXOverRecur: Crossover Operator

```
    Input: $N$: A TIDD node
    Global: $MpgSt$: Mapping set representing the individual in the genetic algorithm
    Output: Unsigned integer value
 1  begin
 2    if $N$ is the terminal node then return $Task\_Graph_{Size}$;
 3    $TskInd \Leftarrow ti_{taskID}$;
 4    if exist Uni-Cache[UNI-XOVER,N] then                          /* $N$ Already Visited? */
 5      return entry Uni-Cache[UNI-XOVER,N];                          /* return saved Value */
 6    end
 7    $ti \Leftarrow N_{ti}$;$TMR \Leftarrow ti \rightarrow$IsTMR();
 8    $TIndex \Leftarrow RIndex \Leftarrow LIndex \Leftarrow Task\_Graph_{Size}$;
 9    $ParAMpgSt \Leftarrow MpgSt1$, if $BoolVec[TskInd]$ =true otherwise $MpgSt2$;
10    $ParAMpgSt \Leftarrow MpgSt1$, if $BoolVec[TskInd]$ =false otherwise $MpgSt2$;
11    if $ti_{MES} \subseteq MpgSt$ then $TIFeasible \Leftarrow$true;
12    else
13      if $ti_{MES} \subseteq ParAMpgSt$ then
14        $MpgSt = MpgSt \cup ti_{MES}$;$tiFeasible \Leftarrow$true
15      end
16    end
17    if $TIFeasible$ and $TMR$ then
18      $RIndex \Leftarrow$DAXOverRecur($N_R$)$-1$;                    /* process $N$'s right edge */
19      if $MinTskInd > TIndex$ and $TIndex = TskInd$ then
20        $MinTskInd = TskInd$
21      end
22    end
23    $LIndex \Leftarrow$DAXOverRecur($N_L$);                           /* process $N$'s left edge */
24    if $LTIndex = TskInd$ then $TIndex \Leftarrow LIndex$;
25    if !$TIFeasbile$ and $TMR$ and $MinTskInd > TskInd$ then
26      if $TIndex \neq TskInd$ and $ti_{MES} \subset ParBMpgSt)$ then
27        $MpgSt = MpgSt \cup ti_{MES}$;
28        $TIndex \Leftarrow$DAXOverRecur($N_R$)$-1$;                  /* process $N$'s right edge */
29        if $TIndex = TskInd$ then $MinTskInd \Leftarrow TskInd$;
30      end
31    end
32    return insert Uni-Cache[UNI-XOVER,N, $TIndex$];                   /* store value */
33  end
```

5.3 Evaluation of Individuals

As depicted in Fig. 11, the process of evaluation of each individual in the population of the genetic algorithm needs to be performed after the initialization, mutation, and crossover operators. The attributes of the system must be evaluated, resulting from combining the MES stored as individual with the system task and resource graphs.

Given the information of the MES stored in the individual of the genetic algorithm, the evaluation is composed of the following steps.

1. First, the system specification graph, constructed by appending MES to the system task and resource graph.
2. The resulting SSG is used to construct the SEDD using the procedure detailed in Sec. 4.2.
3. Given the decision diagram, one of the algorithms in the following section is used to evaluate the attributes of the system.

It was shown in [5] that the time to construct a SEDD diagram is greater than the time required to evaluate the lifetime reliability of the system. The process of evaluating the system attribute values given the MES represented in the individual of the genetic algorithm is going to take quite a lot of time while following the three steps mentioned above.

The second possibility is to make a copy of the original SEDD. The TI nodes of the SEDD that are not feasible to the MES of the individual are removed. This is followed by the algorithms to evaluate the attributes of system. The SEDD constructed in this manner will be faster as compared to the approach described in the previous paragraph. This will simplify the evaluation of the individual in the population.

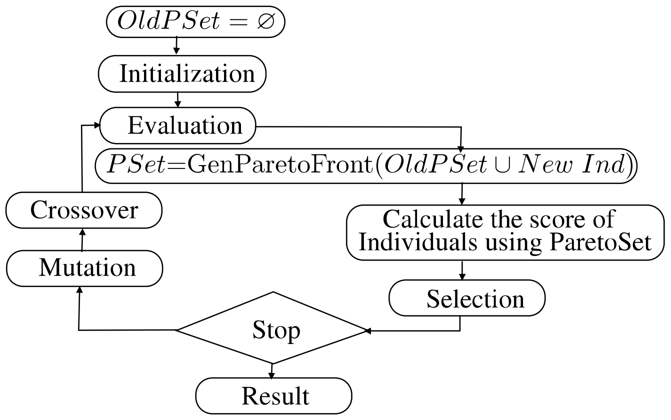
The process can be further simplified by ignoring the TI nodes that are not feasible according to the MES during the evaluation of system attribute instead of "*physically*" removing those nodes. This procedure is detailed in Sec. 6.

5.4 Selection Operator: Assigning Score values to Individuals

A uni-objective genetic algorithm is a lot simpler to conceive and to implement as compared to the multi-objective genetic algorithms. These algorithms can use the evaluated value of the single objective as the score for the individuals. The algorithm discussed in Section 6.2, for example, uses the execution time attribute of the individual as it score value. Thereafter, the selection process chooses the individuals with low scores, i.e., low execution time values, to be selected to form the next population. It is, however, not easy to assign score values for the individuals of the multi-objective genetic algorithm. Aggregation of the objectives is a possible method of assigning score/fitness value to the individuals in a multi-objective genetic algorithm. In this method each individual is assigned a weight $w_i \in]0,1[$, and $\sum w_i = 1$. The evaluated values of each objective are multiplied by the respective weights and are then accumulated to form the value, which is used as the score for the individual. This method forces the genetic algorithm to focus its search in only one particular direction. This means that the designer has to be exactly sure about the proportion of each objective in the final solution. This is not always the case and it is difficult to set meaningful weights for the different objectives that can lead to "less than ideal" solution.

The genetic algorithm presented in this work produces a set of Pareto-optimal set of DAs with lifetime reliability and execution time as design criteria. Two DAs that have same values for all design objectives except for one, cannot be simultaneously part of the Pareto-set. However, if they are dissimilar in more than one design objectives then they might be part of the set. Set of DAs is considered a Pareto-optimal if for each member set, there exists no other feasible DA which would decrease some objective without causing simultaneous increase in at least one other objective. After the generation of a new population every individual in it is tested to see if it can be added in the Pareto-set (also referred to as Pareto-front). Given the individuals in the Pareto set for a current generation in the genetic algorithm, the score/fitness value for each individual of the population can be assigned. The individual of the population closer in "distance" to the Pareto front is assigned a "better" value as compared to the one with greater distance from the Pareto front. Given the individual I of the population and the Pareto set P the value "score(i)" is given as:

$$\text{score}(i) = \min_{p_j \in P}[||i.Rel - p_j.Rel| + |i.ExTm - p_j.ExTm||] \tag{3}$$



$OldPSet = \varnothing$
Initialization
Evaluation
$PSet$=GenParetoFront($OldPSet \cup New\ Ind$)
Crossover
Mutation
Calculate the score of Individuals using ParetoSet
Selection
Stop
Result

Fig. 9: **Detailed Outline of Outer Genetic Algorithm**

The values thus assigned to the individuals of the old population and the newly created individual are used to discard the individuals with "worst" scores, thus forming the new population. The process is continued until one of the Pareto set is not modified for a certain number of generations.

6 Evaluation of Lifetime Reliability and Execution of a DA

Based on the system error decision diagram and mapping edges set the lifetime reliability and execution time of the system alternative can be derived. In this section an efficient algorithm for the system attributes calculation which operates on the SEDD extracted in Section 4.2, is presented. Classically, a system represented as event or fault tree [45] is converted into a BDD. Then a simple algorithm is used on this BDD to calculate the reliability of the system, e.g., [46]. Such solutions apply recursive reliability calculation on the decision diagram starting from the root node. The reliability of a node N, which is denoted as $R(N)$, depends on the specific reliability of this node $R_{sp}(N)$, and on the reliabilities of right and the left successor nodes N_{R} and N_{L}, respectively:

$$R(N) := R_{sp}(N) \times R(N_{\text{R}}) + (1 - R_{sp}(N)) \times R(N_{\text{L}}) \tag{4}$$

This equation calculates the average reliability value for the system being in various states. Eq. 5, which is similar to Eq. 4, calculates the average(or lifetime) execution time value for the system being in various states.

$$T(N) := R_{sp}(r) \times T(N_{\text{R}}) + (1 - R_{sp}(r)) \times T(N_{\text{L}}) \tag{5}$$

Similar to Eq. 4, Eq. 5 requires time value of 0.0 to be returned for the left terminal node. The algorithm to evaluate the lifetime reliability and execution time values for a DA is detailed in Sec. 6.3.

It is noted that the system error decision diagram is not a pure BDD, rather a mixed BDD-ZBDD diagram. The upper portion of the SEDD can be considered as a compact resource state tree, from which the resource state vectors can be generated. In the SEDD, the ZBDD representations of the SIS corresponding to the resource state vectors are related directly to these vectors. Eq. 4 and 5 can be used on the upper portion of the SEDD as it consists of BDD nodes only. During recursion the algorithm represented by these equations will encounter TI nodes. The first node encountered after a number of resource nodes is the root node of the ZBDD representing a SIS. When the system is in the state as described by the RSV the SI with best reliability and execution time value must be chosen to represent the system. The procedure to find the SI with best attributes from the SIS has the order of $\mathcal{O}(|SIS||V_t|)$, where $|SIS|$ is an exponential function of number of tasks and resources in the SSG. In contrast, the algorithm represented by the recursive Eq. 6 is of the order $\mathcal{O}(|G_N|)$, where $|G_N|$ is the count of the nodes in the ZBDD representing SIS with N as the root node. In addition, in [5] it was shown that $|SIS| \gg |G_N|$.

$$R(N) := \max(R_{sp}(ti) \times R(N_{\text{R}}), R(N_{\text{L}})) \tag{6}$$

The proof that the Eq. 6 produces the reliability of the SI that has the best reliability value within the SIS is provided in [5]. An equation to calculate the execution time of the SI with best execution time value in the SIS, can only be constructed if the SI and hence its corresponding task graph is linear. The majority of the task graphs, however, are not linear and have various *fork* and *join* structure. An exhaustive search procedure to find the reliability and execution time values for best SI for the working resource state vector is therefore not feasible. A genetic algorithm is, therefore, employed to find the time attributes of the best SI (described in Section 6.2).

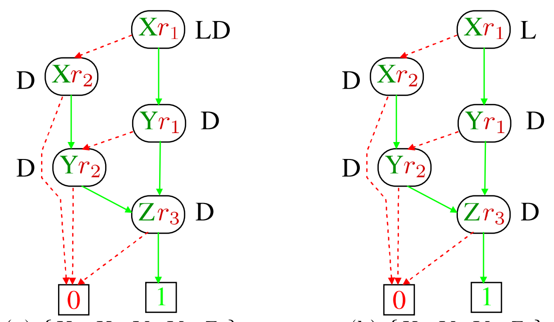
6.1 **Reliability and Time attribute for best SI**

Reliability has a higher precedence over the time attribute for the selection of best SI from the SIS. There may possibly be a number of system instances amongst the SIS of the working resource state vector that have the same highest reliability value. Therefore, instead of constructing a multiobjective genetic algorithm at the level of SIS, a procedure is proposed to *mark* all the system instances that have the highest value of reliability amongst the system instances that are feasible according of MES of the individual.

Algo. 6 returns the reliability value of the SI, which has the highest reliability value that are feasible according to MES, corresponding to a working resource state vector represented by a ZBDD. The algorithm also *mark* the nodes in the ZBDD indicating the paths that represent the SI with the highest reliability value. The conditional statement in Step 7 of Algo. 6 make the respective algorithms passover the nodes with the TIs not feasible to the MES and to mark their direction variable as DIR_LEFT, so that any other algorithm that considers the direction variables also ignores those nodes.

Algorithm 6: InitDirMapSIS: Initialize Direction Variable in each ZBDD Node

```
Input: $N$: Node representing ZBDD for SIS of a WRSV
Global: $MpgSt$: Mapping edges set
Output: $Rel$: Max-Reliability value for the SIS representing the WRSV
begin
  if $N$ is the left terminal node then return 0.0;
  if $N$ is the right terminal node then return 1.0;
  if exist Uni-Cache[UNI-INIT-DIR-MAP, $N$] then                          // $N$ Already Visited?
    return entry Uni-Cache[UNI-INIT-DIR-MAP, $N$];                        // return saved Value
  end
  if $N_{tiMES} \subseteq MpgSt$ then
    $Rel \Leftarrow R_{sp}(N_{ti}) \times InitDirMapSIS(N_R)$;              /* process right edge */
    $RelE \Leftarrow InitDirMapSIS(N_L)$;                                  /* process left edge */
    if $Rel = RelE$ then $Dir \Leftarrow$DIR_LEFT_DOWN;
    if $Rel > RelE$ then $Dir \Leftarrow$DIR_DOWN;
    else $Rel \Leftarrow RelE$;$Dir \Leftarrow$DIR_LEFT;
  else
    $Rel \Leftarrow InitDirMapSIS(N_L)$;$Dir \Leftarrow$DIR_LEFT            // process left edge
  end
  if $Rel = 0.0$ then $Dir \Leftarrow$DIR_ZERO
  $N_{Dir} \Leftarrow Dir$
  return insert Uni-Cache[UNI-INIT-DIR-MAP, $N$, Rel];                    // store value
end
```



(a) $\{X_1, X_2, Y_1, Y_2, Z_3\}$ (b) $\{X_2, Y_1, Y_2, Z_3\}$

Fig. 10: **Example TIDD and Specific Time Values of its TIs**

Consider the SSG of Fig. 2 and the soft reliability values for its mapping edges given in Table 2. The SIS for the RSV $r_1r_2r_3$ consists of three system instances. Their soft reliability values are given in Table 3. The system instances (X_1, Y_1, Z_3) and (X_2, Y_2, Z_3) have the highest reliability value amongst the SIS. The TIDD of Fig. 10(a) is the ZBDD representation of the SIS for RSV $r_1r_2r_3$. It has been properly labeled/marked so that any traversal algorithm, considering the markings introduced in this section, only consider these two system instances. The nodes of TIDD of Fig. 10(a) with TIs X_2, Y_1, Y_2, and Z_3 are labeled DIR_DOWN. These nodes, except for the node Y_1, have their left edge as the left terminal node. The traversal algorithm that considers these labels should also consider the left edge of the node B_1 as the left terminal node. Otherwise, it will also consider the SI (X_1, Y_2, Z_3) which is not one of the highest reliability SI. If (X_2, Y_2, Z_3) is the only highest reliability SI or the MES does not include X_1 then the node X_1 will be labeled DIR_LEFT as in the TIDD of Fig. 10(b). Any algorithm traversing this TIDD will ignore the system instances (X_1, Y_1, Z_3) and (X_1, Y_2, Z_3).

6.2 SI with Minimum Time Attribute

In this section the genetic algorithm to determine the execution time of the SI from a set of system instances that has the minimum execution time, is described. In a genetic algorithm a set of possible solutions (collectively called as "population") is modified by crossover, mutation, and selection operations. The selection operation is controlled by means of the quality of each individual (a single solution) in the population, which in this situation means only those solutions are selected to be part of the next generation that have least values of execution time. The implementation outline depicted in Fig. 11 is based on the GALib package [44]. The framework of GALib is used due to its easy implementation. It is only needed to specify the representation of the individual and to construct the needed operators (Initialize, Mutate, and Crossover) to yield a functioning genetic algorithm.

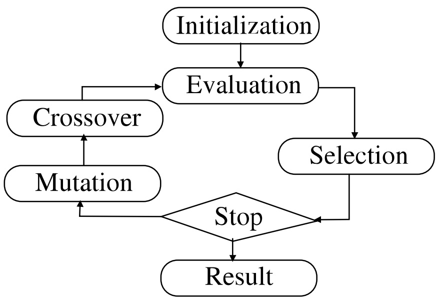


Fig. 11: **Outline of Genetic Algorithm**

6.2.1 Individuals in the Genetic Algorithm

An individual of the population in the genetic algorithm represents a possible solution. The result of the genetic algorithm is the execution time of the best SI, therefore the individual of the genetic algorithm should be able to represent all the information required for the calculation of the execution time value of the SI. A TPV along with the actual SI is required to determine the specific time attribute of the SI. Therefore, the TI vector is part of the information stored in the individual of the population. The SI, however, cannot directly be part of the individual. This is because the operators of the genetic algorithm (i.e., mutation, crossover) might result in an *invalid system instances*. The SI is instead represented by the DV. The direction vector (DV) is an integer vector representing the traversal through the input ZBDD corresponding to the SIS. The size of the DV is equal to the number of the tasks in the system task graph ($|V_t|$). Using the information in a DV, a SI can be represented.

Consider the TIDD given in Fig. 10(a). If all the system instances represented by this TIDD have equal reliability value then all the nodes of this TIDD will be labeled LEFT_DOWN. The SI in the SIS represented by this ZBDD is assumed to have equal reliability values, i.e., all of the constituent nodes of the ZBDD are labeled LEFT_DOWN. The system instances (X_1, Y_1, Z_3), (X_1, Y_2, Z_3), and (X_2, Y_2, Z_3) are represented by the DirVecs (0,0,0), (0,1,0), and (1,0,0), respectively. Given the DV and the node representing the ZBDD, the procedure in Algo. 7 generates the SI which can be used along with the TPV to evaluate the specific time value of the SI.

6.2.2 Genetic Operators for Direction Vector

This section describes the three genetic algorithm operators (i.e., initialize, mutation, and crossover). The individual representation of this genetic algorithm consists of two data structures. Therefore, operators

Algorithm 7: DirVec2SI: SI represented by DV

```
  Input: $N$: Node representing ZBDD for SIS of a WRSV; $DV$
  Output: $TIVector$: Vector of TIs representing the system instances
1 begin
2   for $i$ 1 to $Task\_Graph_{size}$ do
3     $j \Leftarrow 0$;
4     while $DV[i] \neq j$ do  $N \Leftarrow N_{\mathrm{SpL}}$;$j \Leftarrow j + 1$;
5     $TIVector[i] \Leftarrow N_{ti}$;
6     $N \Leftarrow N_{\mathrm{SpR}}$;
7   end
8   return $TIVector$;
9 end
```

for both of them (i.e., TPV and DV) have to be considered. For the TPV data structure, the operators for genetic algorithm for the salesman problem [47], whose various implementations are readily available from the Internet and can be used with almost no modification. The output TPV generated by these operators might not represent the correct data dependence in the task graph. However, the procedure in Algo. 1 is able to circumvent this problem. Please note that the steps in the algorithms, of the genetic operators, implementing the dynamic programming are not shown due to their similarity with the previously given algorithms. The ZBDD representation of the SIS for RSV $r_1r_2r_3$ of SSG in Fig. 2 will be used as the input TIDD to help explain the genetic operators. In addition, all the nodes connected with left terminal nodes are labeled DOWN, and the other ones are labeled LEFT_DOWN, implying that all the system instances represented by the TIDD are assumed to have same reliability value.

Table 7: **Output of Initialization Operator**

Random Number × 100	Direction Vector	System Instance
45, 24, 26	$(0, 0, 0)$	(X_1, Y_1, Z_3)
62, 11, 33, 25	$(0, 1, 0)$	(X_2, Y_2, Z_3)
91, 75, 27, 27, 10, 20	$(0, 1, 0)$	(X_2, Y_2, Z_3)

Initialization Operator

The initialization function is used only once in the genetic algorithm for each individual of the population. Nevertheless, it is one of the important operators of the genetic algorithm. The function in Algo. 8 initializes the DV that represents a SI as a path through the TIDD from the root node to the right terminal node. The recursive function returns a Boolean value. If this value is true, then it indicates that the set of the combination of TIs represented by the TIDD with the input node as the root node contains portion of the chosen SI. It also indicates that the DV is properly initialized.

Table 7 depicts different DVs initialized by the initialization operator (given in Algo. 8) for different sequences of random numbers (Step 6). The first two rows of the table depict the procedure of initializing the DV following the directives of the random number generator (RNG). The third row shows the possible scenario when Step 10 overrides the directives of the number generator and selects node X_2 as there is no node to its left.

Mutation Operator

Algo. 9 depicts the mutation function for the DV. This function changes the DV depending upon the probability of mutation ($pmut$) and sequences of numbers generated by the random number generator. The DV modified by Algo. 9 represents a valid SI. Mutation is one of the important operators of the

Algorithm 8: DirVecInit: Initialization Operator.

```
   Input: $N$: TIDD node; $index$: Integer
   Global: $DV$: Direction Vector
   Output: Boolean value
 1 begin
 2   if $N$ is the special left terminal node then return false;
 3   if $N$ is the special right terminal node then return true;
 4   $level \Leftarrow N \rightarrow$GetTask() $\rightarrow$GetVecIndex();
 5   $Connected \Leftarrow$false;
 6   if GetRandNum() $< 0.5$ and ($Connected =$DirVecInit($N_{SpR}, 0$)) then
 7     $DV[level] \Leftarrow index$
 8   else $Connected =$DirVecInit($N_{\mathrm{SpL}}, index + 1$);          // N's special left edge
 9   if !$Connected$ and ($Connected =$DirVecInit($N_{SpR}, 0$)) then
10     $DV[level] \Leftarrow index$
11   end
12   return Connected;
13 end
```

genetic algorithm. Without the mutation there is a possibility of a premature convergence of the genetic algorithm, meaning the genetic algorithm does not return the "best" individual.

Algorithm 9: DirVecMut: Mutation Operator

```
   Input: $N$: A TIDD node; $index$: An integer
   Global: $DV$: Direction Vector; $pmut$: Probability of Mutation
   Output: A Boolean value
 1 begin
 2   if $N$ is the special left terminal node then return false;
 3   if $N$ is the special right terminal node then return true;
 4   $level \Leftarrow N \rightarrow$GetTask() $\rightarrow$GetVecIndex();
 5   $Cntd \Leftarrow$false;                                             /* Connected⇐false */
 6   if GetRandNum() $< pmut$ then                                       /* Mutation Portion */
 7     if $DV[level] = index$ then $Cntd =$DirVecMut($N_{\mathrm{SpL}}, index + 1$);
 8     else if ($Cntd =$DirVecMut($N_{SpR}, 0$)) then $DV[level] \Leftarrow index$;
 9   else                                                                /* Non-Mutation Portion */
10     if $DV[level] = index$ and ($Cntd =$DirVecMut($N_{SpR}, 0$)) then
11       $DV[level] \Leftarrow index$
12     else $Cntd =$DirVecMut($N_{\mathrm{SpL}}, index + 1$);              /* N's special left edge */
13   end
14   if !$Cntd$ then
15     if ($Cntd =$DirVecMut($N_{SpR}, 0$)) then $DV[level] \Leftarrow index$;
16     else $Cntd =$DirVecMut($N_{\mathrm{SpL}}, index + 1$);              /* N's special left edge */
17   end
18   return Cntd;
19 end
```

Table 8 depicts different results of the mutation operator for various DVs, given the various sequences of numbers produced by RNG. The third column in the table contains a sequence of letters M and N, instead of actual numbers like in Table 7. M implies that the DV entry for the node under process is going to be modified, whereas N indicates otherwise. When the output of RNG is less then *pmut* variable, then the mutation operator changes the DV entry of the respective task (Steps 7 and 8). Consider the first row in Table 8. The mutation portion is initiated for the root node. The original DV entry for the task X denotes X_1 (which is the TI for this node), therefore to mutate the entry the mutation operator for left edge of the node, thereby ignoring the node. The mutation portion of the algorithm is again initiated for the node X_2, and it is not the part of the original DV, it is made to represent the DV entry for task X.

When the random number generated is greater than *pmut* variable then the mutation operator does not change the entry of the DV for the respective task (Steps 10 to 12 in Algo. 9). Consider the example given in the second row of Table 8. For the root node non-mutation portion is initiated and the function calls the mutation operator for the right edge of the node, so as to preserve the old entry of the DV for the respective task X. On the other hand, for example in the third row in Table 8, the non-mutation

Table 8: **Output of the Mutation Operator**

Input		Relation	Output	
DV	SI		DV	SI
$(0,0,0)$	(X_1,Y_1,Z_3)	M, M, N, N	$(1,0,0)$	(X_2,Y_2,Z_3)
$(0,0,0)$	(X_1,Y_1,Z_3)	N, M, N	$(0,1,0)$	(X_1,Y_2,Z_3)
$(1,0,0)$	(X_2,Y_2,Z_3)	N, M, N, N	$(1,0,0)$	(X_2,Y_2,Z_3)

portion calls the mutation function for the left node. This is because the TI for the node is not part of the original DV entry for task X. For the node with TI X_2 mutation portion is initiated and the mutation function for left node is called. However, there is no TI node to the left of this node, therefore, the directives of RNG will be ignored for this node. Steps 15 to 16 provide a fail-safe return and make sure that the mutation operator always generates a valid DV.

Algorithm 10: DirVecXOver: Recursive Portion of Crossover Operator

```
    Input: $N$: A TIDD node; $index$: An integer
    Global: $DV$: Direction Vector; $DVec1$, $DVec2$: Direction vectors of Parent 1 and 2.
    Output: A Boolean value
 1  begin
 2    if $N$ is the special left terminal node then return false;
 3    if $N$ is the special right terminal node then return true;
 4    $level \Leftarrow N \rightarrow$GetTask() $\rightarrow$GetVecIndex();
 5    $Cntd \Leftarrow$false;                                        /* Connected⇐false */
 6    if $CrossOverTemp[level]$ then $DVec \Leftarrow DVec1$;
 7    else $DVec \Leftarrow DVec2$;
 8    if $DV[level] = index$ and ($Cntd =DirVecXOver(N_{SpR}, 0)$) then
 9      $DV[level] \Leftarrow index$
10    else $Cntd =$DirVecXOver($N_{\mathrm{SpL}}, index + 1$);      /* N's special left edge */
11    if $!Cntd$ and ($Cntd =DirVecXOver(N_{SpR}, 0)$) then
12      $DV[level] \Leftarrow index$
13    end
14    return $Cntd$;
15  end
```

Crossover Operator

Algo. 10 depicts the recursive portion of the *Crossover Operator* for the *DV*. This function combines the DVs of two individuals in such a way that the resulting DV represents a valid SI. The crossover operator is main reason why this algorithm is called "genetic". This operator represents the process taking place in the multi cellular heterosexual living organisms. The two parents provide portion of the genetic material that result in the production of a new organism, which has the properties of both of its parents. The non-recursive portion of the crossover operator initializes the Boolean vector $CrossOverTemp$ using the random number generator. In Steps 6 and 7 of Algo 10 the DV entry is either chosen from one of the parents depending upon the corresponding entry of the $CrossOverTemp$ vector. For example the boolean vector for first row in Table 9 is [**false,true,false**] which results in the DV of $(0,0,0)$. Steps 8 to 10 calls the recursive crossover function to right or left edge based on the information in the inherited DV entry for the task. However, there might be problems like the ones that were observed in the mutation operator. Steps 11 and 12 handle such problems.

Table 9 shows two examples of the recursive crossover operator. The first two columns depict the DV of the two parent individuals. The third column shows the temporary DV, which results by applying the information in the $CrossOverTemp$ Boolean vector to the input parent DVs. The fourth column shows the output DV of the crossover operator corrected by Steps 11 and 12 in Algo. 10 (if there is any need of correction).

Table 9: **Output Crossover Operator**

Input DV		Temp DV	Output DV	SI
Parent 1	Parent 2			
$(1,0,0)$	$(0,1,0)$	$(0,0,0)$	$(0,0,0)$	(X_1,Y_1,Z_3)
$(1,0,0)$	$(0,1,0)$	$(1,1,0)$	$(1,0,0)$	(X_2,Y_2,Z_3)

6.2.3 Evaluation of the Individual

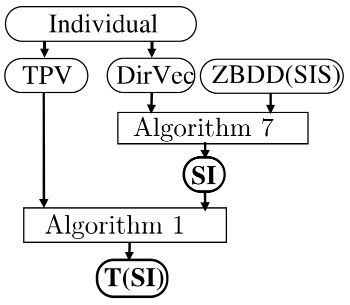


Fig. 12: **Data Flow for Execution Time Attribute for Individual**

Before the selection process in the genetic algorithm, each individual in the population contains a TPV and a DV. The time attribute evaluation algorithm in Algo. 1, however, requires the *SI*. It can be easily generated using the function in Algo. 7 (as depicted in Fig. 12) with DV and the root node to ZBDD representing the input SIS to the genetic algorithm. Algo. 1 then uses the SI and the TPV to get the time attribute value for the SI. When all the individuals in the population are evaluated, then selection process can start.

The individuals in the current population and the newly constructed individuals that have the least execution time values are selected (selection operator of Fig 11) to be part of the next generation. The genetic algorithm terminates (stopping criteria of Fig 11), after processing a certain number of populations or when all individuals in the population have converged to a single individual.

6.3 Lifetime Reliability and Execution Time of the System DA

Fig. 13 depict the data flow for calculating the lifetime reliability and execution time of the system as described by the original system specification graph and the given MES. The function *RelExTmMapSEDD* in Algo. 11 evaluates the Lifetime reliability and execution time values for the system represented by the MES of the individual, with the SEDD constructed using the original system specification graph as the input. This procedure implements Eq. 4 and 5 on the resource nodes of the SEDD. This procedure calls Algo. 6 function for every TI node encountered during the traversal. This function not only calculates the reliability value, but also marks the TIDD with the directional labels indicating the path representing the system instances with the highest reliability value amongst the SIS represented by the TIDD. The

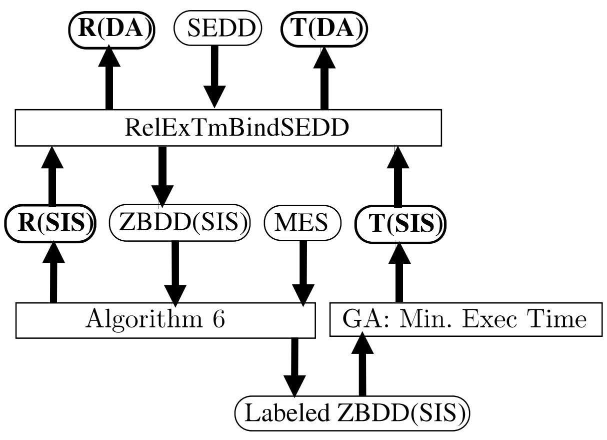


Fig. 13: **Data Flow for Lifetime Reliability and Execution Time of DA using SEDD**

genetic algorithm described in Sec. 6.2 then calculates the time attribute of the SI with minimum time value.

Algorithm 11: RelExTmMapSEDD: Lifetime Reliability and Execution Time

```
    Input: SEDD, a node $N$, $MpgSt$: MES/individual in the main Genetic Algorithm
    Output: $Rel$, $ExTm$: Lifetime reliability and execution time of the system
 1  if $N$ is the left terminal node then return (0.0, 0.0);
 2  if exist Uni-Cache[UNI-SEDD-REL-EXTM,$N$] then                    /* $N$ Already Visited? */
 3      entry Uni-Cache[UNI-SEDD-REL-EXTM,$N$,Rel,ExTm];
 4      return ($Rel$, $ExTm$);                                  /* return saved Values */
 5  end
 6  if $N$ represents a resource then
 7      RelExTmMapSEDD($N_R$, $RelRight$, $ExTmRight$);               /* process right edge */
 8      RelExTmMapSEDD($N_L$, $RelLeft$, $ExTmLeft$);                 /* process left edge */
 9      $RRslt \Leftarrow R_{sp}(r) \times RelRight + (1 - R_{sp}(r)) \times RelLeft$;
10      $ExRslt \Leftarrow R_{sp}(r) \times ExTmRight + (1 - R_{sp}(r)) \times ExTmLeft$
11  else                                                              /* $N$ represents a TI */
12      RelExTmMap($N$, $MpgSt$, $SysRel$, $SysExTm$);                /* Start the Genetic Algorithm */
13  end
14  insert Uni-Cache[UNI-SEDD-REL-EXTM,$N$,SysRel,SysExTm)];          // store values
15  return ($SysRel$, $SysExTm$);
```

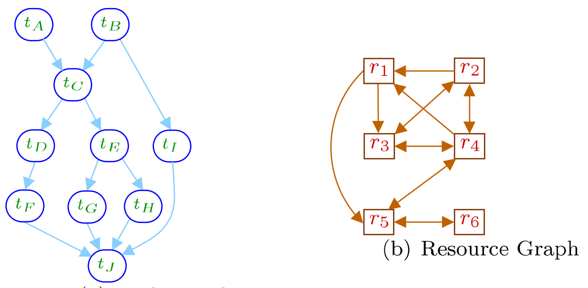
7 Case Study

To evaluate the functionality and the performance of the proposed approach, it was applied to a real-world system specification provided in [48]. This work presents a cruise control system (CCS) for modern cars. The CCS maintains the speed of the vehicle between 35 km/h and 200 km/h. It offers an interface to increase or decrease the cruise speed. The cruise control operation is disabled when brake is applied. Many of the tasks in the task graph given in [48] are combined to form a compound task. This does not affect the functionality of the system as constituent tasks were being executed on same processing resources. The resulting task graph is given in Fig. 14(a) while Fig. 14(b) represents the resource graph. Due to industrial nondisclosure requirements, neither the full tasks code nor the absolute execution time values were available. Only their approximate ratios were given. We therefore take the execution time

Table 10: Execution Time attribute of the tasks on the resources

Tasks	Time[s]
t_A	0.096
t_B	0.08
t_C	0.08
t_D	0.04
t_E	0.056
t_F	0.024
t_G	0.04
t_H	0.024
t_I	0.136
t_J	0.04

values shown in Table 10 and denote them in relative time units. Despite these restrictions, this approach still allows for a high-level assessment of the overall dynamic behavior of the system. The soft-error rate used in these experiments is 0.01 errors/sec.



(a) Task Graph (b) Resource Graph

Fig. 14: **Task and resource graphs for the case study**

To estimate the performance of the presented approach the construction time and the memory required was measured on an Intel Pentium 4 Dual Core 3.00 GHz machine with 2.00 GB memory. The SEDD constructed using the SSG of the case study consists of 12465 nodes. Whereas, a pure BDD solution has a higher memory requirement as it consists of 84595 nodes. In addition, the time required to construct SEDD was 0.134727273 sec, whereas 0.411909091 sec are required to construct a pure BDD model. It was found that original SSG results in 63 RSVs representing a non-failed working system. In addition, the total number of system instances in the SIS for all-one RSV is 13876438948.

This SEDD is used as an input to the nested genetic algorithm described in Sec. 5 and 6, which generates the DAs with various lifetime reliability and execution time. Kindly, note that time required to construct the Pareto set was 278 sec, whereas the genetic algorithm executed for the given SSG from other technique in the literature [27] requires 219 minutes.

Fig. 15 shows the Pareto front for the SSG for this case study. It consists of 40 DAs. The bottom left point on the Pareto front is (0.99998514396877, 0.615999986784s), whereas the top right point on the Pareto front is (0.99998514413651, 0.615999999856s). These DAs are then presented to a human designer for further processing. In our study we have selected the DA marked by the red circle in the Fig. 15.

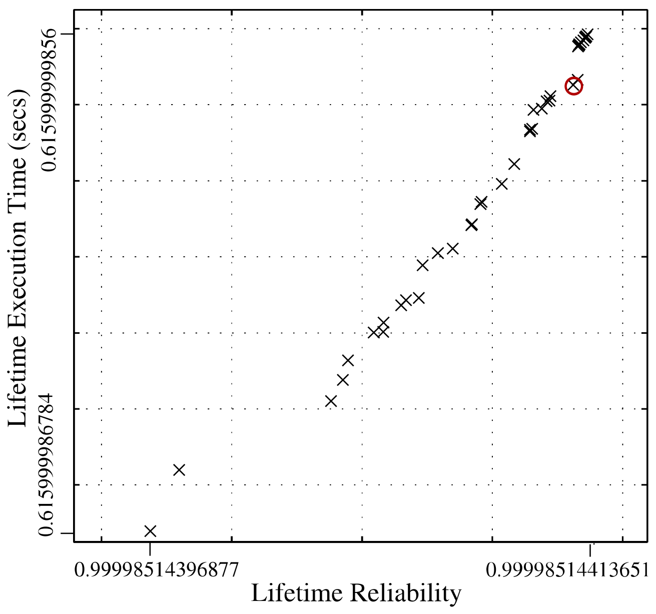


Fig. 15: **Pareto Optimal DAs**

Table 11: Mapping Edges Set for the selected DA

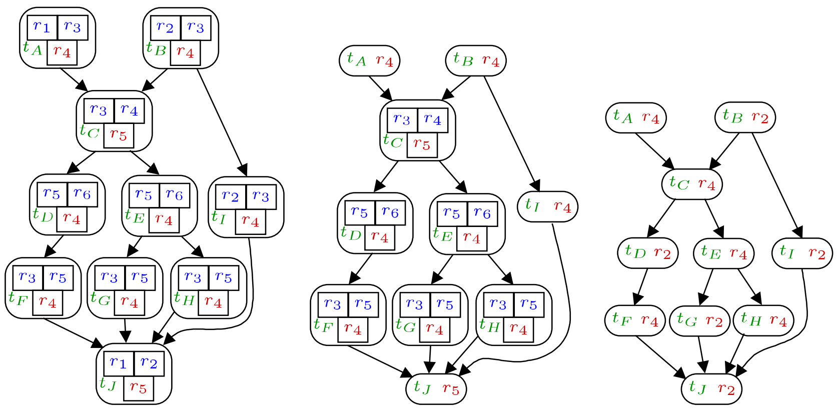
Tasks	Binded Resources
t_A	r_1, r_3, r_4
t_B	r_2, r_3, r_4, r_6
t_C	r_3, r_4, r_5
t_D	r_2, r_4, r_5, r_6
t_E	r_3, r_4, r_5, r_6
t_F	r_2, r_3, r_4, r_5
t_G	r_1, r_3, r_4, r_5
t_H	r_2, r_3, r_4, r_5
t_I	r_1, r_2, r_3, r_4
t_J	r_1, r_2, r_5, r_6

The lifetime reliability and execution time value for this DA is 0.9999851441327 and 0.615999998656 sec, respectively. This solution is chosen because it has the best execution time attribute amongst the other alternatives with almost similar lifetime reliability values. Table 11 represents the MES for the DA.

Table 12 shows all (28) of the permutations for the RSV-string with the corresponding non-failed system. The columns 2 and 4 in Table 12 show the number of system instances value for the corresponding RSV that consist of only SMR TIs and TMR TIs, respectively. Whereas column 3 represents the number of system instances that consists of mixture of SMR and TMR TIs. Columns 5 and 6 of this table depicts the reliability and execution time value of the “Best” SI. For each of the RSV these attributes are calculated using the algorithms in Sec. 6.

Table 12: RSVs and corresponding system instance sets and reliability and execution for Best SI

RSVs	SIS			Reliability	Exe-Time[s]
$r_1, r_2, r_3, r_4, r_5, r_6$	SMR	Mixed	TMR		
111111	2026	1668268	126	0.99998514415898	0.616
111110	1966	1334194	72	0.99998514415898	0.616
011111	653	95719	0	0.99862932434817	0.616
011110	640	81716	0	0.99862932434817	0.616
101111	709	59031	0	0.99878827590388	0.616
101110	649	31947	0	0.99838931939344	0.616
001111	229	2547	0	0.99648204038223	0.616
001110	216	1624	0	0.99608400507571	0.616
110111	756	12516	0	0.99703661850929	0.536
110110	696	6042	0	0.99647937092236	0.536
010111	203	977	0	0.99528936494313	0.48
010110	190	548	0	0.99473309390291	0.44
100111	336	1332	0	0.99520961140331	0.48
100110	276	168	0	0.99425608007021	0.424
000111	77	135	0	0.99481208435648	0.496
000110	64	0	0	0.99385893390244	0.432
111101	700	42756	0	0.9978313157768	0.576
111100	700	42756	0	0.9978313157768	0.576
011101	180	1980	0	0.99647841621256	0.576
011100	180	1980	0	0.99647841621256	0.576
101101	45	205	0	0.9965571638796	0.616
101100	45	205	0	0.9965571638796	0.616
110101	128	32	0	0.99520599186141	0.448
110100	128	32	0	0.99520599186141	0.448
010101	32	0	0	0.99385893390244	0.352
010100	32	0	0	0.99385893390244	0.352
100101	4	0	0	0.99385893390244	0.44
100100	4	0	0	0.99385893390244	0.44



(a) RSV 111111 (b) RSV 001111 (c) RSV 100100

Fig. 16: “Best” system instances for selected RSVs

Fig. 16(a), 16(b), and 16(c) depicts the pictorial representation for the “Best” SI for RSVs 111111, 001111, and 100100, respectively. The reliability and execution time attributes for these system instances are given in the corresponding row of Table 12.

We have repeated the experiment number of times, presented in this section on the system specification graphs adapted from numerous research publications that are consider as benchmarks. It was found that model constructed to evaluate lifetime reliability and execution time (i.e., SEDD) is far superior to others presented in literature [25–27] as it requires 4 times less number of nodes and the construction time is 3 times faster. Similarly, the execution of our genetic algorithm for various SSGs requires few minutes whereas the execution time for other genetic algorithm considering reliability and execution time is in hours.

8 Conclusion

In this work, we have presented a nested genetic algorithm to explore the design alternatives with best reliability and execution time values. First, we have introduced a novel reliability model that considers both hard and soft errors occurring in the digital resources (which are assumed to be fail non-silent) of the system is presented. Based on the reliability model and the accompanying execution time models a novel framework for system design is proposed and explained.

System error decision diagram which is a novel data structure, was introduced as an efficient means to calculate the lifetime system reliability and execution time. In addition, the process of its construction was described. This efficiency stems from the two aspects: a) the SEDD has fewer nodes than traditional BDDs, b) the algorithm for reliability evaluation demands fewer computations than comparable algorithms working on BDDs.

A novel design space exploration algorithm that consider the reliability and the execution time is presented. These algorithms take the system specification graph and system error decision diagrams as input and generate the required set of Pareto design alternatives. This provides the system designer a broader spectrum of implementation variants to select the "best" system implementation for the given requirements. The individual of each algorithm contains a mapping edges set. The genetic operators were constructed that operate on the TIDD, a novel data structure introduced in this work. For these algorithms, unlike the ones presented in the published research, a new system error decision diagram is not needed to be constructed for the evaluation of each individual of the genetic algorithm. The score values are assigned to individuals of the algorithm in a novel and unique manner. The Pareto design alternative set is updated for every generation of the genetic algorithms. For every RSV of each design alternative constructed by the genetic algorithm, a SI that has best reliability and execution time value is selected. The genetic algorithm for exploring the SI with minimum execution time is presented. The algorithm takes the ZBDD representing the set of highest reliability system instances as input.

References

1. G.E. Moore, Electronics **38**(8), 114 (1965). URL `\url{http://dx.doi.org/10.1109/JPROC.1998.658762}`
2. S. Feng, S. Gupta, A. Ansari, S.A. Mahlke, in *ACM Architectural Support for Programming Languages and Operating Systems* (2010), pp. 385–396
3. J. Srinivasan, S.V. Adve, P. Bose, J.A. Rivers, in *IEEE/IFIP Dependable Systems and Networks* (IEEE Computer Society, 2004), pp. 177–186. URL `\url{http://dl.acm.org/citation.cfm?id=1009382.1009733}`
4. J. Mulroy. 1000-core processor eats quad-core cpus for lunch. `http://www.pcworld.com/article/215113/1000core\_processor\_eats\_quadcore\_cpus\_for\_lunch.html` (2011)
5. A. Israr, Reliability aware high-level embedded system design in presence of hard and soft errors. Ph.D. thesis, Technische Universität Darmstadt (2012)
6. L. Huang, Q. Xu, in *ACM/IEEE Design, Automation, and Test in Europe* (2010), pp. 51–56
7. L. Huang, F. Yuan, Q. Xu, in *ACM/IEEE Design, Automation and Test in Europe* (2009), pp. 51–56
8. E. Karl, D. Blaauw, D. Sylvester, T.N. Mudge, in *ACM/IEEE Design Automation Conference* (2006), pp. 1057–1060
9. C. Zhu, Z.P. Gu, R.P. Dick, L. Shang, in *ACM/IEEE Intl. Conference on Hardware/Software Codesign and System Synthesis* (2007), pp. 239–244
10. J. Srinivasan, S.V. Adve, P. B., J.A. Rivers, in *ACM/IEEE Intl. Symp. on Computer Architecture* (2005), pp. 520–531
11. C. Lee, H. Kim, H.W. Park, S. Kim, H. Oh, S. Ha, in *ACM/IEEE Intl. Conference on Hardware/Software Codesign and System Synthesis* (2010), pp. 307–316
12. B.H. Meyer, A.S. Hartman, D.E. Thomas, in *ACM/IEEE Design, Automation, and Test in Europe* (2010), pp. 1596–1601
13. C. Bolchini, L. Pomante, F. Salice, D. Sciuto, J. Electron. Test. **18**(3), 351 (2002). DOI \url{http://dx.doi.org/10.1023/A:1015047524985}

14. C. Bolchini, L. Pomante, F. Salice, D. Sciuto, in *IEEE Intl. On-Line Testing Workshop* (IEEE Computer Society, 2002), pp. 32–36
15. Y. Xie, L. Li, M.T. Kandemir, N. Vijaykrishnan, M.J. Irwin, VLSI Signal Processing **49**(1), 87 (2007)
16. Y. Xie, L. Li, M.T. Kandemir, N. Vijaykrishnan, M.J. Irwin, in *IEEE Application-Specific Systems, Architectures, and Processors* (2004), pp. 41–50
17. V. Izosimov, P. Pop, P. Eles, Z. Peng, in *ACM/IEEE Design, Automation, and Test In Europe* (2006), pp. 706–711
18. V. Izosimov, P. Pop, P. Eles, Z. Peng, in *ACM/IEEE Design, Automation, and Test In Europe* (2008), pp. 915–920
19. V. Izosimov, I. Polian, P. Pop, P. Eles, Z. Peng, in *ACM/IEEE Design, Automation, and Test in Europe* (2009), pp. 682–687
20. V. Izosimov, P. Pop, P. Eles, Z. Peng, in *ACM/IEEE Design, Automation and Test in Europe* (2005), pp. 864–869
21. J. Conner, Y. Xie, M.T. Kandemir, R.P. Dick, G.M. Link, in *ACM/IEEE Conference on Asia South Pacific Design Automation* (2005), pp. 709–712
22. C. Calvert, G. Hamza-Lup, A. Agarwal, B. Alhalabi, in *IEEE Intl. Systems Conference* (2011), pp. 261–266
23. J. Srinivasan, S.V. Adve, P. Bose, J.A. Rivers, IEEE Micro **25**(3), 70 (2005)
24. K. Constantinides, S. Plaza, J.A. Blome, V. Bertacco, S.A. Mahlke, T.M. Austin, B. Zhang, M. Orshansky, ACM Transactions on Architecture and Code Optimization **4**(1) (2007)
25. K. Grüttner, A. Herrholz, U. Kuhne, D. Grosse, A. Rettberg, W. Nebel, R. Drechsler, in *IEEE Intl. Symp. on Object/Component/Service-Oriented Real-Time Distributed Computing Workshops* (2011), pp. 181–188
26. A. Jhumka, S. Klaus, S. Huss, IEEE/ACM Design, Automation, and Test in Europe pp. 372–377 (2005)
27. M. Glaß, M. Lukasiewycz, T. Streichert, C. Haubelt, J. Teich, in *ACM/IEEE Design, Automation, and Test in Europe* (IEEE Computer Society, 2007), pp. 409–414
28. M. Glaß, M. Lukasiewycz, F. Reimann, C. Haubelt, J. Teich, in *ACM/IEEE Design, Automation, and Test in Europe* (IEEE Computer Society, 2008), pp. 158–163
29. M. Glaß, M. Lukasiewycz, C. Haubelt, J. Teich, in *ACM/IEEE Design Automation Conference* (2010), pp. 234–239
30. A. Gerasoulis, T. Yang, J. Parallel and Distributed Computing **16**(4), 276 (1992)
31. A.A. Khan, C.L. McCreary, M.S. Jones, in *IEEE Intl. Conference on Parallel Processing* (1994), pp. 243–250
32. B. Shirazi, M. Wang, G. Pathak, J. Parallel Distrib. Comput. **10**, 222 (1990). DOI 10.1016/0743-7315(90)90014-G. URL `\url{http://portal.acm.org/citation.cfm?id=89227.89230}`
33. Y.K. Kwok, I. Ahmad, in *IEEE Intl. Parallel Processing Symp.* (IEEE Computer Society, 1998), IPPS '98, pp. 531–. URL `\url{http://portal.acm.org/citation.cfm?id=876880.879554}`
34. A.R. Jensen, L.B. Lauritzen, O. Laursen. Optimal task graph scheduling with binary decision diagrams (2004)
35. E.L. Lawler, in *Algorithmic Aspects of Combinatorics*, *Annals of Discrete Mathematics*, vol. 2, ed. by P.H. B. Alspach, D. Miller (Elsevier, 1978), pp. 75–90. DOI DOI:10.1016/S0167-5060(08)70323-6. URL `\url{http://www.sciencedirect.com/science/article/B8G56-4SD21YF-9/2/134232e3507895b39aa53e66417d4569}`
36. H. Topcuoglu, S. Hariri, M.Y. Wu, in *IEEE Heterogeneous Computing Workshop* (IEEE Computer Society, 1999), HCW '99, pp. 3–. URL `http://portal.acm.org/citation.cfm?id=795690.797895`
37. H. Topcuoglu, M.Y. Wu, IEEE Trans. on Parallel and Distributed Systems **13**(2), 260 (2002)
38. Y.K. Kwok, I. Ahmad, IEEE Trans. on Parallel and Distributed Systems **7**(5), 506 (1996)
39. Y. Fei, D. Xiaoli, J. Changjun, D. Rong, J. of Algorithms & Computational Technology **3**(2), 247 (2009)
40. M. Maheswaran, H. Siegel, H.J. S., in *IEEE Heterogeneous Computing Workshop* (Society Press, 1998), pp. 57–69
41. A. Girault, E. Saule, D. Trystram, J. Parallel Distrib. Comput. **69**(3), 326 (2009)
42. S. Minato, in *ACM/IEEE Design Automation Conference* (1993), pp. 272–277
43. R.E. Bryant, IEEE Trans. on Computers **C-35**(8), 677 (1986)
44. M. Wall. Galib, a c++ library of genetic algorithm components. `http://lancet.mit.edu/ga/` (2008)
45. Z. Vintr, M. Vintr, J. Malach, in *IEEE Intl. Carnahan Conference on Security Technology* (2011), pp. 1–5
46. A. Rauzy, Reliability Eng. and System Safety **40**, 203 (1993)
47. Traveling sale's man problem. `http://en.wikipedia.org/wiki/Travelling\_salesman\_problem`
48. P. Pop, Analysis and synthesis of communication-intensive heterogeneous real-time systems. Ph.D. thesis, Institue of Technology Linköping University (2003)

Your PDF file "MainLatexfile.pdf" cannot be opened and processed. Please see the common list of problems, and suggested resolutions below.

Reason:

Other Common Problems When Creating a PDF from a PDF file

You will need to convert your PDF file to another format or fix the current PDF file, then re-submit it.