# Evolutionary Algorithm for Scheduling Real-Time Applications in System of Systems

DISSERTATION
zur Erlangung des Grades eines Doktors
der Ingenieurwissenschaften

vorgelegt von
M.Sc. Setareh Majidi

eingereicht bei der Naturwissenschaftlich-Technischen Fakultät
der Universität Siegen
Siegen 2022

Betreuer und erster Gutachter
Prof. Dr.-Ing. Roman Obermaisser
Universität Siegen

Zweiter Gutachter
Dr. Raimund Kirner
University of Hertfordshire

Vorsitzender der Promotionskommission
Prof. Dr. Günter Schröder
Universität Siegen

Tag der mündlichen Prüfung
20.Mai.2022

Gedruckt auf alterungsbeständigem holz- und säurefreiem Papier.

# Abstract

In recent years, systems engineering and management have evolved from developing distributed systems to the integration of complex adaptive systems and the advent of Systems-of-Systems (SoS). SoS emerge from the collaboration of multiple systems with operational and managerial independency in order to accomplish a higher goal. SoS have been successfully deployed in different domains such as enterprise systems and smart cities. However, there is a critical challenge that must be tackled in order to adopt SoS in safety-relevant embedded applications: reliability and real-time capability are today not addressed in SoS. An open research challenge is the development of a distributed embedded system architecture for constantly evolving and dynamic SoS with support for verifiable real-time and reliability properties. The system architecture needs to support reliable closed loop control with stringent real-time requirements for applications.

Most of the existing scheduling solutions are developed for monolithic systems or complex systems with centralized authorities, which may violate the restrictions of SoS and not be able to satisfy its requirements. In this thesis, we develop an efficient heuristic approach for scheduling SoS applications with real-time and fault-tolerance requirements. In order to respect the SoS architectural restrictions, we model the scheduling decisions at two levels using a Genetic Algorithm (GA) optimizer as a solver, which iteratively interact to reach a feasible and efficient schedule for the SoS. The computational results show improvement in the average transmission makespan of SoS applications compared to the state-of-the-art scheduling solutions up to 31 percent in different scale scenarios. This work also investigates the capability of our scheduling approach in computing time-triggered schedules for a sequence of incrementally added SoS applications in a real-time SoS network. In this regard, a heuristic approach is developed at both scheduling levels to improve the schedulability of our algorithm by efficiently sparing free time slots on resources for the upcoming applications. Testing the schedulability and timeliness of the new incremental scheduler on a set of applications shows improvements in schedulability of up to 50 percent. Furthermore, we design a fault-tolerant scheduling approach for real-time SoS applications to tolerate permanent faults. Accordingly, fault-tolerance techniques such as re-execution and replication are integrated into our two-level GA scheduling algorithm to enhance the reliability of the system in combination with satisfying deadline constraints. The reliability is improved on average by 15 percent compared to the non fault-tolerant scheduler in different scenarios.

# Kurzfassung

In den letzten Jahren haben sich Systemtechnik und -management von der Entwicklung verteilter Systeme hin zur Integration komplexer adaptiver Systeme und zum Aufkommen von Systems-of-Systems (SoS) entwickelt. SoS entstehen durch die Zusammenarbeit mehrerer Systeme, die betrieblich und verwaltungstechnisch unabhängig sind, um ein höheres Ziel zu erreichen. SoS sind in verschiedenen Bereichen wie Unternehmenssystemen und intelligenten Städten erfolgreich eingesetzt worden. Es gibt jedoch eine kritische Herausforderung, die angegangen werden muss, um SoS in sicherheitsrelevanten eingebetteten Anwendungen einzusetzen: Zuverlässigkeit und Echtzeitfähigkeit werden bisher in SoS nicht berücksichtigt. Eine offene Forschungsherausforderung ist die Entwicklung einer verteilten eingebetteten Systemarchitektur für sich ständig weiterentwickelnde und dynamische SoS mit Unterstützung für überprüfbare Echtzeit- und Zuverlässigkeitseigenschaften. Die Systemarchitektur muss eine zuverlässige Steuerung im geschlossenen Regelkreis mit strengen Echtzeitanforderungen für Anwendungen unterstützen.

Die meisten der existierenden Scheduling-Lösungen wurden für monolithische Systeme oder komplexe Systeme mit zentralisierten Instanzen entwickelt, die möglicherweise die Einschränkungen des SoS verletzen und nicht in der Lage sind, dessen Anforderungen zu erfüllen. In dieser Arbeit entwickeln wir einen effizienten heuristischen Ansatz für die Planung von SoS-Anwendungen mit Echtzeit- und Fehlertoleranzanforderungen. Um die architektonischen Einschränkungen von SoS zu berücksichtigen, modellieren wir die Planungsentscheidungen auf zwei Ebenen und verwenden einen genetischen Algorithmus (GA) als Optimierer, der iterativ interagiert, um einen machbaren und effizienten Plan für den SoS zu erreichen. Die Berechnungsergebnisse zeigen eine Verbesserung der Übertragungszeit von SoS-Anwendungen im Vergleich zu anderen Scheduling-Lösungen um bis zu 31 Prozent in verschiedenen Szenarien.

In dieser Arbeit wird auch die Fähigkeit unseres Scheduling-Ansatzes untersucht, zeitgesteuerte Schedules für eine Sequenz von inkrementell hinzugefügten SoS-Anwendungen in einem Echtzeit-SoS-Netzwerk zu berechnen. In diesem Zusammenhang wird ein heuristischer Ansatz auf beiden Planungsebenen entwickelt, um die Planbarkeit unseres Algorithmus zu verbessern, indem freie Zeitfenster der Ressourcen für die kommenden Anwendungen effizient genutzt werden. Die Prüfung der Planbarkeit und Aktualität des neuen inkrementellen Schedulers an einer Reihe von Anwendungen zeigt eine Verbesserung der Planbarkeit um bis zu 50 Prozent. Darüber hinaus entwerfen wir einen fehlertoleran-

ten Scheduling-Ansatz für Echtzeitanwendungen, der permanente Fehler toleriert. Dementsprechend werden Fehlertoleranztechniken wie Re-Execution und Replikation in unseren zweistufigen GA-Scheduling-Algorithmus integriert, um die Zuverlässigkeit des Systems in Kombination mit der Einhaltung von Zeitvorgaben zu verbessern. Die Zuverlässigkeit wird im Vergleich zu einem nicht fehlertoleranten Planer in verschiedenen Szenarien um durchschnittlich 15 Prozent verbessert.

# Acknowledgments

This thesis is the result of four years of research at the university of Siegen. It would not have been possible without the support of many people.

I would like to express my deepest gratitude to my supervisor Professor Obermaisser for his encouragement, guidance and support during this time. I appreciate his continuous interest in my work. I would also like to thank Professor Kirner for taking the time to act as a reviewer for this thesis.

I would also like to thank my colleagues from the Embedded Systems group for all the pleasant discussions. You made the work an enjoyable one.

Finally, I would like to thank my family for their endless love and support through my life. Special thanks to my fiancé for always being there for me.

# Contents

# List of Figures

# List of Tables

# List of Abbreviations

| | |
|---|---|
| CS | Constituent Systems |
| CSM | Constituent System Manager |
| DAG | Directed Acyclic Graph |
| DM | Deadline Monotonic |
| EDF | Earliest Deadline First |
| GA | Genetic Algorithm |
| GLS | Greedy Local Search |
| GPS | Global Positioning System |
| HLF | Highest Level First |
| ILP | Integer Linear Programming |
| LSTF | Least Space-Time First |
| MBT | Maximum Blocking Time |
| MILP | Mixed Integer Linear Programming |
| ND | Network Domains |
| QoS | Quality of Service |
| SNAP | Stanford Network Analysis Package |
| SoS | Systems-of-Systems |
| SoSE | Systems-of-Systems Engineering |
| TB | Time Budget |
| TI | Time Interval |
| TSN | Time-Sensitive Networking |
| TTE | Time-Triggered Ethernet |
| TTP | Time-Triggrered Protocol |
| TTSoS | Time-Triggered Systems-of-Systems |
| WCET | Worst Case Execution Time |

# 1 Introduction

In recent years, the functionality of distributed systems with strict timing requirements has increased significantly. Distributed real-time systems consist of a set of interconnected computers with a real-time network and belong to the most important applications of computers, in terms of both commercial and social impact. In a real-time system, the correctness of the system depends on achieving deadlines and completing a process at the application-defined time as well as the logic of results [1]. In other words, real-time system requirements include not only the functional requirements such as data collection, digital or interaction control, but also the stringent temporal demands. The examples of real-time system applications can range from simple controllers in embedded systems to complex time-critical distributed systems such as avionics.

Generally, the System-of-Systems (SoS) term can refer to the collaboration of a set of independent systems, which are temporarily networked to provide novel services beyond the capabilities of each system alone. SoS can be established in different fields. The socio-technical elements of an SoS are called constituent systems. The constituent systems of an SoS can be technical, human or organizational elements. Distributed computing SoS architectures consist of self-contained cyber-physical systems which are normally provided by different organizations and with different implementation technologies and also human that use the system for personal purposes. The common SoS objectives are achieved by realizing emergent services by the interaction of constituent systems [2].

The SoS application domain is broad and still expanding. For example, it can be defined in any business enterprises by integrating a set of back office systems or customer-facing systems such as inventory management system, billing system and customer help center. Another example is the healthcare SoS, which integrates different patient care systems, such as laboratory system, pharmacy system, telemedicine system and patient management systems. In the safety-relevant application domains, SoS structures tend to be relatively static, which means their constituent systems change infrequently. Examples of SoS with real-time and reliability requirements are medical device systems, electric energy systems and defense systems.

There are five key characteristics for the distinction of SoSs from traditional complex systems: (1) independence of constituent in the operational mode, (2) independent management systems of entities, (3) geographical distribution, (4) complex emergence, (5) and evolutionary development process [3]. Among theses characteristics, the primary and sec-

ondary are the most important ones for applying the SoS term. The SoS domain is facing many unclear and ambiguous foundation concepts in the areas of dependencies, capabilities etc. However, there are agreed assumptions in Systems of Systems Engineering (SoSE) to be used in design and development of SoS [4]:

- Constituent systems are independent and operable entities, each can be also considered as a complex system.
- The collaboration among the constituent systems results in achieving higher levels of performance and purposes, which are beyond the capabilities of each individual. This emergence results in high degrees of complexity and uncertainty in SoS application environments.
- The boundaries among the constituent systems hide the internals of their implementations from each other.
- Data and information exchange is possible through different interfaces of constituent systems based on their amount of collaboration and integration into an SoS.
- The dynamic interaction of constituent systems with each other and their environment over time affects the integration of the constituent systems into an SoS.

Considering the degree of independence of constituent systems, four distinct categories are identified for the SoSs [5]:

1. Directed SoS owns and centrally manages all its constituent systems, e.g., the control systems in an automobile owned by a car company.
2. Acknowledged SoS refers to cooperative agreements between the owners of constituent systems to an aligned purpose.
3. Collaborative SoS is composed of independent constituent systems, which voluntarily interact to achieve a beneficial goal.
4. Virtual SoS is formed by constituent systems from different organizations without central alignment.

This taxonomy is considered as a framework for better understanding of the SoS objectives and the relationship between its constitutes. The level of collaboration and access to information varies in different SoS archetypes. For example in a collaborative SoS, we face a complex system with no central control and limited access to the operational information of its constitutes. As Figure 1.1 shows, there is limited control and unity of command in the collaborative and acknowledged forms comparing to the directed form of SoS. In real

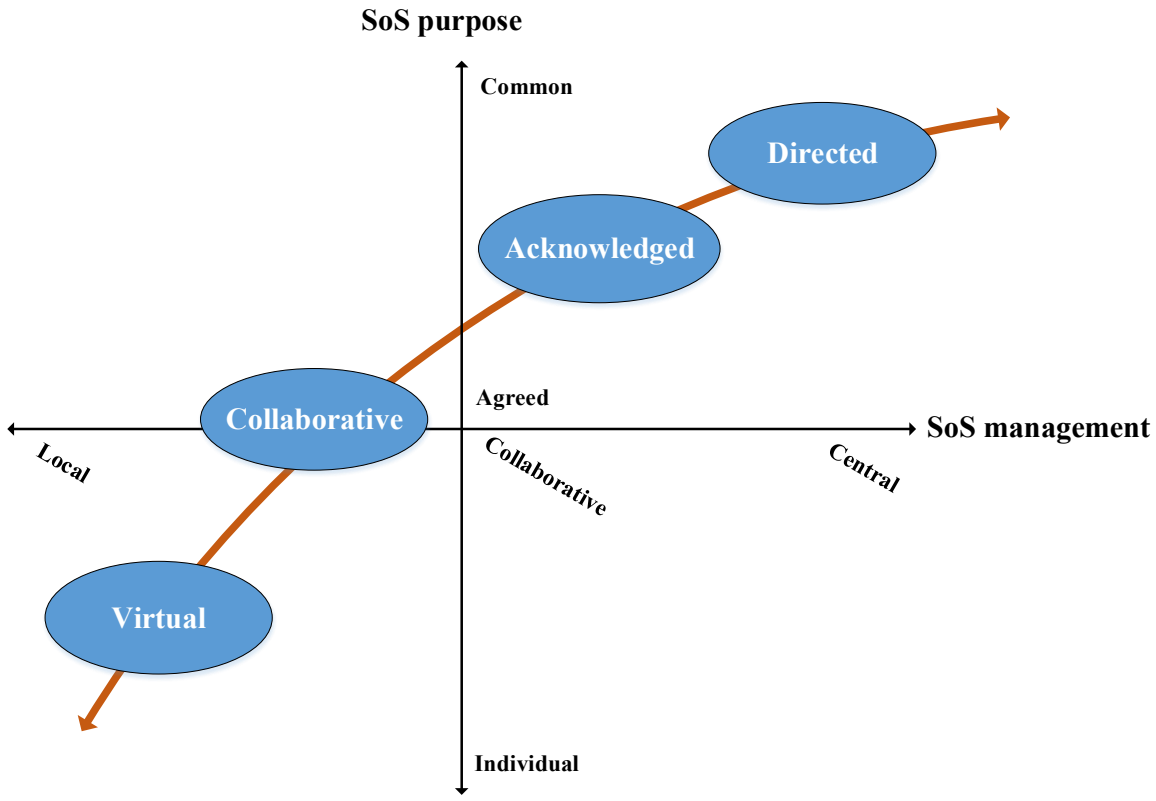


Figure 1.1: SoS archetypes

cases, an SoS may change its type over time or reflect the combination of theses types or become even unrecognized because of the new levels of interconnectivity.

In this study, we deal with the collaborative SoS architecture in the embedded application domain. In our SoS, all the constitutes agree to interact in order to fulfill the central purposes and collectively decide how to provide the SoS services. To ensure the reliability and real-time requirements in our SoS, there must be the agreement on the same communication standards and protocols. Moreover, an external reference time is required to synchronize all the constituent systems' local clocks, e.g., by means of satellite-based navigation systems such as GPS.

## 1.1 Context and Motivation

Facing today's interconnected world, SoS can be found in numerous application domains [6]. First, it was identified in the defense sector but now most of today's infrastructures, such as transportation networks, power supply, health care and many embedded systems depend on the functionality of SoS [7]. For safety-relevant applications, the SoS perfor-

mance is determined by addressing dependability and real-time requirements.

There are many examples of SoS applications that rely on real-time communication. Considering an emergency center for real-time patient monitoring [8], the successful delivery of a service (i.e., the treatment of a patient) relies on the cooperation and sharing the common resources between a variety of self-contained systems with possible conflicting requirements, including ambulance center and hospital management, as well as the direct interaction of human such as the patient, his/her family and the team of doctors. Further examples of SoS applications that rely on real-time communication are traffic control as well as command and control systems.

In this study, it is assumed that a time-triggered protocol is an acceptable transport service for all the constituent systems to provide time-sensitive SoS applications. We develop an efficient and reliable method for mapping and scheduling the time-triggered actions from the complex services inside and between the constituent systems of a time-sensitive SoS with respect to the existing challenges of exchanging information and control in such systems.

The complexity of communication control in SoSs, and strict requirements on safety, fault-tolerance and reliability of time-sensitive SoS applications, can be effectively addressed with time-triggered control and corresponding scheduling frameworks [9]. A time-triggered schedule includes predefined time instants derived from the progression of the global time related to the execution of jobs and transmission of periodic messages. Time-triggered communication protocols support higher dependability and determinism during the regular operation, as well as lower jitter based on a global time base [10]. Furthermore, there is a wide availability of time-triggered technologies for an SoS such as time-triggered networks (e.g., Time-Sensitive Networking (TSN) [11], Time-Triggered Ethernet (TTE) [12]) and operating systems (e.g., PikeOS [13]) inside each constituent system as well as time-triggered networks for the network domains between the constituent systems (e.g., DetNet [14]).

Since the high predictability of time-triggered operations can make the timing behavior analysis of an SoS more straight-forward, and due to the possibility of deploying time-triggered technologies for SoSs, we introduce Time-Triggered SoS (TTSoS) to address an SoS with time-triggered architectures and scheduled communication networks. In a TTSoS, each constituent system is a networked embedded system consisting of end-systems that are interconnected by time-triggered communication networks with different protocols and topologies. The interconnection of constituent systems occurs using a backbone communication infrastructure consisting of multiple network domains. TTSoSs guarantee the strict temporal constraints by determining a time-triggered schedule. Additionally, TTSoSs promise to satisfy the dependability requirements of time-critical applications, such as medical systems, smart manufacturing and defense systems, by enabling fault-tolerant techniques, e.g., the ability of performing redundant computations of multiple

services and voting on the outputs. Consequently, the resulting properties make the time-triggered architectures well-suited for the SoSs.

TTSoS networks may encounter different applications which are introduced and/or removed over time, while sharing the underlying resources of the constituent systems. Regarding the limited accessibility to information from the internal scheduling process within the constituent systems, the timing analysis process in a TTSoS is restricted and challenging [15]. Despite the wide research on time-triggered scheduling frameworks, they were mostly developed for monolithic systems and do not support the requirements and constraints of TTSoSs. The existing incremental scheduling methods in time-triggered SoS networks are focused on optimizing each application individually, i.e., not considering the requirements of future applications. The lack of an appropriate customized scheduling strategy will cause the shortage of resources which could result in catastrophic outcomes in safety-critical domains such as health care systems.

The main optimization challenges that we face in scheduling TTSoS services are the temporal and spatial coordination of the autonomous constituent systems including the service allocation and the shared resource access, while satisfying the temporal constraints of real-time applications. Moreover, there are possible conflicting goals between the SoS and its constituent systems besides the lack of central control and global knowledge w.r.t. internals of constituent systems. In scheduling a TTSoS application, each constituent system is only aware of its own resources and takes care of scheduling its operations hidden from other partners, which means there is no need to share information with any central authority about how to utilize resources. However, in case of releasing a sequence of applications, local schedulers should update their resource allocation approach accordingly. Therefore, the resource allocation mechanism should contemplate not only the timing constraints of the current SoS application but also the schedulability of applications introduced in the future.

## 1.2 Objectives and Contribution

The main objective of this thesis is to design a customized scheduling approach for SoS architectures, which supports temporal guarantees of operations. Addressing the scheduling problem in the SoS domain mentioned in the previous section, the main contributions of this thesis are as follows,

- Proposing a two-level iterative scheduling model which provides time-triggered schedules for TTSoS applications. This model ensures the timeliness of the operations as well as satisfying the special boundary conditions of SoS architectures, i.e., lack of centralized control and global knowledge.

- Developing two scheduling methods based on different heuristic frameworks. (1) two-level iterative Genetic Algorithm (GA) that determines near-optimal solutions but in a long run time. (2) two-level iterative greedy local search algorithm that deploys faster searching strategy but not ideal. We compare the schedulability and functionality of these scheduling methods on different generated scenarios of TTSoS.

- The integration of a new resource allocation and search evaluation approach in the scheduling model to cope with limited communication resources in case of expecting new SoS applications which are incrementally added to the network and required to be scheduled.

- The development of a fault-tolerant scheduling method with executing replicated services and considering redundant paths for exchanging the time-triggered messages within and between the constituent system to increase the reliability of our system.

This thesis investigates efficient scheduling methods for TTSoS. First, we consider a collaborative TTSoS with a high-level operational view to lead constituent systems to handle their operations and there is no restriction to access information about their operations. In our system, the information exchange happens at the high-level interaction between constituent systems managers and network management services. Accordingly, we distinguish two levels for our scheduling model, namely the CS level and the SoS level.

At the SoS level, a set of global schedules are generated and suggested to the CS level. Each global schedule determines the resource allocation (At this level means selecting suitable constituent systems as service providers and the shortest communication path for transporting time-triggered messages between these systems) as well as the temporal domain of services (start and finish instant of execution). Based on these global schedules, each constituent system determines a time-triggered schedule for the corresponding service and sends back a feedback to the SoS level. When these results are not acceptable for the SoS level, it will update its global schedules and continue to iterate this process until reaching feasible solutions.

The scheduling model includes two phases: (1) scheduling phase (2) execution phase. In the first phase, the admission of a new SoS application is performed and time-triggered schedule tables are generated. After the successful establishment, there is an execution phase of the SoS application. In other words, a new schedule is computed at run time whenever a new SoS application is established. Indeed, our scheduling approach is considered as a semi-static model. Moreover, it is assumed that a new SoS application joins after completing the scheduling process of the previous one.

Regarding our scheduling model, we propose a two-level iterative heuristic algorithm using a GA optimizer. The first level refers to the SoS level and starts with generating global schedules, which include random time slots for selecting suitable constituent systems

to provide the services from an SoS application. The second level scheduler is run after receiving information from its preceding level. According to the autonomy of constituent systems, they can run the local schedulers simultaneously. This parallel execution saves extra time required for solving the large-scale scheduling problems. After completion of local schedulers, the best solutions are sent back to be assessed by the SoS level. In case of facing an infeasible solution, the SoS level will update its initial schedules and send them again to the local schedulers. This procedure iterates until finding a feasible schedule, otherwise, it terminates after a specific number of iterations.

Moreover, we develop a two-level scheduling algorithm based on the Greedy Local Search heuristic (GLS) as a reference scheduling algorithm which employs a different search strategy to optimize the global and local schedules. We apply both GA and GLS schedulers to different examples in different scales to examine their schedulability and scheduling capabilities in terms of makespan.

Furthermore, we develop a new job allocation policy to efficiently utilize the limited resources of our constituent systems for a large number of applications arriving over time. Based on the current processor selection strategy of each local scheduler, the earliest feasible time slots on resources will be assigned to the jobs. The current resource allocation approach maps efficiently the available resources to the initial SoS applications but gradually with incrementally emerging new applications over time, it is not possible to find feasible schedules since the resources are fully occupied by earlier applications. To avoid the shortage of resources, we require a new allocation approach which considers more dispersed busy time slots on resources and prevents long blocking times.

Therefore, we propose a new evaluation measure called Maximum Blocking Time (MBT) and integrate it into our scheduling method. Moreover, we propose a new fitness function which resembles the trade-off between the MBT of resources and the makespan of each application. As a result, the new approach may generate non ideal solutions for individual applications from the makespan point of view, but there is a better chance of finding feasible schedules for later applications. We examine the functionality of our new incremental scheduler by computing the schedules of a set of generated real-time SoS applications arriving over time.

To address the reliability requirements of safety-critical applications and due to the ability of a time-triggered protocol to support redundant communication, we integrate our base two-level GA-based scheduling algorithm with fault-tolerance techniques. Accordingly, the scheduling approach at the first level sends suggested time windows and reliability preferences of providing services to candidate constituent systems. This level also selects a set of multiple services to perform redundant computations as well as replicating each time-triggered message to transmit within another path (i.e., the second shortest path between its sender and receiver constituent systems) through network domains. In the scheduling model for each service, we implement the redundant computations of jobs and

transmit replicated message paths inside the network of constituent systems. The scheduling problem at this level is expressed as a multi-objective optimization problem including maximizing reliability and minimizing makespan with respect to the timing constraints.

## 1.3 Thesis Structure

The rest of thesis is presented as follows:

Chapter 2 covers the basic terms and concepts in our work. It gives an overview of SoS and its characteristics. It provides a brief introduction of real-time scheduling and investigates different solution approaches such as GA. A review of definitions regarding faults and fault-tolerance techniques are also given in this chapter.

Chapter 3 analyzes the state of the art in scheduling algorithms for distributed systems. It shortly discusses the main SoS modeling frameworks and scheduling algorithms, and lastly illustrates the research gaps.

Chapter 4 describes the TTSoS architecture from the logical and physical points of view, and models the problem of scheduling time-triggered applications in the system with respect to its managerial and structural constraints.

Chapter 5 deals with our proposed methodology for computing time-triggered schedules in a TTSoS as well as the development of the incremental scheduling approach to cope with incrementally added TTSoS applications in the system.

Chapter 6 presents the proposed fault-tolerant scheduling algorithm for TTSoS to ensure the system reliability.

Chapter 7 discusses the improved schedulability and timelines of our heuristic scheduling algorithms by running different scenarios and comparing the results with the state-of-the-art approaches.

Chapter 8 concludes the thesis along with giving some suggestions for the future works.

# 2 Concepts and Terms

In this chapter, we review the main definitions and concepts that are used in this thesis. First, we start with the fundamental concept of time-triggered systems. Then, the definition of an SoS followed by highlighting its important characteristics is presented. The main modeling standards and architecture frameworks specific to this type of systems are briefly reviewed in this section. Next, scheduling in real-time systems is explained and a quick review on GA as a scheduling optimization heuristic is given. Finally, a general review about fault-hypotheses and fault-tolerance techniques are given.

## 2.1 Time-Triggered Systems

Time-triggered and event-triggered control are two paradigms for architecting a distributed real-time system. The difference is in the source of control signals, which makes time-triggered systems more predictable and dependable but inflexible [16]. In a time-triggered system, there is an instant according to a static schedule to trigger an action (i.e., injection of a message or the execution of a job) [17]. Generally, the communication structure for time-triggered networks is generated in advance and not modified during operation [10].

Many time-critical applications (e.g., aircraft systems) receive repeatedly data from the real world (via senors) and must provide a timely response (via an actuator) after processing. In distributed safety-relevant applications it is important to have a deterministic response. Predictability and highly deterministic behavior of time-triggered systems make them the preferred choice for such applications. The use of time-triggered architectures can have other benefits for the system such as better fault containment, reduced CPU and memory usage [18].

## 2.2 Systems of Systems (SoS)

Systems-of-Systems (SoS) are a class of systems that have unique characteristics, distinguishing them from classic complex systems. An SoS refers to large-scale distributed systems composed of various interconnected self-contained constitutes gathered to achieve higher goals. These constituent systems are separately acquired and continue to be managed as independent systems and can be provided by different organizations and with

different implementation technologies. SoS objectives are achieved by emerging services realized by the interaction of constituent systems. They can operate in a useful manner by using protocols and standards to enable interoperability. In a real-time SoS, each constituent system is a networked embedded system consisting of end-systems that are interconnected by real-time communication networks with different topologies to deliver the requested emerging real-time services. The interconnection of constituent systems occurs using a backbone communication infrastructure consisting of multiple network domains. The interaction between the constituent systems happens through their Constituent System Managers (CSM). The SoS has broad coverage of application domains, such as energy, telecommunications, health care, transportation, and military [3]. As an example of the healthcare SoS, we can point to the cooperation of different partners such as healthcare centers, laboratory systems, hospitals and emergency centers.

[19] SoS can be distinguished from conventional systems based on a number of unique characteristics, which are listed as follows,

- Autonomy: Each constituent system of an SoS can operate independently from the operational and managerial points of view.
- Connectivity: The level of connectivity between constituent systems is based on their needs and agreements.
- Emergence: In an SoS, the behaviors and capabilities of a system can be developed from interacting with other systems.
- Geographical distribution: The constituent systems of an SoS can be located in any locations.
- Evolutionary development process: Changing environment or introducing new technologies can lead to changes to existing operational strategies and capabilities of the constituent systems.

### 2.2.1 Model-based development standards

The unique characteristics of SoSs have lead the system engineering community to investigate new languages, models and frameworks to have better definitions of these systems [20]. Systems-of-Systems Engineering (SoSE) investigates new SoS capabilities by leveraging synergies of component systems and consists of best practices in design, development, testing, analysis and maintenance. In contrast to traditional system engineering, in which the system architecture remains relatively stable during the life cycle of the system, SoSE typically considers a service oriented architecture to dynamically reconfigure as needs change [21].

There are standard representation-oriented architecture frameworks for SoSs, e.g., DoDAF (Department of Defense Architecture Framework) [22], FEAF (Federal Enterprise AF), MODAF (The British Ministry of Defense Architecture Framework), UPDM (The Unified Profile for DoDAF/MODAF), as well as models for behavior, interfaces, requirements and performance of constituent systems, e.g., SysML, Modelica, MARTE.

## 2.3 Real-Time Scheduling

A real-time system must complete the execution of a set of jobs at periodic intervals within a specific time bound. The response time is a crucial criterion to evaluate the correctness of real-time applications. Schedulers count as one of the essential components in real-time systems and have a very important impact on the system performance [23]. Moreover, the lack of sufficient resources can also lead to failure in order to meet the timing constraints, which can bring catastrophic results in some cases, e.g., in safety-critical systems [24].

Scheduling is defined as the temporal and spatial allocation of shared resources to a set of jobs. There are constraints to solve a scheduling problem for time-critical systems.

1. Timing constraints include deadlines, and worst case execution times,
2. Precedence constraints determine the permitted temporal order for the execution of jobs,
3. Resource constraints determine how many jobs are allowed to be executed on a specific type of resource,
4. Per-job constraints are associated with a job, e.g., it may need to be executed on a specific resource.

The scheduling models for distributed systems can be categorized into two distinct groups: static and dynamic scheduling. Static scheduling is suitable for the problems with a fixed list of jobs within the expected completion time, where all information about these jobs is known in advance. In contrast, dynamic scheduling supports a continuous flow of incoming jobs and the changes of the workload. Making decisions about the next job to run and handling overload are key challenges in dynamic scheduling. The scheduling of jobs in safety-critical systems are typically constructed in a static manner to support the deterministic timing requirements.

Another classification of scheduling models is preemptive and non-preemptive scheduling. A preemptive scheduler can interrupt any running job in order to execute another job on the same resource, which may result in a reduced response time. Contrarily, when a job begins to execute in a non-preemptive scheduling process, it will not stop until it is done.

## 2.4 Scheduling Optimization Heuristic

A good scheduling algorithm should ensure that deadlines and requirements of the system are met, besides having high efficiency and scalability [25]. The scheduling optimization methods can be categorized into exact and heuristic. Exact algorithms such as MILP guarantee to result in an optimum solution, on the other hand, heuristic algorithms such as GA or neighborhood search find a near optimum solution in a shorter execution time. The selection between the exact and heuristic methods depends on the type and the size of the problem. Genetic Algorithm (GA) is one of the common heuristic approaches for solving NP-hard optimization problems such as traveling salesman problem or the scheduling problems. In the following section, we review the basic concepts of GA.

### 2.4.1 Genetic Algorithm (GA)

GA belongs to the group of population-based optimization algorithms inspired by biological evolution. In a GA, a population of feasible solutions is evolved to approach better solutions. Figure 2.1 shows a general scheme of this algorithm. It starts with initializing the population, then the selection of fitter solutions follows by fitness evaluation and reproduction using the crossover and mutation operators. The iteration continues until termination. This section explains different steps of a GA in detail.

There are generally nine steps in a GA:

1. Generate an initial population including $K$ individuals each with a predefined number of chromosomes.

2. Calculate a specific fitness function for the evaluation of the chromosomes.

3. Use a roulette wheel method to determine whether to perform crossover or mutation.

4. If the crossover method is selected, select randomly two chromosomes as parents and produce two offspring chromosomes.

5. In case of mutation, select randomly one chromosome and apply the operator.

6. Add new chromosomes at the end of the population list.

7. Return to step 3 until a fixed number of iterations is reached.

8. Order the population based on their fitness values. Select the first $K$ individuals (size of population) from the list and transfer them to the next generation.

9. Return to step 1 until the maximum number of iterations is reached.

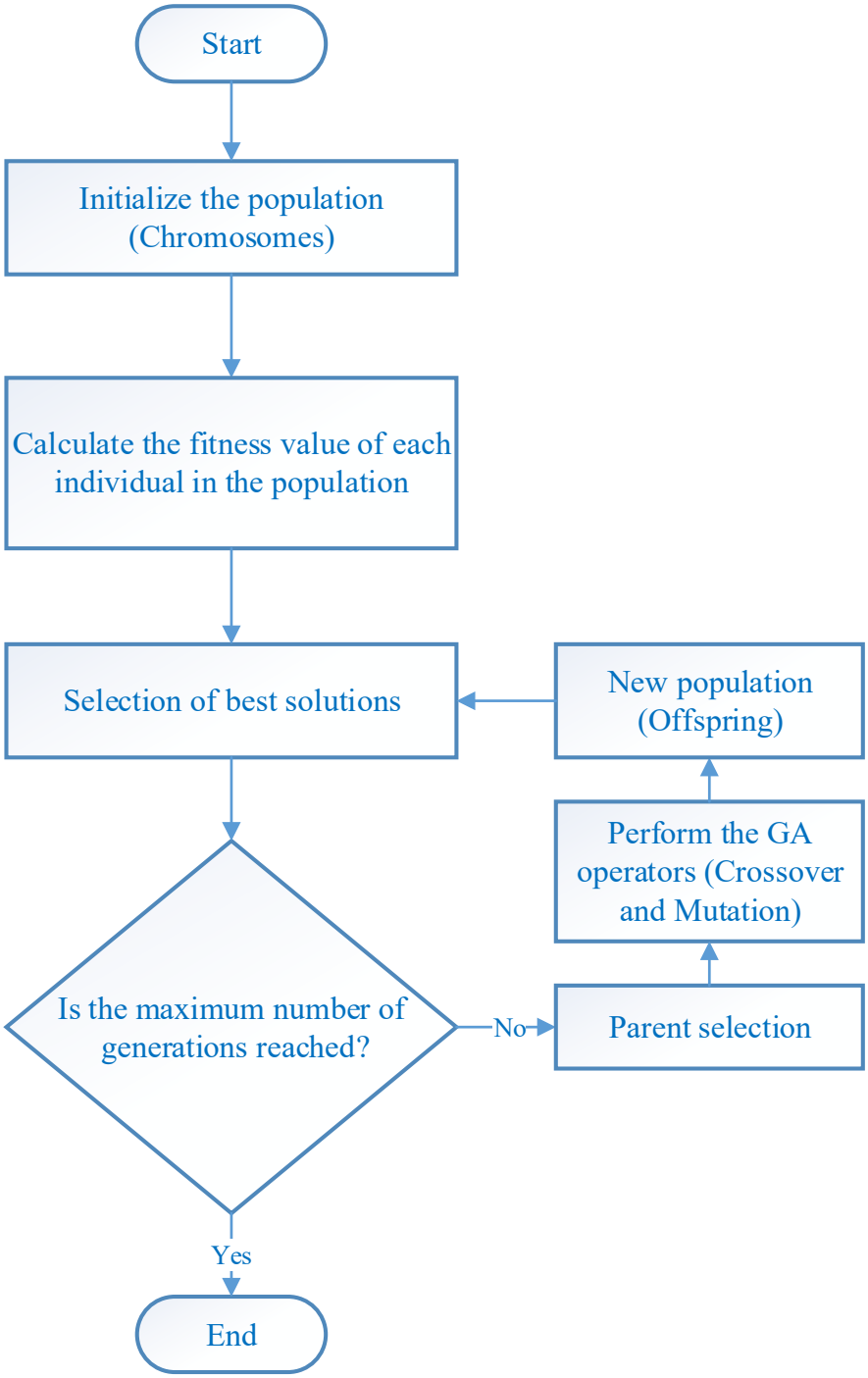


Figure 2.1: Flowchart of GA

### Initialization

The first step in GA is to generate an initial population, which is a collection of chromosomes. A chromosome refers to one possible solution and contains elements, which are known as genes. Depending on the nature of the problem, these genes can hold different values, e.g. binary, real, permutation or integer. The use of real-valued genes is more efficient than the binary type, as it can shorten the process of evaluation. In each iteration of GA, chromosomes will be evaluated by means of an appropriate fitness function.

### Crossover

Crossover is one of the most efficient operators of a GA for searching the solution space. It happens between two parents' chromosomes to exchange the genes and results in generating two offspring chromosomes. There are several strategies for this operator, such as one-point, two-point, k-point, and uniform crossover. For example, in a single-point crossover, an integer number $i$ is randomly selected between 1 and $n$ (the size of the chromosome), and it is considered as a "crossover point". At this point, the chromosome is split up into two parts and by exchanging the similar parts, two new chromosomes will be generated. Figure 2.2 shows the general mechanism of a single point crossover.

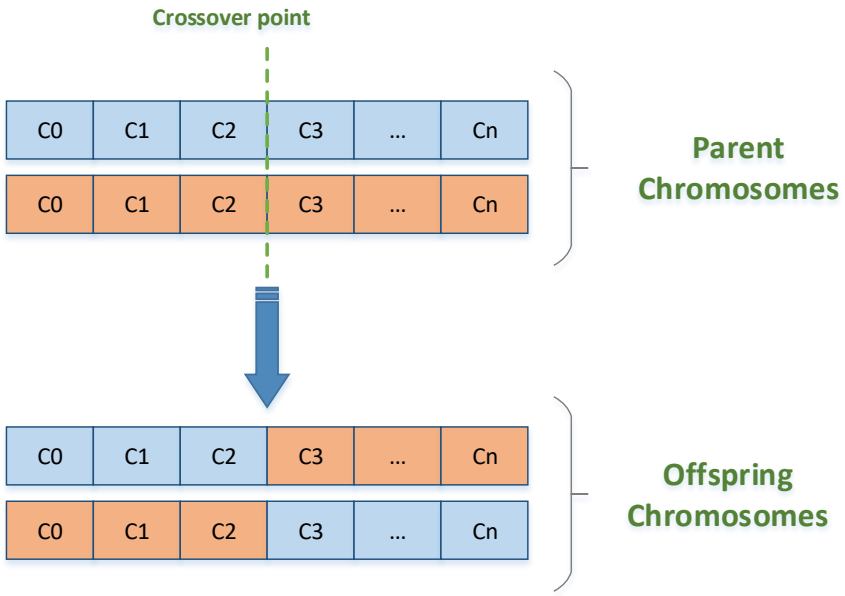


Figure 2.2: Single point crossover

### Mutation

The mutation operator is another selection mechanism in a GA to have a comprehensive search in the solution space. Figure 2.3 shows a simple example of a single cell mutation. The mutation operator is applied on one chromosome as a parent from the population. The single cell mutation operator randomly selects a cell to mutate and changes the value of the corresponding cell.

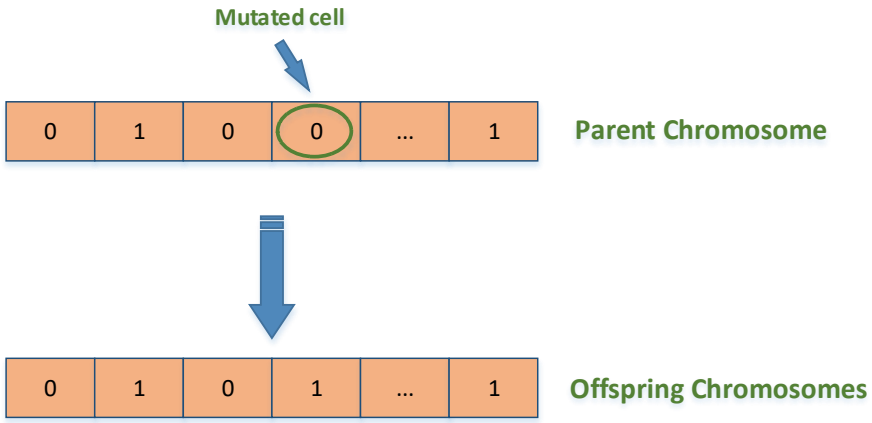


Figure 2.3: Single cell mutation

### Fitness Function

The GA process determines the weight of each chromosome by its fitness function, i.e., it shows how close each solution is to the optimum solution. These weights are sorted either in an ascending or a descending order based on the optimization problem. Thereafter, the chromosomes with better fitness values will be selected for the next generation to be evolved.

## 2.5 Faults and Fault-Tolerance in Distributed Systems

Distributed real-time systems are increasingly applied in safety-relevant sectors, such as health care, transportation and telecommunication, due to their capabilities. However, if they do not continue to function correctly in the presence of faults, they may have catastrophic consequences. Therefore, it is necessary to apply fault tolerant techniques specially in the scheduling models [27]. In this regard, some essential definitions and basic concepts related to this field are presented in the following sections.

### 2.5.1 Faults

Failure to meet deadlines in real-time systems can be caused by software or hardware faults. They are categorized in three types:

- Permanent faults: a permanent fault will remain over time and is repaired by replacing the faulty unit e.g., damaged end-systems, routers or links.

- Intermittent faults: this type of faults refers to the irregular malfunction of a device or system which occurs in intervals. For example, a connector problem with loose contacts can cause intermittent faults.

- Transient faults: a transient fault will disappear over time, e.g., electromagnetic interference.

### 2.5.2 Dependability

Dependability is one of the fundamental characteristics of a computing system besides functionality, usability and cost. It is defined as the ability of the system to operate accurately and trustworthy. There are four techniques to develop a dependable system including fault prevention, tolerance, removal and forecasting [32].

Fault prevention techniques are mainly employed in the design and manufacturing of a product, e.g. considering information hiding for software or radiation hardening for hardware. Fault tolerance techniques intend to keep the system operating correctly in spite of active faults. As an example of fault removal during the operational life of a system, we can mention corrective or preventive maintenance. Lastly, fault forecasting is based on qualitative or quantitative evaluation of the system behavior respecting the occurrence of faults [32].

### 2.5.3 Fault Tolerance Techniques

There are two ways to design a fault-tolerance system. (1) Hardware redundancy techniques rely on hardware replication to tolerate permanent faults. There are several hardware architecture solutions, such as MARS [28], TTA [29] and XBW [30]. Due to the limited resources and large hardware costs, hardware redundancy techniques are only applied for highly safety-critical systems such as a passenger aircraft. (2) Software redundancy techniques rely on duplicating messages or re-execution of jobs and are more commonly used to make a fault resistant system.

### 2.5.4 Reliability Measure

Reliability is related to the probability of providing correct services and meeting the deadlines of the system [33]. Probabilistic tools are prevalent to evaluate and measure system reliability. The most important parameter in the determining the reliability is the failure rate, i.e., the number of failures that happens to an individual unit per time [34].

The reliability of a system $R(t)$ is calculated as the probability of providing timely and correct services. $R(t)$ in terms of a failure-rate function with constant failure rate $\lambda$ is assumed as follows,

$$R(t) = e^{-\lambda t}$$

There are three types of systems depending on the system architecture: series, parallel and series/parallel systems. In a serial system, the reliability of the entire system $R_s(t)$ is computed by the following mathematical formula,

$$R_s(t) = \prod_{i=1}^{N} R_i(t)$$

Where $R_i(t)$ is a reliability of unit $i$ which can be either an end-system, a router or a link. For a parallel system the reliability of system $R_p(t)$ can be expressed by,

$$R_p(t) = 1 - \prod_{i=1}^{N} (1 - R_i(t))$$

However, there are some systems which may have a mixed structure of series and parallel units. In these systems, reliability cannot be calculated by the above equations. Therefore, the reliability of theses systems is calculated by expanding a module $i$ as follows,

$$R_{system} = R_i \cdot Prob\{System\ works|\ i\ is\ fault-free\}+$$
$$(1 - R_i) \cdot Prob\{System\ works|\ i\ is\ faulty\}$$

# 3 Related Work

SoSs can be established in a variety of safety-relevant domains, e.g., in medical centers, smart manufacturing, transportation networks and defense sectors. However, SoSs should be able to efficiently address the real-time and dependability requirements of time-critical applications, otherwise they may result in irrecoverable damages. To improve the reliability and timeliness, we introduce Time-Triggered SoS (TTSoS). TTSoS is constituted of a set of independent distributed systems which voluntarily interact to fulfill agreed purposes and are interconnected using a backbone communication infrastructure consisting of multiple network domains with different protocols and topologies. Each constituent distributed system consists of a set of end-systems that share a time-triggered communication network.

SoS research is still restricted by many unclear and ambiguous concepts. The literature in this domain is diverse, ranging from attempts to clarify the conceptual foundation of SoS to industrial views and solutions for managing the complexity of communication and data exchange between systems. This chapter reviews the scientific works in different aspects of the SoS domain particularly in in the area of scheduling algorithms and indicates the research gap.

## 3.1 Requirements and Research Challenges

SoS is a relatively new area, so there are still open research challenges in design, management and implementation. There are several notable studies which develop different modeling frameworks and supporting tools in the SoS domain, but not enough work focusing on scheduling problems for the SoS networks is available. Due to the specific characteristics such as evolution and emerging behavior found in an SoS, we deal with different challenges ranging from complex architectural design problems to technical decision making issues such as task allocation and scheduling.

Dealing with SoS scheduling problems, there are critical challenges including satisfying the temporal constraints of real-time applications and optimizing the system's overall goals considering the autonomy of CSs and their possible conflicting goals [35]. Furthermore, real world SoS networks encounter different applications which are introduced and/or removed over time, while sharing the underlying resources of the constituent systems. In scheduling an SoS application, each constituent system is only aware of its own resources and takes care of scheduling its operations independently, which means there is no need to share

information with any central authority about how to utilize resources. However, in case of releasing a sequence of applications, local schedulers should update their resource allocation accordingly when new applications arrive. To deal with scheduling time-triggered traffic in this situation, the resource allocation mechanism should contemplate not only the timing constraints of the current SoS application but also the schedulability of applications introduced in the future.

These are the main challenges that we face to optimize the incremental scheduling problem for real-time communication in our SoS network as well as improve the reliability of the system in case of faults.

1. Ability to cope with lack of central control and global knowledge w.r.t. internals of constituent systems,

2. Establishment of service contracts: the establishment of an SoS application at runtime after its arrival is implemented by providing a contract of involved constituent systems with relied upon services and temporal constraints,

3. Common resource reservations: allocation of the resources at the different constituent systems as well as in the interaction domain between constitutes,

4. Support for dynamic and incremental arrival of SoS applications,

5. Guaranteed timeliness: ensuring ability of meeting deadlines of an application after completing its establishment process within reasonable time (e.g. minutes)„

6. Fault-tolerance: tolerating failures that may happen in end-systems and communication links,

## 3.2 Scheduling Algorithms for Distributed Systems

Deriving an efficient schedule in real-time distributed systems is a well researched optimization problem.The scheduling process deals with the temporal and spatial allocation of computation and communication activities to the resources. An optimal solution for the scheduling problem in time-critical distributed systems is challenging and can be computationally infeasible for large problem sizes. The scheduling problem is NP-hard and time required for computing optimal solutions increases exponentially with the system size. The scheduling process and schedulability analysis are essential in distributed embedded systems and have been intensively addressed in several works.

Zhao et al. proposed an efficient optimization algorithm to compute the priority of tasks and their maximum unscheduled virtual deadlines minimizing the response time of

real-time systems [36]. Pahlevan et al. developed a GA-based algorithm to schedule multicast time-triggered flows in TSN, which combines the routing and scheduling constraints [37]. Bingqian et al. developed a hybrid genetic algorithm for the scheduling problem in TTEthernet networks and improved the performance of scheduled time-triggered traffic to be more compatible with further changes in the network [38]. Schweissguth et al. presented an Integer Linear Programming (ILP) formulation to optimize the joint routing and scheduling problem in TTE networks which supports application-specific cycle times. They compared the results of their scheduling model with the solutions from models using a fixed table of shortest paths [39]. Pop et al. dealt with a heuristic approach to process the scheduling problem of safety-critical applications based on the time-triggered protocol in distributed embedded systems [40]. They examined the proposed algorithm on a large number of generated experiments and a real-life example. Kuchcinski formulated the process scheduling problem in embedded systems based on the timing constraints solving techniques [41], while the solution proposed in [42] is based on MILP.

Liestman and Campbell proposed two different scheduling algorithms for their software system to satisfy the real-time and reliability requirements. Although the primary algorithm provides a good quality schedule, its timely completion is not guaranteed. When the primary schedule fails to complete within the deadline, the backup algorithm produces an acceptable and reliable solution [43]. Sass et al. proposed a novel model for periodic task scheduling with specific parameters for the operating system in case of emergency situations, inspired by two deadline mechanisms and skip-over models [44]. In [45], both event-triggered and time-triggered traffic in distributed embedded systems were scheduled considering mixed static and dynamic communication over bus protocols.

There are several studies regarding task scheduling parallelism for large-scaled complex distributed systems. Qamhieh and Midonnent in [46] studied the scheduling problem of hard real-time parallel tasks in multiprocessor systems and conducted different simulations to analyze the performance of two global scheduling algorithms, i.e., Earliest Deadline First (EDF) and Deadline Monotonic (DM). Qamhieh et al. in [47] proposed a stretching algorithm to transform a set of parallel graphs of tasks into a set of threads and then employed two global scheduling methods such as EDF to ensure the execution of these threads. Stavrinides et al. in [48] analyzed various scheduling policies such as EDF, Highest Level First (HLF) and Least Space-Time First (LSTF), for assigning priorities to the real-time interdependent tasks in ultrascale systems. For the processor selection phase, they examined 3 bin packaging policies including First Fit (FF), Best Fit (BF) and Worst Fit (WF) in order to utilize the idle time slots. Their simulation results show the efficiency of the EDF-BF scheduling strategy.

Tchernykh et al. in [49], dealt with parallel jobs scheduling problem and proposed two-level hierarchy scheduling approach, where in the first level, computational jobs are assigned to the parallel computers and then in the second level, each local scheduler gen-

erates schedules of its own jobs. In [50] Ahmed et al. proposed two-level approach for the base station uplink scheduling which supports the Quality of Service (QoS) for different classes of traffic in digital cellular networks. The first level is to allocate the bandwidth in order to ensure the QoS and priority for the all types of traffic and high bandwidth utilization. The second level deals with the distributing the bandwidth among the flows for each traffic class. In [51], a hybrid meat-heuristic approach was developed for scheduling tasks in heterogeneous parallel computing systems, which aims to minimize the total execution time. The proposed algorithm performance was compared in term of average makesapn to the other exciting scheduling solutions. In [52], a comprehensive real-time task scheduling approach was proposed for complex embedded systems. A dynamic measurement model was also established to securely change the scheduling methods for different tasks based on the deep learning networks.

The state-of-the-art scheduling solutions are developed for monolithic systems which can not fulfill the requirements of SoSs. Complex heuristic methods for solving the parallel task computing also cannot fully cope with the SoS architecture. In these complex systems, components are not fully independent and they still rely on a central control authority for decision making, while the local schedulers in the SoSs are independent from the operational and managerial point of view.

The other important characteristics of a real-time system is to have flexibility and resilience towards faults. Several researchers dealt with hardware-based solutions to tolerate permanent faults, which may bring on high hardware costs. There is also a lot of work done on combining fault-tolerance policies and scheduling methods in real-time embedded systems. Bertossi and Mancini in [53] proposed efficient solutions to preemptively schedule a set of independent periodic tasks. The state of each task will be checked and when it is completely executed, the next same one can start. Burns et al. in [54] analyzed the schedulability of time redundancy technique for safety-critical systems. The re-execution of tasks can cause missing the deadlines and affect the required predictability for the safety-critical applications. Han et al. in [55] considered different possible software faults and scheduled two versions of real-time periodic tasks based on their functions, the precise of their results and ease of verification. Zhang et al. in [56] integrated the check-pointing scheme into dynamic scheduling approach for the real-time tasks in embedded systems. The simulation results show the better performance of their approach in completing tasks in the presence of faults as well as reducing the power consumption.

The aforementioned preemptive on-line scheduling solutions for supporting fault-tolerance have less predictability but more flexibility toward unpredictable faults comparing to the static off-line scheduling approaches. Several approaches have been proposed to integrate the fault-tolerant techniques into the static schedulers. Izosimov et al. designed an optimization scheduling approach using re-execution and replication of processes in time-triggered communication [31]. Chetto and Chetto [57] developed a model in which the

current task schedule is replaced with other off-line-precomputed schedules upon failures. Their model guarantees to meet hard deadlines in the face of failures while maximizing the chance of success for the primary schedules.

All the aforementioned scheduling solutions can be applied as a scheduling method for real-time traffic inside each constituent system. However, in an SoS it is still required to have a global scheduler to share the common resources and schedule the real-time communication between these constitutes. The closest study to our work is [58], where the restrictions and challenges of scheduling process in an SoS structure is addressed. Murshed in [58] developed a scheduling model for real-time communication in an SoS based on Mixed Integer Linear Programming (MILP). The author conducted some simulation experiments to examine the scheduling results. Considering the NP-hard nature of scheduling time-triggered traffic, ILP-based models are time-consuming and in large-scale systems are incapable of finding good solutions in an acceptable runtime.

Moreover, we reviewed the research regarding real-time incremental scheduling in distributed embedded systems, which is also a well-researched problem in monolithic systems but not in SoSs. The first study in the incremental design of distributed embedded systems was by Pop et al [59]. They proposed an incremental design process for a broadcast communication channel designed for real-time applications. They applied an approach to map and schedule new applications so that the previous schedules are not disturbed. In [60] they have discussed the implications of an incremental design process in the context of a fixed-priority preemptive scheduling policy. Schoeler et al. [61] proposed an incremental scheduling model based on the Satisfiability Modulo Theories (SMT) approach to compute static schedules in a multi-cluster system and compared the results with an optimal scheduler based on MILP. In [62] authors proposed various scheduling algorithms based on ILP to incrementally add time-triggered flows in a time-sensitive software-defined network.

Several works have introduced different scheduling policies in real-time networks to deal with overloaded systems in a dynamic environment. In [63] authors proposed a novel framework to handle possible overloads while the tasks are dynamically scheduled in a single processor environment. The framework executes a sequence of approximate scheduling algorithms while adjusting the load and refining the quality of the solution. In [64], an approach was developed for process scheduling in computer systems which are used in control applications such as spacecraft. The best-effort service is introduced as a rejection policy for overloaded systems by removing tasks with the minimum value density. The scheduler is evaluated in a real-time system simulator for a heavily loaded system.

Obermaisser et al. in [65] addressed the incremental scheduling approach in time-triggered SoS networks. They compute schedules for multiple applications arriving incrementally using IBM CPLEX. They examined the feasibility of solutions on different generated examples, however, in this approach each application is optimized individually and the requirements of future applications are not considered. Therefore, a heuristic

method is developed and integrated in our scheduling approach to increase the chance of finding feasible solutions for the future applications.

## 3.3 Development Methods and Models for SoSs

The complexity and multidisciplinary of the SoS challenges require a new customized engineering effort in design, implementation and monitoring to prevent wasting the resources due to the poor system performance. This is a particularly concerning issue in scheduling the process inside and among the constituent systems of an SoS. Therefore, a strong conceptual foundation in this domain and awareness of their specific requirements and tactical needs are required to cope with the complex SoS problems. In this section, we review the fundamental studies in the SoS domain which can be used as a guideline.

The different standard architecture frameworks (e.g., DoDAF and MODAF) discussed in the previous chapter are to model an SoS from different viewpoints and determine the perspectives and the interdependencies of collaborating constituent systems. However, these architectural frameworks are mainly prescriptive and not focused enough on concrete modeling methods. There are major attempts in the SoS domain focused on analyzing these standard frameworks to customize them for specific applications and develop executable models for different SoS architectures.

Vierhauser et al. in [66] conducted a systematic review on the existing requirements-based monitoring frameworks for the software systems and analyzed the suitability of theses approaches for the SoSs. A comprehensive overview of the theories, methods, and solutions in modeling and simulation for SoSE can be found in [67]. Nielsen et al. in [68] also identified the state-of-the-art SoSE activities and the existing research challenges. Kilicay et al. focused on developing appropriate tools and methodologies to customize the behavior of the complex adaptive systems for an SoS using Complexity Theory [69]. Jamshidi in [3] covered all the fundamental concepts and terms of SoS and principles for SoSE. Acheson et al. in [70] developed an agent-based model to simulate the dynamic interactions between the independent constitutes and studied the key factors that influence the performance of an SoS. Guessi et al. in [75] addressed the architectural feasibility challenges from forming new coalitions in an SoS and its governing rules. They presented an approach to validate the coalition feasibility from the architectural perspective in the flood monitoring SoS.

Built on the previous conceptual work, there are notable studies which describe specific SoS applications and integrate different system engineering activities within this type of system. Kotov in [71] dealt with modeling and analyzing the communication and data transmission in an SoS consisting of complex distributed components (e.g., enterprise intranets). Since the standard architecture frameworks do not support real-time SoSs, Sanduka established a modeling framework to satisfy the real-time and reliability requirements in an SoS [35]. Lane and Turner in [72] dealt with the information flow management sys-

tem to enhance the visibility in interacting the subsystems of large operational systems. Turner et al. in [73] integrated the lean process concepts, e.g., the kanban scheduling system, within a large-scale hospital system. Oquendo in [74] introduced software-intensive SoS and addressed the potential challenges.

## 3.4 Research Gap and Contribution

Based on the literature survey, most of the research found in the SoS domain is about developing conceptual models of SoSs and rarely deals with mathematical modeling concerning scheduling and operational planning. The time-triggered scheduling algorithms in the state-of-the-art were mostly developed for monolithic systems, which do not support the constraints of SoSs. Modeling and optimizing the resource allocation in an SoS is more complex than in monolithic systems due to the autonomy of the constituent systems, the lack of central control and no global information about the resources in different constituent systems.

There is a research gap in comprehensive scheduling models for SoS networks with real-time support. The first contribution of this thesis is to introduce a two-level iterative GA-based heuristic algorithm to optimize the allocation and scheduling of SoS applications in TTSoS, which supports the real-time requirements as well as considering the specific constraints of an SoS. To model the scheduling problem in our network, we distinguish two levels, namely the SoS level and the CS level. These two levels interact iteratively to achieve the agreed common goal. The SoS level generates multiple solutions to assign and schedule the services and the shared resources to the constituent systems. These solutions are suggested to the CS level which refers to the local schedulers inside the constituent systems. Each of them provides feedback based on its scheduling strategy and sent them back to the SoS level for the further update. These flows of information continue iteratively to reach acceptable solutions for all the constitutes as well as accomplishing the SoS goal. For every new SoS application, there are two phases of admission and execution. After establishing the new arriving SoS application, which is performed off-line, its schedule is computed at runtime.

As the second contribution, we focus on increasing the reliability of our scheduling system. Therefore, we integrate two fault-tolerance techniques in our scheduling model, i.e., replicating the communication flows and redundant paths for transmitting both original and replicated messages as well as re-executing the computational activities. These fault-tolerance techniques are deployed in both SoS and CS schedulers to leverage the reliability of system.

As the third contribution, we design an incremental scheduling approach for a sequence of SoS applications arriving in the future. In this regard, we propose a new allocation method which efficiently shares the resources among the constituent systems as well as

defining a new fitness function for our GA schedulers to balance the trade off between minimizing the completion time of the current established application and reserving the common resources for the future applications.

# 4 System Model of Time-Triggered SoS (TTSoS)

There are many examples of SoS applications that rely on real-time communication, e.g., traffic control or emergency response. Time-triggered architectures can manage the high complexity of control in these large-scale time-critical SoS applications and support higher dependability and determinism [10].

Due to the wide availability of time-triggered technologies for SoS's constitutes, e.g., TSN [11] and TTE inside the constituent systems, we realize the strengths of time-triggered control in the SoSs and introduce the time-triggered SoS (TTSoS). In a TTSoS, each constituent system is a networked embedded system that consists of end systems which are interconnected by time-triggered real-time communication networks with different protocols and topologies. The interconnection of constituent systems occurs using a backbone communication infrastructure consisting of multiple network

In this chapter, we propose a scheduling model for the time-triggered operations in the TTSoS. Establishing time-triggered schedule for the TTSoS guarantees the timeliness of the services. Besides the high determinism, time-triggered control brings the ability of fault tolerance, e.g., by letting multiple services and performing redundant computations and voting on the outputs. Additionally, due to the time-triggered tables, there is no need for explicit synchronization.

## 4.1 Physical and Logical models of the SoS

Generally, SoSs can be explained from logical and physical perspectives. At the physical level, the SoS contains a set of networked independent constituent systems and network domains as depicted in Figure 4.1. Each constituent system can have a complex internal structure with end-systems and switches, which are not visible to the other constitutes. The network domains in an SoS serve for the message-based communication between constituent systems and include switches with complex structures which are hidden outside of their physical scopes.

From a logical point of view, the SoS comprises applications, each consisting of distinctive services with precedence constraints and possessing a deadline, which determines the maximum makespan for the completion of its services. Each service possesses jobs

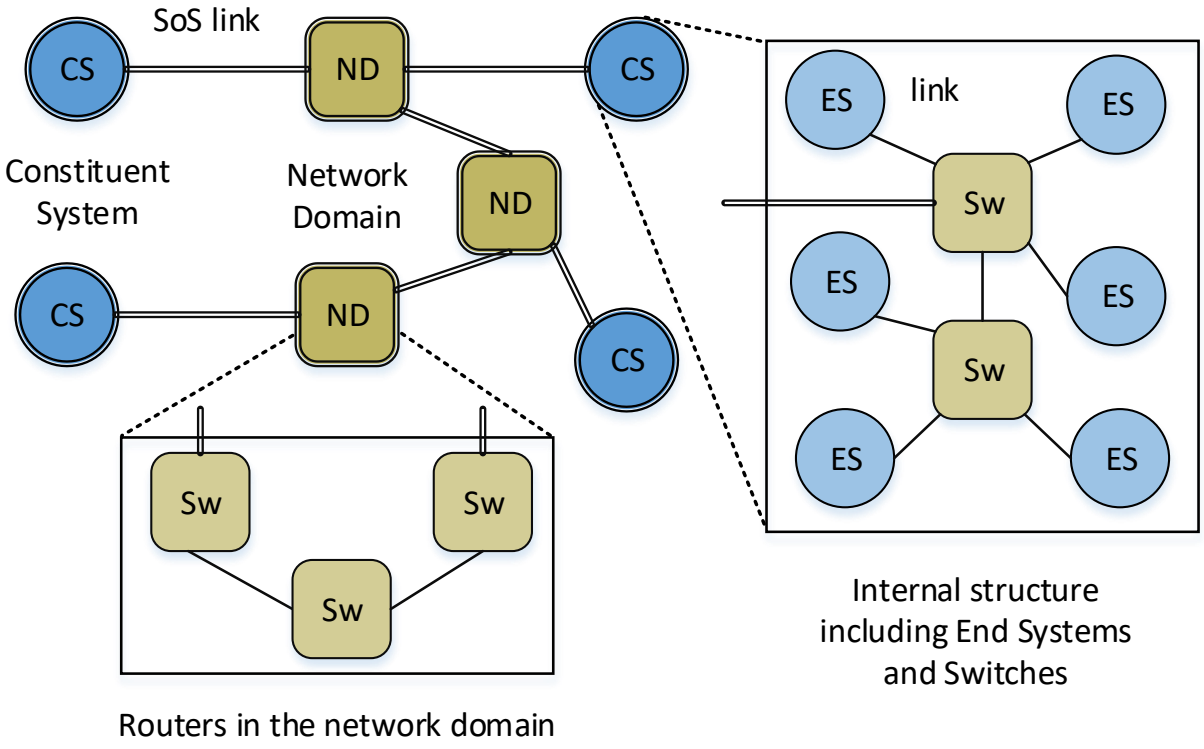


Figure 4.1: Physical Structure of SoS

and messages which are organized in a Directed Acyclic Graph (DAG) and are not visible outside the service. Figure 4.2 depicts the logical model of an SoS.

Each service of an SoS application needs to be provided by a corresponding constituent system, while satisfying the precedence constraints between the services and the deadline of the SoS application. In order to provide a service, a constituent system will in general perform a decomposition of the service into computational jobs (cf. left hand side in Figure 4.2). These computational jobs are not visible to the environment of the constituent system. Their computational jobs are organized in a DAG and must be mapped to the end-systems of the corresponding constituent system.

As a real-world example of an SoS application, we consider a medical monitoring system. In this system, there are different organizations and parties involved such as a hospital, a ambulance center, a medical data laboratory, a medical doctor, and a patient. The medical monitoring system is an collaborative SoS with real-time requirements, which should provide a reliable service, i.e., detecting a medical emergency when the patient is alone at home, with temporal guarantees. The doctor as an independent constituent system can be considered as an initiator in our terminology and establishes the medical monitoring system at the patient home after leaving the hospital. When the admission is completed and the SoS application was scheduled, the medical monitoring application

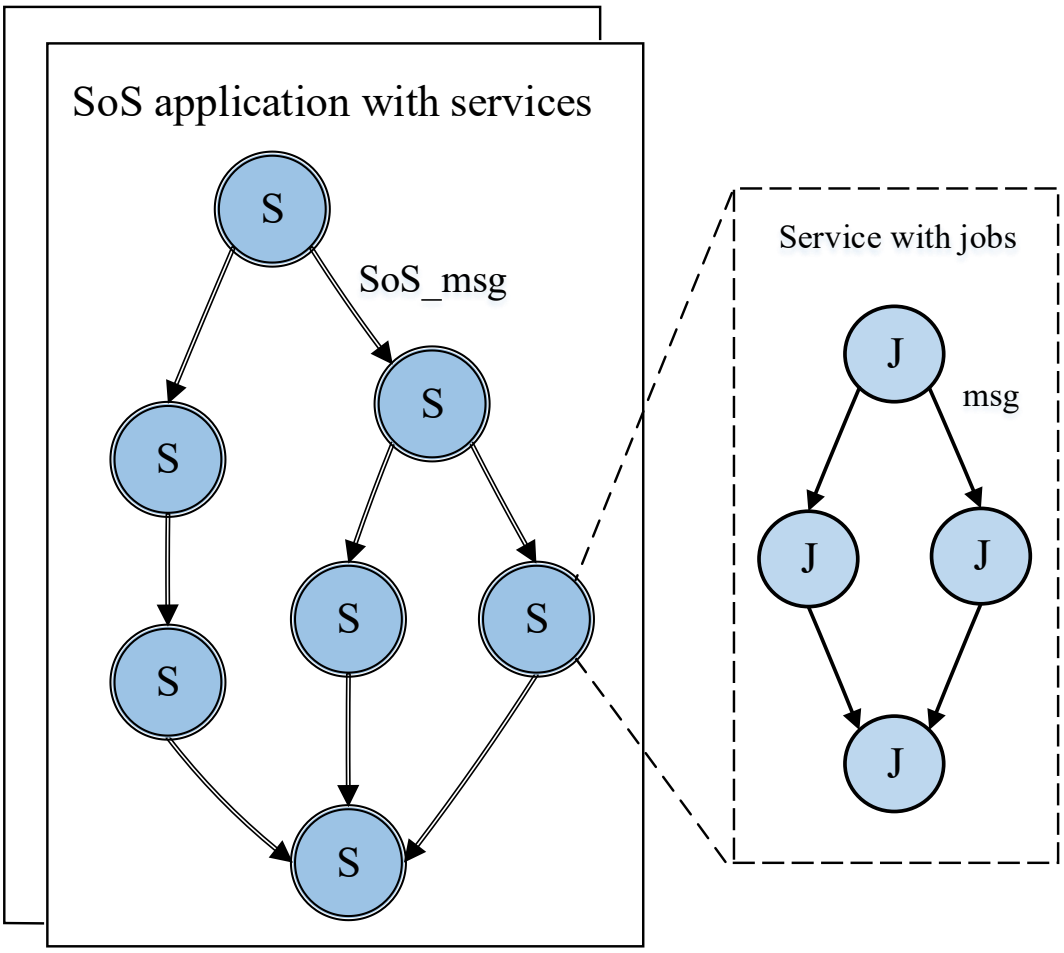


Figure 4.2: Logical structure of SoS

starts to work with reliability and real-time requirements.

In a TTSoS, each service is allocated a time slot with a specific start instant with respect to a global time base. Likewise, each message between two constituent systems is allocated a time slot, in which the message has to be transported along one or more network domains and is received by the destined constituent systems. The deadline of a service determines the earliest possible start instant for injecting its messages. The scheduled reception time of a message at a constituent system marks the earliest possible start instant for the execution of the service that relies on the message. Respectively, each time-triggered constituent system allocates time slots to the execution of jobs on its end-systems as well as to the communication of messages between the end-systems. Table 4.1 summarizes the time-triggered schedules of the SoS and of constituent systems.

The above-mentioned scheduling decisions are generally made by the management building blocks of constituent systems (e.g., CS manager). Each CS manager also determines

the collaboration extent of its system with other constituent systems. The interaction between the constituent systems will be managed by the network management services.

Table 4.1: Time-Triggered Schedule in an SoS

| **Entity** | **Mapped to** | **Scope** | **TT Schedule** |
|---|---|---|---|
| Service | Constituent system | SoS | Start instant and deadline |
| SoS-message | Path between constituent systems along network domains | SoS | Send instant and deadline |
| Job | End system | CS | Start instant and deadline |
| Message | Path between end systems along switches | CS | Send instant and deadline |

## 4.2 Dynamic Establishment and Scheduling of SoS Applications

One of the characteristics of an SoS is its dynamic composition where applications and constituent systems can be dynamically established, modified and removed at run-time. Therefore, we distinguish two scheduling phases: first the establishment and scheduling phase of a new SoS application and the later execution of the SoS application. In the first phase, the scheduling table with time-triggered activities is generated, i.e, assigning the services to the constituent systems, start instant of jobs and messages (see Table 4.1), which will be dispatched in the execution phase. There is no need for the temporal guarantees of the scheduling phase, which might take minutes. However, in the execution phase there are strict temporal limits.

We define the time-triggered scheduling model in two scopes of SoS and CS. In the SoS level, we denote the constituent system that establishes a new SoS application as the initiator. For each service, the initiator discovers a set of compatible constituent systems that can provide the service by using DNS or by contacting a broker (e.g., [76]). The SoS-level scheduler has following tasks,

1. **SoS Allocation.** Allocation of the services to their compatible constituent systems,
2. **SoS Communication.** The initiator determines the routing plan of SoS messages between the respective constituent systems.
3. **SoS Schedule.** Scheduling the SoS-messages and services.

The initiator establishes a set of solutions for the new SoS application and sends an optimization problem to each compatible constituent system of every solution. These solutions determine execution time windows for the services and mapping them to the constituent

systems. For each solution, the local schedulers inside these candidate constituent systems should provide local schedules and send them back to the SoS level as a feedback. Each local scheduler starts with unrolling its assigned service and establishing a DAG with jobs and time-triggered communication messages. The local schedule comprises the temporal and spatial resource allocation such as assigning the jobs to the end-systems, the allocation of messages to paths between the end-systems, determining the execution time of jobs and the injection time of messages. The CS-level scheduling problem has the following characteristics:

- **Service deadline**: Each service should be executed in a time period which is determined by the SoS-level scheduler w.r.t. the SoS application deadline.
- **Real-time jobs**: Each job is characterized by its deadline, resource requirements, and worst case execution time (WCET).
- **Precedence constraint between the jobs**: We define the order of executing jobs regarding to this constraint.
- **Time-triggered messages**: Each message is sent by a job and must be delivered within its deadline.
- **Communication costs**: It is assumed that there is a communication cost to transmit messages from a job on one end-system to a succeeding job on a different end-system. The communication cost between two tasks on the same end-system is assumed to be zero.

The SoS-level evaluates the performance of local schedulers based on the lateness of their services w.r.t. the suggested deadline provided by the initiator. After collecting the local lateness values from the constituent systems, the initiator updates the initial solutions and requests local schedulers for repeating the scheduling process until the feasible schedules for all services are obtained, otherwise this iterative process is stopped once reaching a number of iterations.

The proposed scheduling model will satisfy the special boundary conditions of SoS, e.g., there is no need for global knowledge or centralized control, and will also guarantee the timeliness and the safety requirements with the help of the time-triggered plan.

## 4.3 Time Synchronization in SoSs

Similar to any cyber-physical distributed system, the role of the time is fundamental in SoSs. The time-triggered message-based communication inside constituent systems and network domains of an SoS must be synchronized. Each autonomous constituent system

has its own local timer to make local measurements but it is uncoordinated with the timer in any other constitutes. In order to synchronize messages in an SoS, we resort to the clocks. With a synchronized clock, we can measure the duration of messages which start from one constituent system and end in another constituent system.

Additionally to the synchronized clocks in each constituent systems, it is required to establish a global SoS time across constituent systems nodes to solve the temporal coordination problem in an SoS [19]. The clocks synchronization in an SoS can be internal by using the Precision Time Protocol (PTP) or assigning a global timestamp to every message transporting within the whole SoS. Since the global timestamps may not be shared or interpreted correctly by different constituent systems, an external global time is considered as a preferred means of clock synchronization in an SoS. The external global time can be established by synchronization standards such as Global Positioning System (GPS) or Global Navigation Satellite Systems (GNSS).

Accordingly, we assume that the SoS-level initiator allocates a time slot to each service with a specific start instant concerning a global time base. Likewise, each SoS-message between two constituent systems is assigned a time slot, in which it has to be transported along with one or more network domains and is received by the destined constituent system. The deadline of a service determines the earliest possible transmission instant for a message produced by the service. The scheduled reception time of a message at a constituent system marks the earliest possible start instant for the execution of the service that relies on the message.

# 5 Two-level Interactive Scheduling Algorithm for TTSoSs

The main optimization challenge in scheduling real-time applications in a TTSoS is to temporally and spatially allocate shared resources and services with strict temporal constraints to the set of constituent systems which are candidates to provide these services. This process should be done without any central control authority and limited access to the internal operations of these constituent systems. Moreover, possible conflicts may happen in accomplishing the SoS goal as a whole and achieving the individual goals of its constituent systems. To overcome such issues, we propose a semi decentralized scheduling approach which lets the constituent systems independently optimize their own schedules, while promising to satisfy the overall temporal constraints of SoS applications and minimize their completion time. The constituent system which announces a new SoS application will be in charge of coordinating the other constituent systems and it initiates the scheduling process as well.

In the previous chapter, we proposed a scheduling model in two levels (namely SoS and CS) that enables the constituent systems to interactively process their schedules to accomplish the desired SoS goals without requiring to exchange information about the internals of their operations. In our proposed model, the initiator constituent system manages the SoS-level scheduling process by establishing a new SoS application with a set of services and generating multiple high-level allocation and scheduling solutions to guide other constituent systems in providing the services. These SoS-level solutions include selecting a set of compatible constituent systems and determining approximate execution time intervals for the services satisfying their respective dependencies and the overall deadline of the SoS application. The SoS-level scheduler also conducts the scheduling and the path selection for the time-triggered messages (called SoS-messages) which are transmitted between the chosen constituent systems. The CS-level scheduling model is used by different schedulers inside the constituent systems, where each manages all aspects of the constituent system's own schedule with respect to the determined time windows from the SoS level. These high-level and local schedules are interactivity exchanged between these two levels and updated until reaching acceptable results for all constituents.

Due to the NP-hardness of scheduling problems and better scalability of heuristic algorithms such as GA in solving such problems compared to the performance of optimal

optimization methods (e.g., Mixed Integer Linear Programming), we develop a heuristic approach based on GA which operates at two interactive levels to schedule real-time applications in TTSoS networks. Figure 5.1 gives the overview of our proposed algorithm. The GA optimizer is employed in both levels with different genome structures and fitness functions. In each generation of the SoS-level GA, multiple solutions are generated and sent to the target constituent systems. Thereafter, each CS-level scheduler using GA tries to find a schedule for each solution which can meet the deadlines or have a minimum lateness. Each constituent system will send the output of the optimization process to the initiator for the further updates at the SoS-level by using the genetic operators (i.e., crossover and mutation). In the following sections, we explain both SoS and CS levels scheduling algorithms in detail.

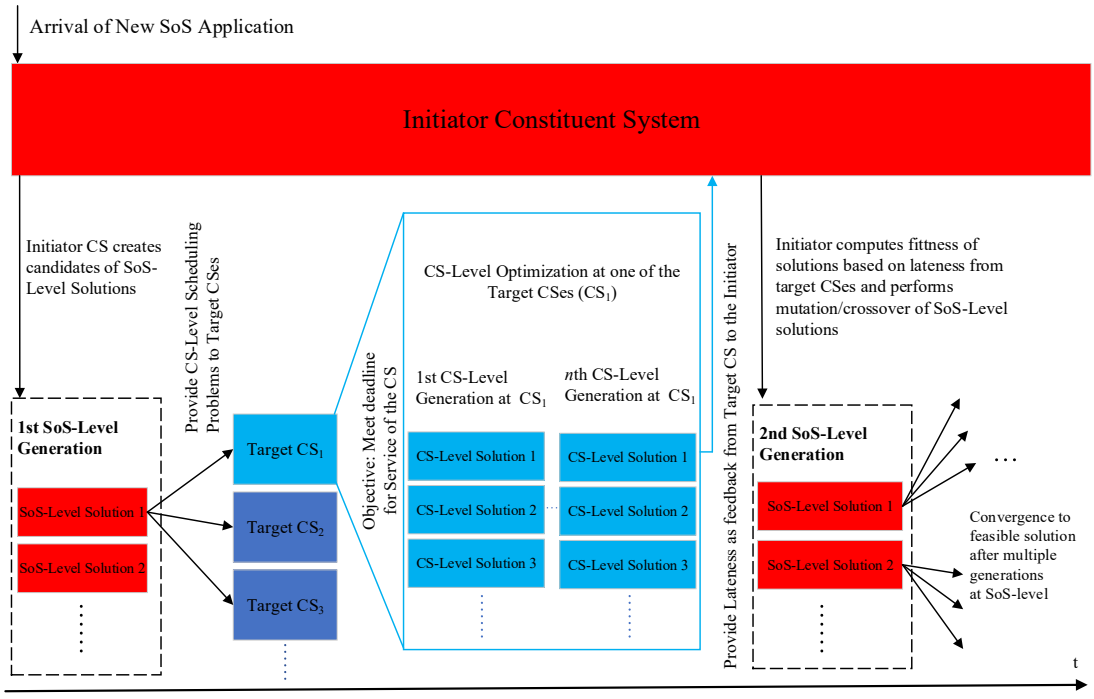


Figure 5.1: An overview of the two-level interactive GA for scheduling TTSoS applications

## 5.1 SoS-level programming

The SoS-level scheduling approach is defined based on the GA and operated for each new arriving SoS application by unrolling its services and messages. This level's tasks include generating global solutions for scheduling the services as well as optimizing the routing and scheduling problems of SoS-messages. These global solutions comprise the following information,

- **Service Allocation**: Each service is mapped to exactly one constituent system.
- **Service Scheduling**: A time interval is considered to execute each service satisfying its precedence constraints based on the SoS application DAG and considering the delays of the SoS communication.
- **SoS communication**: A deadline and path are considered for delivering each TTSoS-message.

Accordingly, the SoS-level genome should contain information regarding the allocation of each service to its compatible constituent system, a time interval for its execution besides a path for each SoS-message. Therefore, the SoS-level genome is defined with 3 types of genes, namely allocation, scheduling, and path selection, which are depicted in Figure 5.2. Each gene is composed of multiple alleles, each can contain a single value or an array of values. In our genome, the allocation gene has a definite number of alleles $K$, each is defined as an array with the length of $N$ (equal to the number of services). Each allele presents the allocated constituent systems for all services. Likewise, the scheduling gene is defined as an array of arrays, each is dedicated to store the time budgets for all the services and the SoS-messages. These time budgets will be later processed to find the feasible time windows for executing the services based on their dependencies.

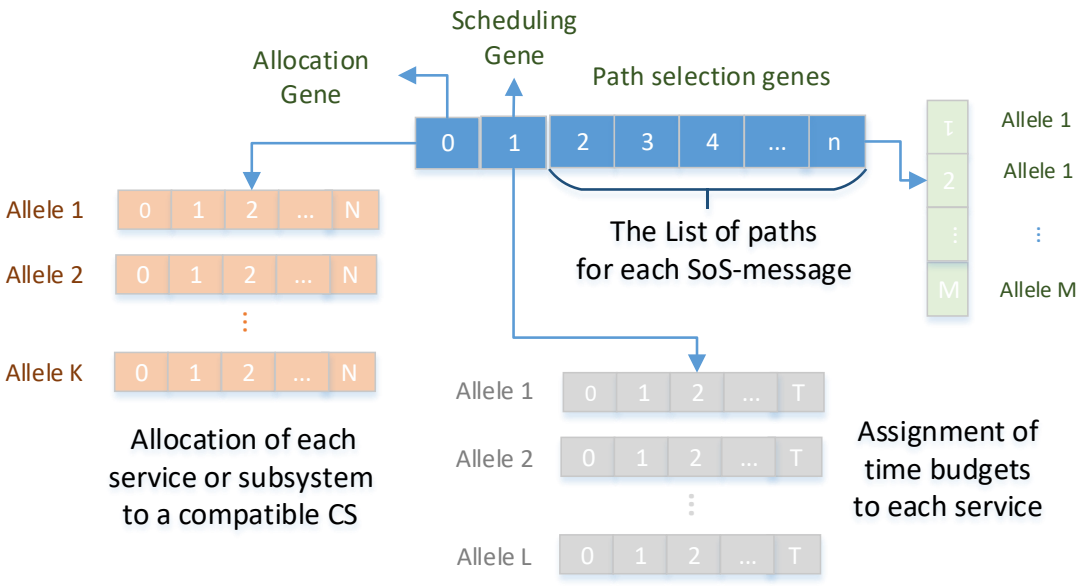


Figure 5.2: Representative genome for the SoS-level GA scheduler

The third part of the SoS-level genome is related to the routing information assigning one gene to each SoS-message. These genes keep the index of routes between each two

constituent systems. The maximum number of these indexes depends on the topology of the network and the applied routing algorithm to find the existing routes. When there are two routes between two specific constituent systems, theses indexes will be 0 and 1 meaning the first and second shortest path found by the proposed routing algorithm based on the Yen's algorithm [77]. Overall, the total length of the SoS-level genome depends on the number of SoS-messages of the application. The possible combinations of exchanging alleles from different genes create new solutions for the SoS-level scheduler.

We consider that an SoS application with 5 services and 6 SoS-messages is being scheduled in an SoS with 4 constituent systems. In this example, the SoS-level genome will have 8 genes. As Figure 5.3 shows, the first gene is defined as an array of arrays, each with the length of 5 to save the allocation of services to the constituent systems with respect to their dependencies, e.g., one possible allele is defined as follows: Services 0-4 are mapped on the constituent systems 0, 1, 3, 2 and 0, respectively.

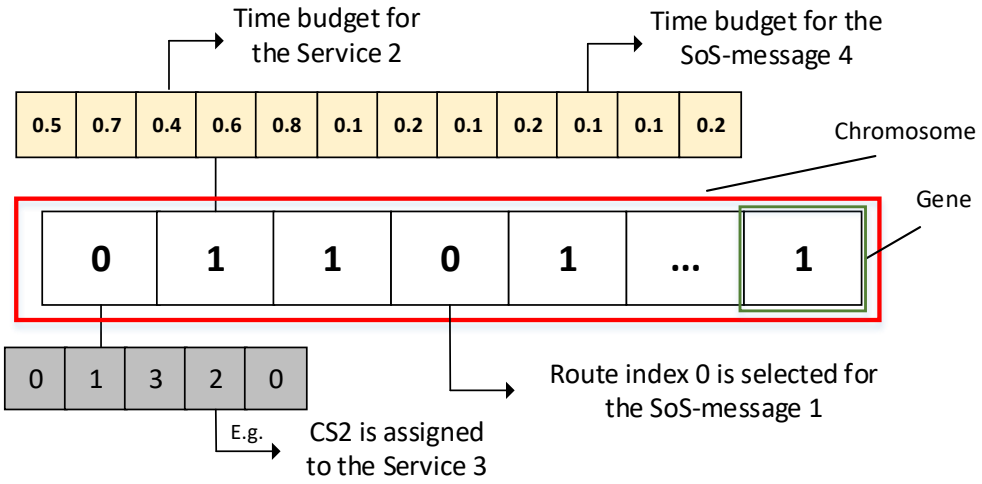


Figure 5.3: An example of an SoS-level genome

The next gene is an array of length 11 and dedicated to the generated time budgets for executing the services and delivering the SoS-messages. In the following allele, we consider 0.7 $ms$ to complete the execution of service 1. The next genes show the indexes of routes selected for the SoS-messages. In this example, the first shortest path between any two constituent systems will be reserved for the message 1. Figure 5.4 shows one of the possible genomes for the aforementioned example as a graph which is generated for scheduling an SoS application with the overall deadline of 3 $ms$.

The SoS-level initiator starts with establishing an initial population $P$ in the first generation from a set of genomes or global solutions (see Algorithm 1). Each solution can

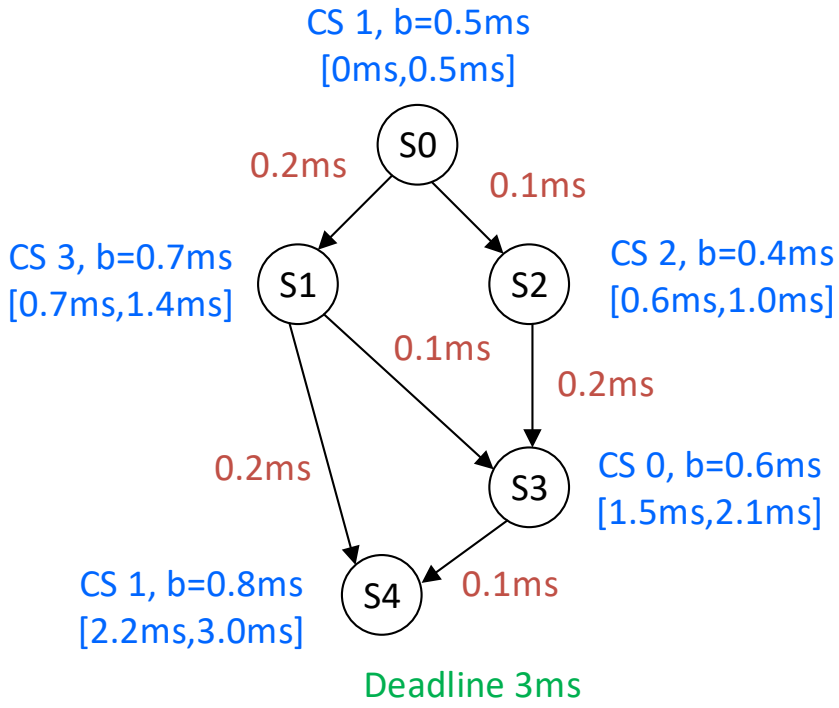


Figure 5.4: Graph-based SoS-level solution

be divided into two types of schedules ($schedule_s$, $schedule_m$), which are generated for the services and the SoS-messages. $Schedule_s$ contains the identifiers of compatible constituent systems mapped to the services, their earliest start time of execution and duration. $Schedule_m$ includes the path id and the deadline for delivering the SoS-messages. The injection time of theses messages are determined by their sender constituent systems after finishing their schedules.

The initiator sends these global solutions to the target constituent systems and waits for the feedback from the CS-level schedulers which are provided autonomously by the constituent systems. These local schedulers operate independently outside the sphere of control of the initiator. Each constituent system sends back the lateness of its local schedules for different solutions to the initiator. Regarding the received results, the SoS-level scheduler evaluates its initial solutions by a fitness function based on the maximum lateness (i.e., the fitness function is to minimize the maximum lateness of services and messages). Afterward, the initiator selects the best ones for the next generation, as well as generating new solutions using mutation and cross-over operators. Thereafter, the initiator requests new solutions from the constituent systems, which subsequently lead to new fitness values and new global solutions. The same procedure iterates in each generation until finding a solution with the fitness value of zero or it stops after reaching the maximum

number of iterations.

**Algorithm 1:** Two-level interactive GA-based scheduling algorithm

**input:** new SoS Application and deadline $< A, d >$
DAG $A = \{< S, M >\}$: Services $S$(vertices), SoS-messages $M$(edges),
SoS network topology undirected graph $G = \{< V, L >\}$:
Constituent systems ( $CS$) and Network domains ( $ND$):
$CS \cup ND \in V$, Links $L$,
Identifiers of compatible $CS_j$ for the service $s_i \in S : comp : S_i \rightarrow \{CS_j\}$,
Table of Routes $R$,
GA Parameters (number of generations, population size, ...)

**begin**
Run SoS-level scheduler:
Initial population $P = \{So_1, So_2, \ldots So_{npop}\}$
$Solution\ So_j =< schedule_s, schedule_m >$
$schedule_s : s_i \rightarrow < CS_i,\ EarliestStartTime,\ Deadline >$,
$CS_i \in comp(s_i), EarliestStartTime,\ Deadline \in \mathbb{N}$
$schedule_m : m_i \rightarrow < Path,\ Deadline >,\ \ Path \in R,\ Deadline \in \mathbb{N}$
**repeat**
$\forall\ So \in P$ :
$\forall\ schedule_s \in So$ :
Run local GA scheduler in the allocated constituent system:
$FinishTime(s_i) = Makespan(s_i) + EarliestStartTime(s_i)$,
$Lateness(s_i) = \max(0, Makespan(s_i) - Deadline(s_i))$,
$\forall\ schedule_m \in So$ :
$InjectTime(m_i) = FinishTime(sender(m_i))$,
$ArrivalTime(m_i) = InjectTime(m_i) + length\ of\ path$,
$Lateness(m_i) = \max(0, ArrivalTime(m_i) - Deadline(m_i))$,
SoS-level fitness function: $fitness(So) = max\ (Lateness(m),\ Lateness(s))$,
mutation/crossover of solutions in $P$,
**until** $fitness(So) \leq 0$;

## 5.2 CS-level programming

For each constituent system, we also assume a GA-based scheduler, which runs locally and independently. This local scheduler starts with receiving information about the list of jobs and messages related to its assigned service, the suggested earliest start time of execution, and the deadline. The following tasks are defined for this level,

1. Allocation of jobs to end-systems,

2. Routing the messages of each service,

3. Scheduling the jobs and messages of each service.

Similarly to the SoS level, CS-level GA starts with generating an initial population consisting of different genomes. The CS-level genome is defined in three parts including allocation, ordering, and path selection genes. In the following genome, one gene is allocated to each job which holds its compatible end-systems. The ordering gene stands for the different feasible sequences of traversing through the DAG of each service. Finally, the path selection genes are generated for the messages with respect to the routing algorithm (see Figure 5.5).

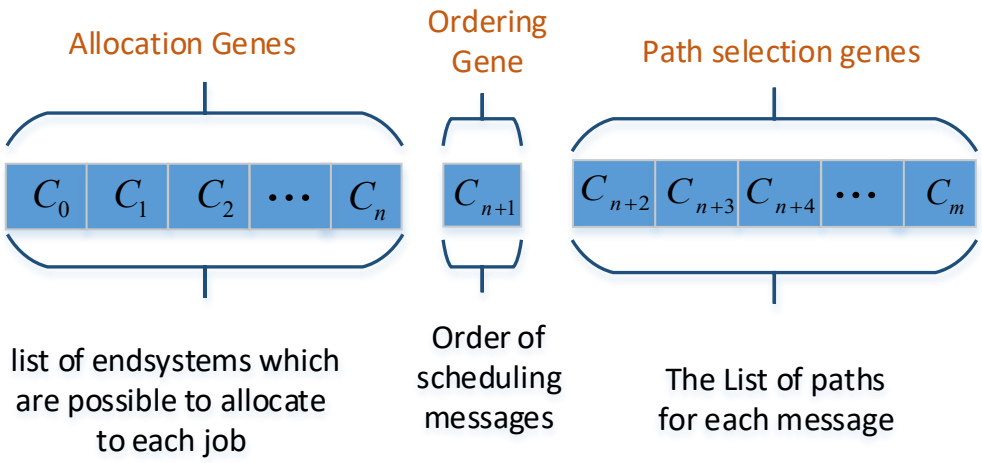


Figure 5.5: Representative genome for the CS-level GA scheduler

In each generation, a set of genomes is used by the GA solver and the corresponding schedules are evaluated by the fitness function, i.e., minimizing the service lateness. Afterward, the best schedules are selected for the next generation besides generating new genomes by applying mutation and crossover operators. The algorithm stops either when a feasible solution (i.e., fitness value greater than or equal to zero) is found or it reaches a specified number of iterations. Algorithm 2 describes the local scheduling process for each genome. It starts with allocating each job to a compatible end-system and determining the earliest start time of execution for all jobs based on the suggested value from the SoS level. Each job can be executed when the messages from its predecessor jobs are delivered. The execution of a job is started after finding the earliest feasible time slot based on its worst case execution time (WCET). Each message likewise is started to transmit within a feasible time slot according to its selected shortest route. Every message is sent by a job

and must be delivered within its deadline. The fitness function is to minimize the lateness based on the assigned deadline to each service.

**Algorithm 2:** CS-level GA scheduling algorithm

**input :** *Service DAG*, $G = \{< J, M >\}$,
jobs $J$ (vertices), messages $M$ (edges),
The earliest start time of the service $EST$,
*deadline*, *Genome* $g$,

$fitness,\ makespan \leftarrow 0$;
**for** *each* $j \in J$ **do**
  $j.RunsOn \leftarrow ES \in J.AllocationGene \in g\ j.StartTime \leftarrow EST$;
**for** *message* $m \in M.OrderingGene \in g$ **do**
  $Js \leftarrow m.sender \in J$;
  $Jr \leftarrow m.receiver \in J$;
  $Js.StartTime \leftarrow$ *find earliest feasible time slot;*
  $m.InjectionTime \leftarrow$ *find earliest feasible time slot;*
  $m.route \leftarrow R \in m.PathSelectionGene \in g$;
  $m.ArrivalTime \leftarrow m.\ InjectionTime + hop\ time * R.size$;
  $Jr.StartTime \leftarrow max(Jr.StartTime,\ m.ArrivalTime)$;
  $Jr.FinishTime \leftarrow Jr.StartTime + Jr.WCET$;
  $makespan \leftarrow max(makespan,\ Jr.FinishTime)$;
$lateness \leftarrow deadline - makespan$;
**if** $lateness < 0$ **then**
  Return 0
**else**
  Return $fitness = lateness$

## 5.3 Mutation and Crossover Operators

Generating new genomes is done by applying mutation and crossover operators. The crossover procedure creates new offspring from each two mating genomes, by choosing a random crossover point and swapping the genes from the same parts with each other. The crossover point can happen in single or multi points. Figure 5.6 shows the mechanism of a single point crossover on our genomes.

To maintain diversity within the population, mutation procedure is performed on single genome to change the allele of one or more cells.

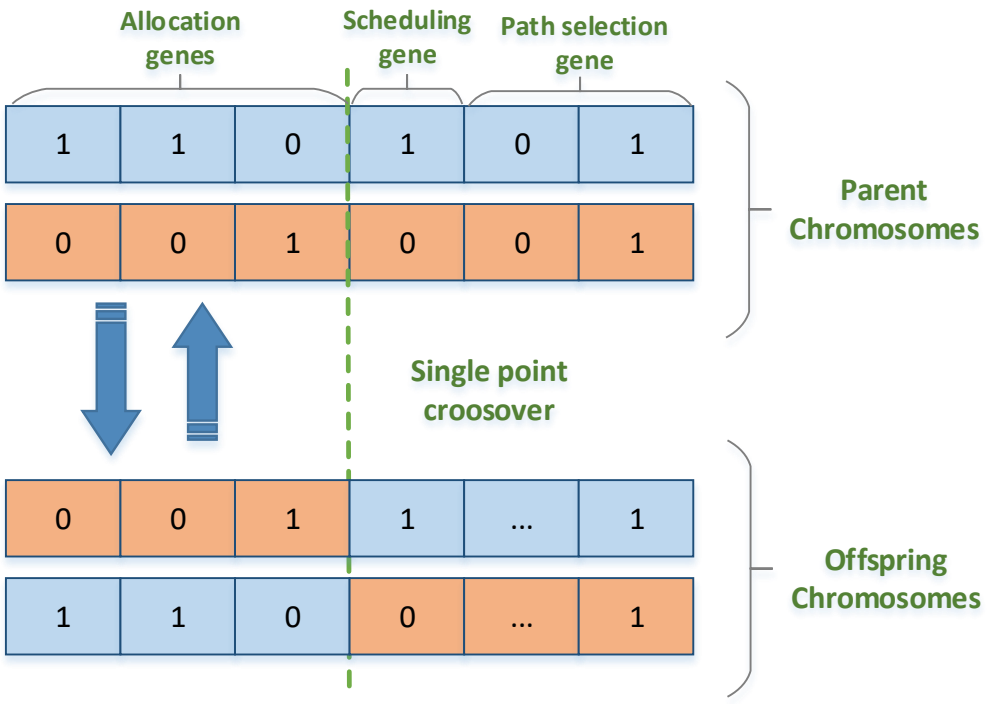


Figure 5.6: Exchanging genes among parents in a single point crossover

## 5.4 Incremental Scheduling for TTSoSs

In the previous sections, we presented our two-level scheduling process using GA for scheduling every new arriving SoS application individually. However in the real world, SoSs may encounter different applications which are introduced and/or removed over time, and they should share the underlying resources of their constituent systems. In case of arriving a sequence of applications, the constituent systems should update their resource allocation approach to contemplate not only the timing constraints of the current SoS application but also the schedulability of applications introduced in the future. Since each constituent system is only aware of the availability and allocations of its own resources, and the limited sharing information between the constitute systems about how to utilize the resources make implementing the incremental scheduling for the SoSs more challenging. In this regard, we extend our two-level scheduling algorithm with a new resource allocation strategy and a new solution fitness evaluation to efficiently schedule incrementally adding SoS applications into the TTSoS. In our incremental scheduling process, it is assumed that each application arrives after finishing the scheduling process of the prior one and the simultaneously arriving different SoS applications will not happen (see Figure 5.7).

Regarding our scheduling algorithm, the scheduling initiator will be determined dynam-

ically based on where an SoS application is requested. The chosen initiator provides global schedules and interacts with the CS-level schedulers using the GA optimizer. The CS-level schedulers determine the local schedules for the services and decide independently how to assign their resources i.e., the end-systems and switches, to a sequence of computational jobs and messages. In case of scheduling a single application by our heuristic algorithm, the CS-level schedulers find and select the earliest possible time slot for executing jobs and transmitting messages in order to finish their schedules in the shortest period of time. Although this approach works efficiently in the sense of completion time, it may bring about unbalanced working time allocation for the resources and increase the likelihood of resource shortage with the appearance of more services to be scheduled. To avoid these problems, we propose a new resource allocation strategy to be used inside each constituent system, which focuses on balancing working time of resources and reserving more free time slots on each resource for the future applications rather than the completion time of the service of the current application. The effect of this new method is measured by a new optimization metric that will be directly reflected in the evaluation process. Accordingly, we define new fitness function for the CS-level schedulers to evaluate the schedules based on the allocation optimization metric as well as the completion time of services.

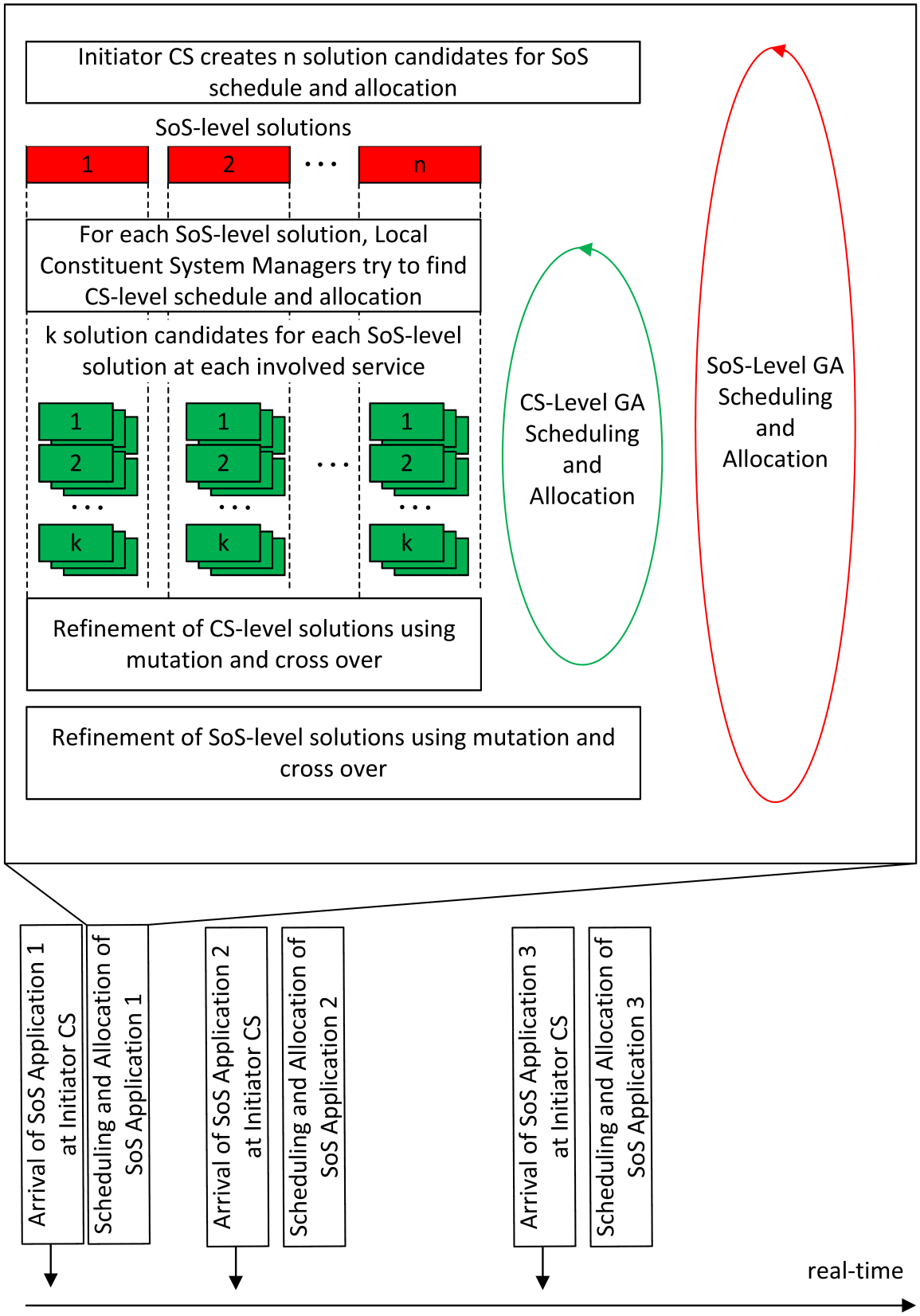


Figure 5.7: The incremental scheduling algorithm for TTSoS applications

### 5.4.1 Resource Allocation Strategy

As mentioned earlier, the current allocation approach aims to finish scheduling tasks in the shortest possible time which results in working efficiently for scheduling an individual application as long as there is no need to reserve the resources for future applications. However, in real cases different applications emerge over time, while the resources are still occupied by the previous applications and a shortage of resources for the future applications is probable. Assuming a set of jobs without any allocation constraints are about to be scheduled in one of the constituent systems, based on this allocation approach all the jobs are probably run on the same resource, since the communication cost between jobs running on the same end-system is zero. This unbalanced allocation causes a problem when the next set of jobs from a different application needs to be scheduled on the occupied resource. Therefore, we have to save free time slots on all the resources beforehand for upcoming applications.

In our new allocation strategy, we focus on reducing the average working times of resources by defining an evaluation function called Maximum Blocking Time (MBT) and a threshold parameter called delta. For scheduling each service on a constituent system, the MBT is computed for its resources as the longest occupied time interval based on the delta parameter, i.e., when the idle time slot between two consecutive working time slots is shorter than delta, this time slot will be treated like a busy time slot for the scheduler as it is not enough to be assigned to any future jobs and messages. The GA optimizer minimizes a fitness function based on the MBT among all resources and the makespan. On the other hand, the scheduler reserves idle time slots on each resource during mapping the jobs. Determining an efficient value for the parameter delta depends on the specifications of a problem, e.g., the WCET of jobs and the hop time in each CS network.

Figure 5.8 shows the difference between theses two allocation strategies. The following problem is to assign the resources of a constituent system with 2 end-systems (es0 and es1) connected through a switch (sw0) to a service with 3 jobs from 3 different SoS applications incrementally adding to the system. In this example, we assume that both the WCET for all jobs and the hop time are equal to 20 $ms$. The first incremental scheduler runs all the jobs of the first application on the same end-system (es0) to save time, because there is no communication cost. When the second application appears, it has to run the jobs on the other end-system (es1). The non MBT-aware incremental scheduler generates schedules with the shortest makespan for these two applications, but it is not possible to assign the current resources to the application 2. Contrarily, the second incremental scheduler establishes the MBT-makespan trade-off and accepts the communication cost which results from balanced use of all the resources. As a result, it is capable of scheduling all three applications.

Algorithm 3 explains the GA incremental scheduler with MBT consideration in more

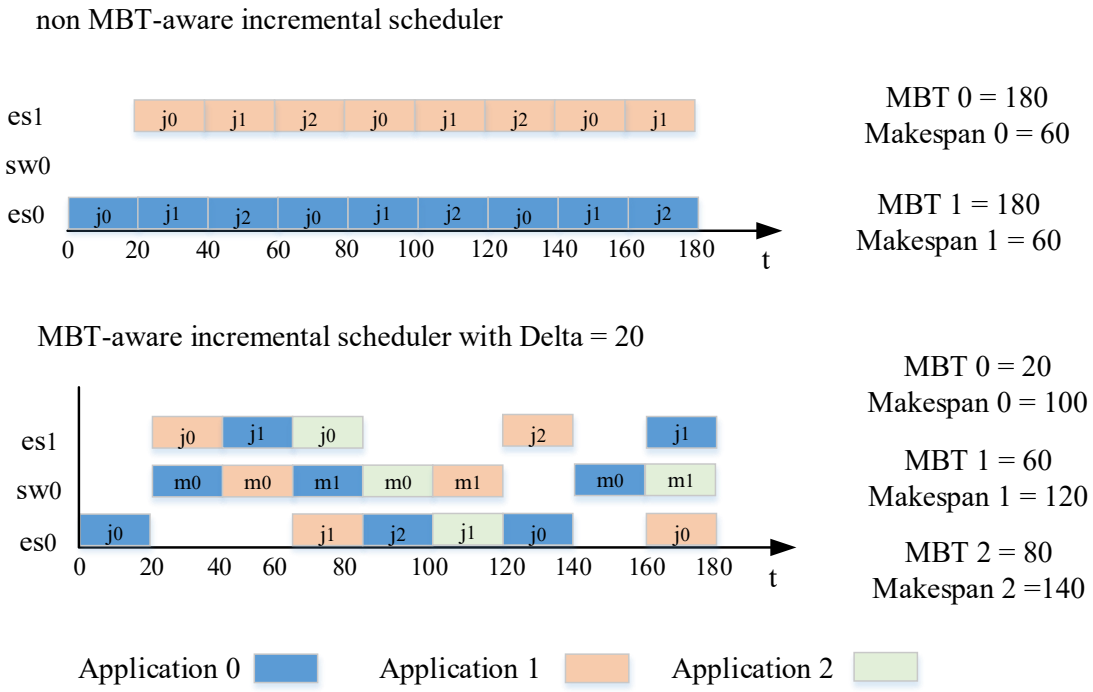


Figure 5.8: Comparing the performance of the CS-level GA scheduler with two different allocation strategies

detail. The scheduling process is run for each application separately upon its release time after finishing the scheduling of previous applications. Similar to our base two-level scheduling algorithm, an initiator creates a set of SoS-level schedules in the initial population $P$ and sends them to the list of corresponding constituent systems, where their local schedulers must determine a schedule within the given time interval. Inside each constituent system, the local scheduler starts with unrolling the services and establishing their DAG with jobs and messages, then it optimizes the temporal and spatial allocation of its resource, i.e., assigning jobs to the end-systems, selecting the start execution instant of jobs and injection time of messages as well as allocating paths to the messages between end-systems. Each local schedule is evaluated based on the MBT of resources and the makespan.

The CS-level fitness function ($F(CS)$) is to minimize the maximum MBT among all the resources and the lateness of the service with respect to its provided deadline by the initiator. After optimizing the local schedules, each constituent system returns its feedback to the SoS level regarding the provided time intervals. The SoS-level evaluation is defined also based on CS-level fitness values. Furthermore, the initiator updates the

initial population using GA operators (mutation and cross-over) to provide new solutions for constituent systems. This iterative process is continued until finding feasible local solutions (i.e., all services are scheduled within the provided time budgets).

**Algorithm 3:** MBT-aware incremental scheduler

**input:** new SoS Application logical and physical models

**begin**

Run SoS-level scheduler:

Create initial population $P = \{S_1, S_2, \ldots S_I\}$, Solution $S_j = \{< sched_s >\}$

$Sched_s : Service_i \rightarrow < CS_i, time\ budget, deadline >, CS_i \in comp(S_i)$, time budget, deadline $\in \mathbb{N}$

**repeat**

**for** $S \in P$ **do**

**for** $Sched_s \in S$ **do**

Run the GA scheduler for $CS_i$ in the time budget with MBT consideration:

Calculate the MBT for all resources;

**for** *Resource* $R \in CS_i$ **do**

**for** *blocking slot* $bs \in n$ **do**

**if** $bs.start_n - bs.finish_{n-1} < \Delta$ **then**

$bs.start_n = bs.start_{n-1}$;

$MBT_R = Max(bs)$ ;

$F(CS_i) = w0 * \max(MBT_R) + w1 * (makespan - deadline, 0)$ , $(w0, w1 \in \mathbb{N})$;

$lateness(Service_i) = \max(0, makesapn - deadline)$;

$F(S) = Max(F(CS_i))$;

Mutation/crossover of solutions and update $P$;

**until** $\sum lateness(Service_i) <= 0$;

## 5.5 Greedy Local search based scheduling algorithm

As a baseline for evaluation the genetic algorithm, we implemented also a two-level scheduling heuristic based on the Greedy Local search (GLS). This algorithm serves as a reference for evaluating the schedulability of the GA scheduler. The used search strategy in the GLS

heuristic is called the "immediate improvement" which means as soon as the algorithm detects a better solution in a neighborhood, it will be selected as the new solution [78]. Based on the structure of the arrived SoS application, the algorithm starts at the SoS level by generating a random global schedule i.e., choosing random time slots for the services and the SoS-messages of the application (with respect to the overall deadline) and random assignments of the services to the constituent systems. This initial global schedule is sent to the scheduler inside each constituent system, which also uses a greedy search method, to find a feasible local schedule within the assigned time slot. The CS-level scheduling algorithm stops either when a feasible schedule is found or when it reaches the maximum number of iterations.

The results from the constituent systems are sent back to the SoS level. In case of a deadline violation in one of the local schedules, the SoS level penalizes the solution by assigning a high value to the cost function, otherwise it saves the makespan of local schedules as a cost function and moves to another global schedule by changing time slots or trying different assignments of the services to the constituent systems. The algorithm stops after reaching the maximum number of iterations.

# 6 Fault-Tolerant Scheduling Algorithm for TTSoS

Considering the breadth of fields where the TTSoS can be used and the deployment for safety-critical applications, the potential effects of failures must be analyzed and mitigated. One of the difficult challenges in distributed systems is to detect faults and attain high reliability and availability. These concerns can be more challenging in large-scale complex distributed systems like SoS. The two important classes of real-time systems based on the safety implications of violating timing deadlines are hard and soft real-time, which can also be considered for SoSs. In a hard real-time SoS, a temporal fault can result in severe consequences and deadly results. Faults can happen in both the SoS-level operations e.g., real-time communication between constituent systems and in the CS-level operations including executing jobs and communication of time-triggered messages between end-systems. Besides the error-detection, these systems must continue to provide acceptable services even in the occurrence of failures. Common faults in these systems are transient and permanent failures in the physical components. The two general techniques to tolerate computational and communication errors are spatial or temporal redundancy. The first technique repeats the same operation on different components, while the second one considers the same component for the replicated operation but using different time intervals [79].

To ensure reliability in TTSoSs, we apply the temporal and spatial redundancy techniques in the elements of SoS application models (i.e., services, jobs and messages) and develop a fault-tolerant scheduling model, which considers the system reliability based on the redundancy to deliver dependable services in the presence of faults. This scheduling model is defined in two levels of SoS and CS and solved by a GA optimizer. In both levels, the spatial redundancy technique are applied for the time-triggered messages, i.e., they are copied and transmitted through different links. Redundancy will also happen in executing services and jobs. In the following sections, we explain the applied fault-tolerant techniques in the scheduling model using a GA solver.

## 6.1 Fault Model

To develop a fault-tolerant scheduling model for the TTSoS, first we need to detect the possible faults in this system. Regarding to the SoS physical architecture, faults can occur in each constituent system as well as each network domain. The fault model considers the high probable transient and permanent faults which affect physical nodes, e.g., end-systems or switches, and the links between them. It is assumed that each end-system, switch and link can fail independently with a constant failure rate and that they are all fail silent. Therefore, we introduced them separately in different levels of SoS and CS.

- SoS-level fault model

  From the high level point of view, faults can happen in the switches and links within the network domains of an SoS and also the links connecting them to the constituent systems. These faults affect the reliability of communication between services by SoS messages.

- CS-level fault model

  The probable faults in this level occur in the end-systems, switches and physical links of each constituent system. These faults influence the reliability of providing services including the reliability of executing jobs and communication between them by their related messages.

## 6.2 Fault-tolerant Scheduler

The scheduling problem is to run SoS applications on TTSoS networks with an acceptable reliability rate. The reliability of an SoS is expressed in terms of the probabilities of success in execution of all services. Respectively, reliability of each service is computed by the probabilities of success in executing its jobs and transmitting the related messages. Therefore, we implement the fault-tolerant techniques in the schedulers from both SoS and CS levels. To ensure the reliability of our network, we apply both the temporal and spatial redundancy methods as fault-tolerant techniques in our schedulers.

### 6.2.1 SoS-level programming

In the SoS-level scheduling model, the reliability is achieved by means of two fault-tolerant techniques: (1) service replication, (2) path redundancy between the constituent systems. Based on the SoS application model, a set of services are selected to repeatedly computed on different constituent systems. The cloned service inherits all the processing requirements such as data dependency with other services, and deadline from the original service.

Figure 6.1 shows an example of an SoS application with 4 services, where service $S1$ is replicated, besides the two linked SoS-messages $bm0$ and $bm1$.

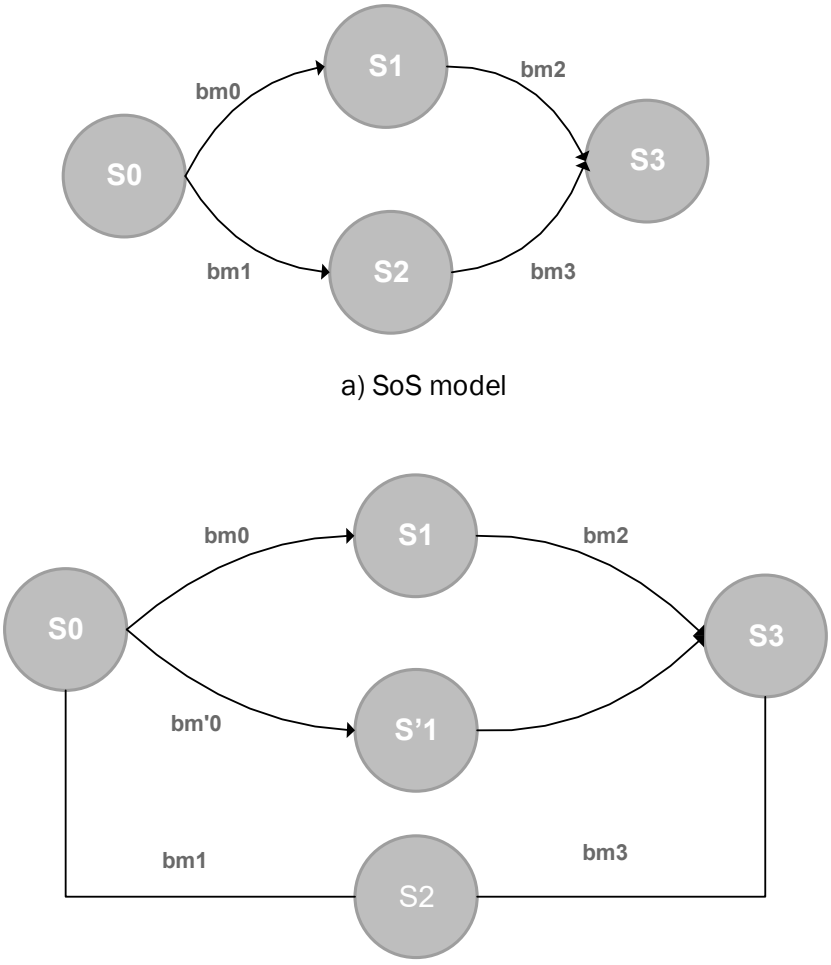


Figure 6.1: Service replication in a SoS model

The second fault-tolerant method to reach a higher reliability in this level is based on the path redundancy technique. Assuming there are multiple connecting paths among constituent systems in the SoS topology, this technique transmits the copied of SoS-messages through different links. Considering an example of an SoS with 3 constituent systems and 2 network domains in Figure 6.2, we assumed that each constituent system is connected with both network domains. Therefore, between each two constituent systems, there is at least two paths through each network domain. That enables fault-tolerant computing of operations related to the high-level SoS. Moreover, the physical model of constituent systems consist of end-systems and switches with bi-directional links. The Time-Triggered

Protocol (TTP) is assumed as the communication infrastructure within and among the constituent systems.

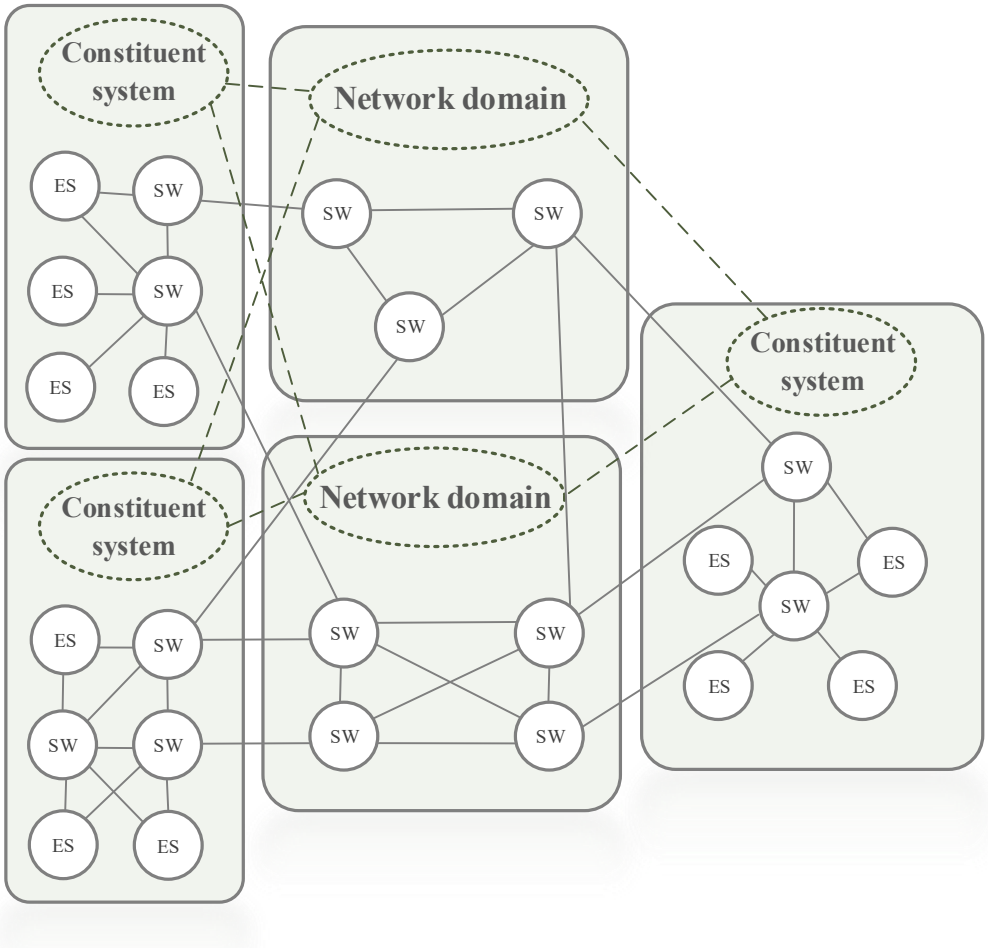


Figure 6.2: Physical model of an SoS with 3 constituent systems and 2 network domains

## 6.2.2 CS-level programming

Similarly in the CS level, we apply two fault-tolerant techniques: (1) job replication and (2) path redundancy for the messages. We show the reliability calculation in this level with an example. Considering a simple service DAG with 3 jobs and 2 messages from an SoS application (see Figure 6.3) to be scheduled on a constituent system of an SoS using TTP. Figure 6.4 shows the network topology of this constituent system with 5 end-Systems denoted by $es$, 4 switches ($sw$) and 10 links ($l$). Table 6.1 shows one of the schedules derived from the local GA scheduler. In this schedule, jobs $j0$, $j1$ and $j2$ are allocated to $es0$, $es3$ and $es2$, respectively.

As mentioned in the chapter 2, the reliability of a system $R(t)$ is calculated in terms of a failure-rate function with constant failure rate $\lambda$ as follows,

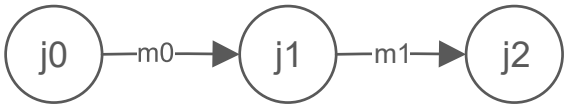


Figure 6.3: Example of DAG for one service of the SoS application

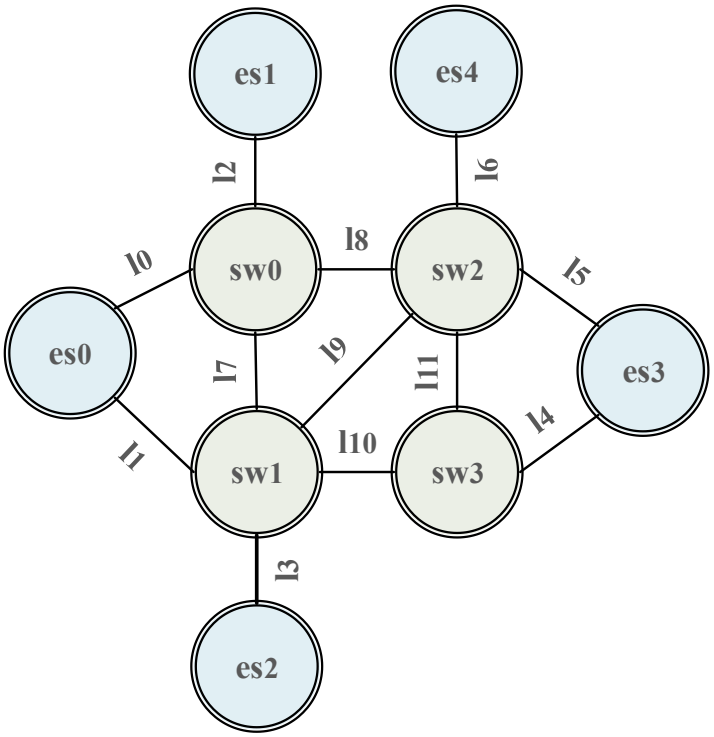


Figure 6.4: The topology of a constituent system

$$R(t) = e^{-\lambda t} \tag{6.1}$$

Based on the equation 6.1, the probability of success in providing the following schedule for the system shown in Figure 6.5 depends on the reliability of all components, i.e., end-systems and switches (assuming here that all the links are fault-free) and is calculated as follows,

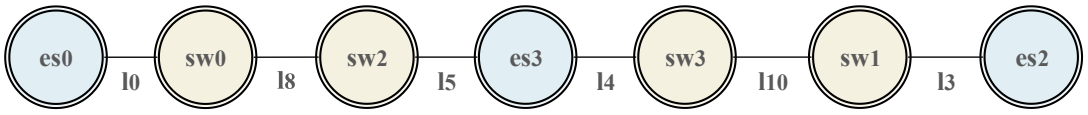


Figure 6.5: The example of a system scheduler

| **Job id** | **start time** | **Runs on** |
|---|---|---|
| 0 | 0 | $es0$ |
| 1 | 80 | $es3$ |
| 2 | 160 | $es2$ |
| **Msg. id** | **injection time** | **Route** |
| 0 | 20 | $es0 - l0 - sw0 - l8 - sw2 - l5 - es3$ |
| 1 | 100 | $es3 - l4 - sw3 - l10 - sw1 - l3 - es2$ |

Table 6.1: CS-level schedule for the example service

$$R_{system} = R_{es0} \cdot R_{sw0} \cdot R_{sw2} \cdot R_{es3} \cdot R_{sw3} \cdot R_{sw1} \cdot R_{sw1} \cdot R_{sw2} \tag{6.2}$$

Regarding the equation 6.2, the reliability of this system scheduler will be less that the reliability of each component. Therefore, we implement the redundancy techniques to improve the reliability of local schedulers inside each constituent system. Implementing the path redundancy technique, the scheduler will replicate the messages and transmit them through different routes. In accordance with requirements of the path redundancy technique, we need at least two routes between each two end-systems in each constituent system. The table of routes are stored in advance and be given to the schedulers. Consequently, we apply the $k$-shortest path routing to find all possible routes in the CS network as well as the SoS routes (i.e, routes between constituent systems and network domains). Considering the network in Figure 6.4, we provide the results of routing algorithm in the Table 6.2.

| **Start → End** | **Number** | **Route** |
|---|---|---|
| es0 → es1 | 1 | R0: l0 sw0 l2 |
| es0 → es2 | 2 | R0: l1 sw1 l3<br>R1: l0 sw0 l7 sw1 l3 |
| es0 → es3 | 2 | R0: l0 sw0 l4 sw2 l5<br>R1: l0 sw1 l10 sw3 l4 |
| es0 → es4 | 2 | R0: l0 sw0 l8 sw2 l6<br>R1: l1 sw1 l9 sw2 l6 |

Table 6.2: 2-shortest paths between $es0$ and other end-systems

As Figure 6.6 shows, $m0$ and its replicated are traversed from $es0$ to $es3$, as well as $m1$ which is sent from $es3$ to $es2$ in parallel with its replicated. To provide a reliable service, at least one of two messages (original or copy) should be received correctly to the destination. In this schedule, the resource $sw1$ is used by the original and the copy of $m1$ and $m0$. Due to this shared unit, this system is considered as neither parallel nor series. In this case, the

reliability of system will be calculated by considering whether this shared unit working or not (see equation 6.3). As a result, the reliability of this system will be as follows,

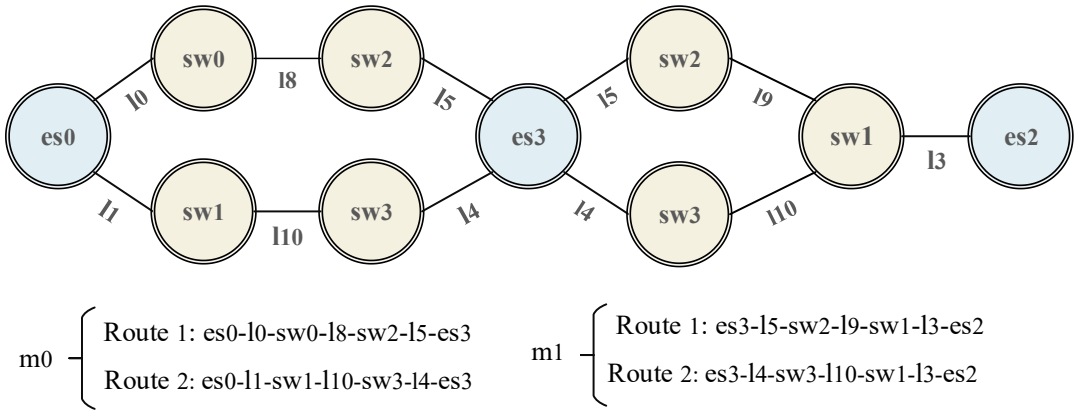


Figure 6.6: Example of a system scheduler considering path redundancy

$$\begin{aligned} R_{system} = R_{shared\ unit} \cdot Prob\{system\ works | shared\ unit\ is\ fault\ free\} + \\ (1 - R_{shared\ unit}) \cdot Prob\{system\ works | shared\ unit\ is\ faulty\} \end{aligned} \tag{6.3}$$

$$\begin{aligned} R_{system} &= R_{sw1} \cdot Prob\{system\ works | sw1\ is\ fault\ free\} + \\ &\quad (1 - R_{sw1}) \cdot Prob\{system\ works | sw1\ is\ faulty\} \\ &= R_{sw1} \cdot R_{es0} \cdot (1 - (1 - R_{sw0} \cdot R_{sw2}) \cdot (1 - R_{sw3})) \cdot R_{es3} \cdot R_{es2} \end{aligned} \tag{6.4}$$

Moreover, we replicate a set of jobs and allocate them to different end-systems. Along with replicating jobs, their related messages must be replicated as well. Figure 6.7 shows the updated DAG of the service after replicating job $j1$ and its messages $m0$ and $m1$.

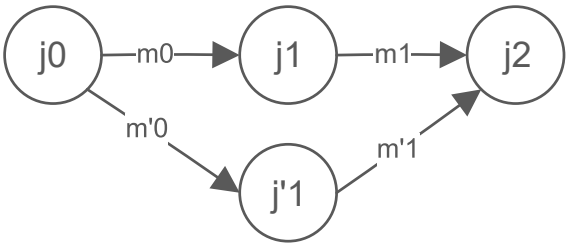


Figure 6.7: Example of a service DAG with one replicated job

Table 6.3 shows the schedule result for the service considering job replication. Although the makespan is getting worse due to the traversing longer paths by the replicated messages,

the reliability of system will improve. The reliability of scheduler in Figure 6.8 is shown in equation 6.5.

| **Job id** | **start time** | **Runs on** |
|---|---|---|
| 0 | 0 | $es0$ |
| 1 | 80 | $es3$ |
| $1'$ | 80 | $es4$ |
| 2 | 180 | $es2$ |
| **Msg. id** | **injection time** | **Route** |
| 0 | 20 | $es0 - l0 - sw0 - l8 - sw2 - l5 - es3$ |
| $0'$ | 20 | $es0 - l1 - sw1 - l9 - sw2 - l6 - es4$ |
| 1 | 100 | $es3 - l4 - sw3 - l10 - sw1 - l3 - es2$ |
| $1'$ | 100 | $es4 - l6 - sw2 - l8 - sw0 - l7 - sw1 - l3 - es2$ |

Table 6.3: CS-level schedule for the service with job replication

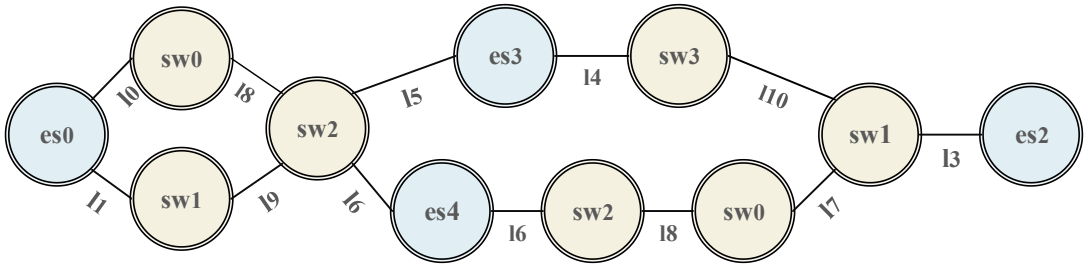


Figure 6.8: The example of a system scheduler

$$R_{system} = R_{sw1} \cdot R_{sw2} \cdot R_{es0} \cdot R_{sw0} \cdot (1 - (1 - R_{es3} \cdot R_{sw3}) \cdot (1 - R_{es4})) \cdot R_{es2} \tag{6.5}$$

## 6.3 GA implementation

To optimize the fault-tolerant schedulers in both CS and SoS levels, we extend the two-level interactive GA discussed in the previous chapter. From the SoS level, reliability constraints as well as temporal constraints for each service will be sent to the candidate constituent system to show the expectation of the whole system about the reliability of each service.

Selecting which services or jobs need to be replicated is also a part of the GA decision-making process. Therefore, theses information are defined in the SoS-level genome. As Figure 6.9 shows, the first gene (G0) allocates services to their compatible constituent systems. The next one (G1) assigns time budgets to services. Gene G2 decides which

service should be replicated and G3 is defined for each message to store the available paths. The last gene stores different reliability goals for each service.

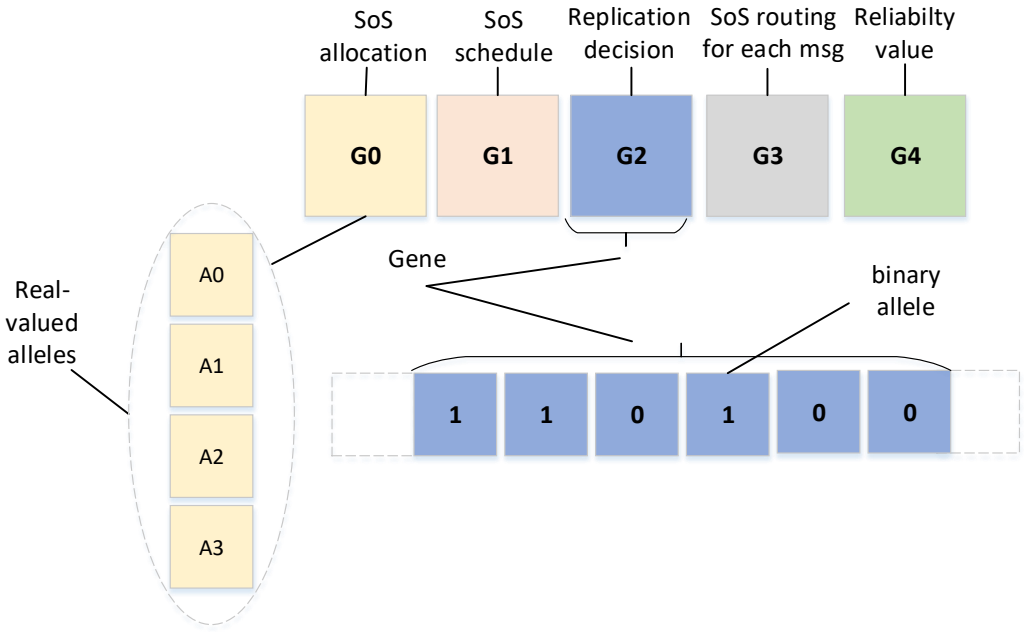


Figure 6.9: SoS-level Genome

After sending the constraints to the compatible constituent systems by the SoS-level scheduler, the CS-level scheduler is called for each service of the SoS application regarding to each solution. The candidate CS must determine a local schedule to provide the service of the respective CS given the time interval from the SoS schedule.

The CS-level scheduler optimizes the scheduling problem of each service with respect to its DAG with jobs and messages. The schedule comprises the temporal and spatial resource allocation such as the allocation of jobs to end systems, the allocation of messages to paths between end-systems, the timing of job executions and the timing of messages. The fitness value of the scheduler is a multi-objective function of maximizing the reliability and minimizing the makespan with respect to the deadline of each service. The scheduler returns the best solution to the SoS-level scheduler upon the termination of the optimization.

The SoS-level scheduler collects the best schedules and the fitness values of the services and calculates the reliability of the whole system and determines a global fitness for each solution from $P$ which is also a multi-objective function of maximizing the reliability and minimizing the makespan. The scheduler generates new solutions using mutation and cross-over and sends them to the CS-level scheduler. This iterative process is stopped until the convergence happens. The pseudo code in Algorithm 4 explains the process of the two-level interactive fault-tolerant GA scheduler.

**Algorithm 4:** Two-level fault-tolerant GA Scheduler

**input:** new SoS Application and deadline $< A, d >$
DAG $A = \{< S, M >\}$: Services $S$(vertices), SoS-messages $M$(edges),
SoS network topology undirected graph $G = \{< V, L >\}$:
Constituent systems ( $CS$) and Network domains ( $ND$):
$CS \cup ND \in V$, Links $L$
Identifiers of compatible $CS_j$ for the service $s_i \in S : comp : S_i \rightarrow \{CS_j\}$,
Failure rates of nodes and links $N_i$, $L_i$,
Number of initial solutions $I \in \mathbb{N}$,
GA Parameters (number of generations, population size, ...)
Table of Routes $R$,

**begin**
Run SoS-level scheduler:
Initial population $P = \{So_1, So_2, \ldots So_{npop}\}$
$Solution\ So_j =< schedule_s, schedule_m >$
$schedule_s : s_i \rightarrow < CS_i,\ EarliestStartTime,\ Deadline,\ Reliability >$,
$CS_i \in comp(s_i), EarliestStartTime,\ Deadline,\ Reliability \in \mathbb{N}$
**repeat**
$\forall\ So \in P$ :
$\forall\ schedule_s \in So$ :
Run local GA scheduler in the allocated constituent system:
$Lateness(s_i) = \max(0, Makespan(s_i) - Deadline(s_i))$,
Calculate the reliability of system for each schedule,
Fitness(S) $= w0 * Reliability + w1 * Lateness(s_i)$ (w0 , w1$\in \mathbb{N}$)
Return the best CS-level schedule with Max fitness and lateness $\leq 0$ ;
SoS-level fitness function:
$Fitness(So) = w0 * Reliability(So) + w1 * Lateness(So)$ (w0 , w1$\in \mathbb{N}$)
mutation/crossover of solutions in $P$,
**until** $Lateness(So) \leq 0$;

# 7 Evaluation and Results

In this chapter, we conduct a series of experiments to evaluate the heuristic scheduling algorithms with randomly generated scenarios at different scales. The first set of our results is dedicated to tuning of the parameters of the GA as well as studying the impact of varying the problem-specific parameters, e.g., number of services and constituent systems, on the convergence of our algorithm. Afterward, we evaluate the capability and efficiency of our GA-based scheduler compared to GLS which is a baseline approach for scheduling real-time traffic in communication networks in the state of the art [80]. Accordingly, we conduct experiments to test the schedulability and transmission makespan of both schedulers. For this purpose, we generate SoS models with different number of constituent systems with meshed grid structures along with different scales of TTSoS applications. The third set of our experiments is intended to evaluate the efficiency of our heuristic approach, namely MBT-aware GA-based scheduling algorithm, to schedule incrementally added applications in the SoS networks. In this regard, we compare the schedulability and transmission makespan of our incremental scheduler for different time-triggered SoS applications with the results from the non-MBT-aware GA-based scheduling algorithm. The last set of our results is dedicated to our designed fault-tolerant scheduling approach for the time-triggered SoS applications. Accordingly, we test the reliability of our scheduler respecting the timeliness feasibility with different scales of examples.

## 7.1 Parameter setting

This section deals with the impact of varying the critical parameters of our model on the convergence of our algorithm. We can distinguish two types of parameters: GA-related and SoS-related parameters. Firstly, we set the optimized values for the common parameters of the GA, i.e., the population size, the number of generations, probability of calling mutation and crossover operators. In the next subsection, we examine the effects of other parameters, e.g., number of constituent systems and application size on the convergence of our scheduling algorithm.

### 7.1.1 GA Parameter Setting

We examine the convergence behavior of the CS-level GA-based scheduler within an acceptable period of time. The GA is used with the following parameter values: uniform selection of crossover with the rate of 0.3, uniform mutation with the rate of 0.5, and four different population sizes (10, 25, 50 and 100). With each population size and other fixed parameter values, the GA is run until observing the convergence. Figure 7.1 shows the convergence of our GA around 500 generations. The fitness value in this figure means the objective function value which is the minimizing the total makespan. At the end of 500 generations, the populations ranking from best to worst is 100, 50, 25 and 10. With the population size of 10 and 25, the GA converges without reaching a near-optimal solution.

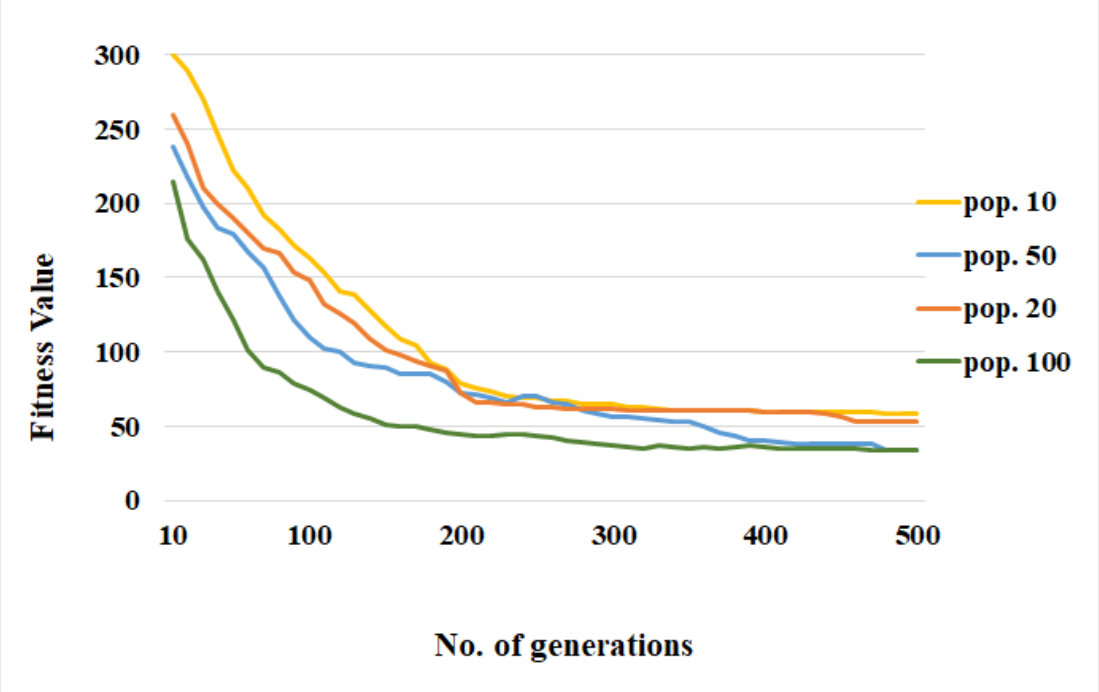


Figure 7.1: GA progress toward the optimal solution

Table 7.1 shows the tuned GA parameters' values at both CS and SoS levels, which result in good results.

### 7.1.2 Customized Parameter setting

According to the above-mentioned GA parameters, we examine the convergence of the SoS-level GA for different SoS sizes. Figure 7.2 demonstrates that the scheduler will converge faster in small-scaled SoS.

Table 7.1: GA parameters

| Parameter | Value |
|---|---|
| **CS-level GA scheduler** | |
| Number of population | 100 |
| Number of generation | 500 |
| Mutation selection rate | 0.3 |
| Crossover selection rate | 0.5 |
| **SoS-level GA scheduler** | |
| Number of population | 50 |
| Number of generation | 100 |
| Mutation selection rate | 0.3 |
| Number of initial solutions | 20 |

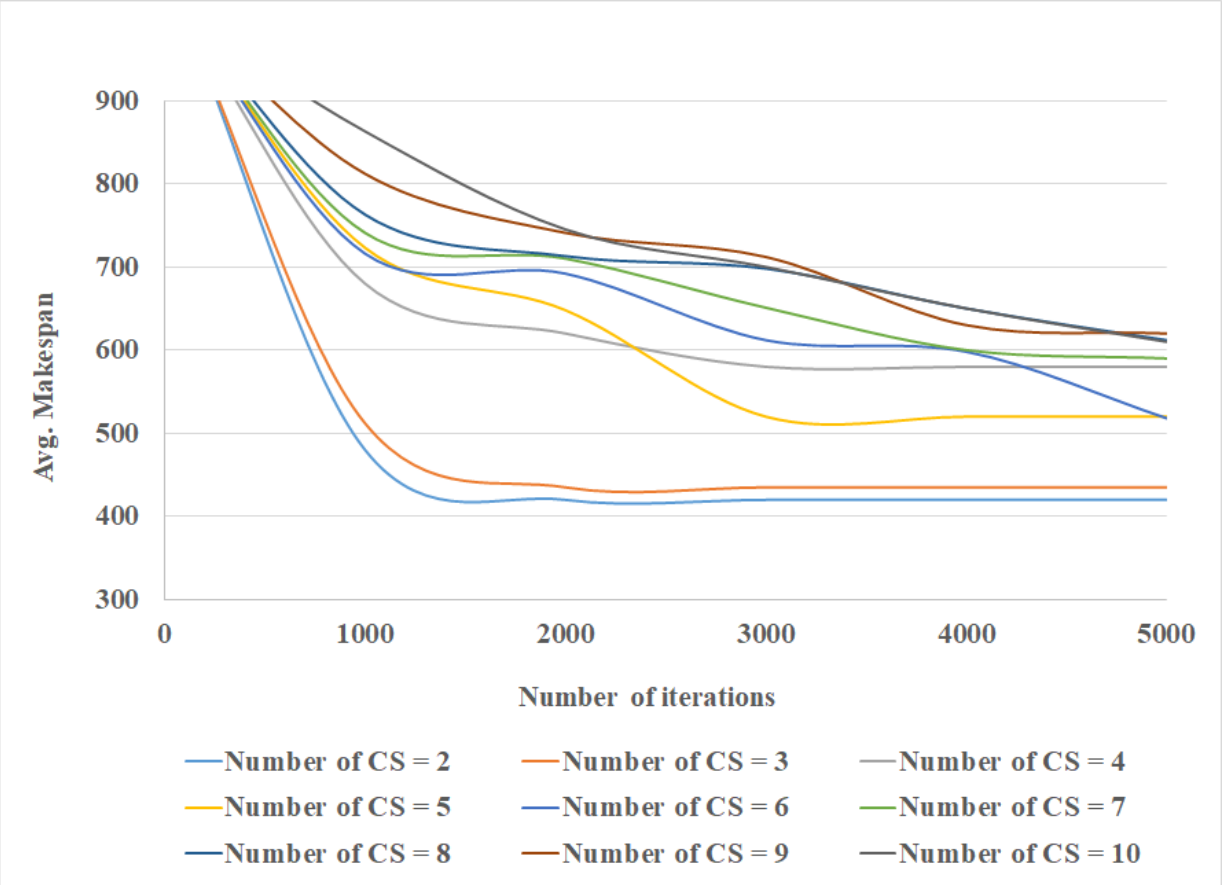


Figure 7.2: Convergence of GA for different network sizes

## 7.2 Scenario Generation

The SoS scenarios are generated based on the Stanford Network Analysis Package (SNAP) library [81]. Each scenario includes the information about the network topology of an SoS and its related application models. One of the network topologies from these scenarios is shown in Figure 7.3. This network includes 4 constituent systems denoted by CS which are connected with each other through 3 network domains denoted by ND. The network topology of these constituent systems is assumed to be a meshed grid, in which every switch is connected to one or more end-systems in a star structure. In this work, we deal with the collaborative SoS, which means there is no central management and the access to information is possible with the permission from the constituent system managements. Moreover, the internal structure of each subsystem is hidden from the other partners.

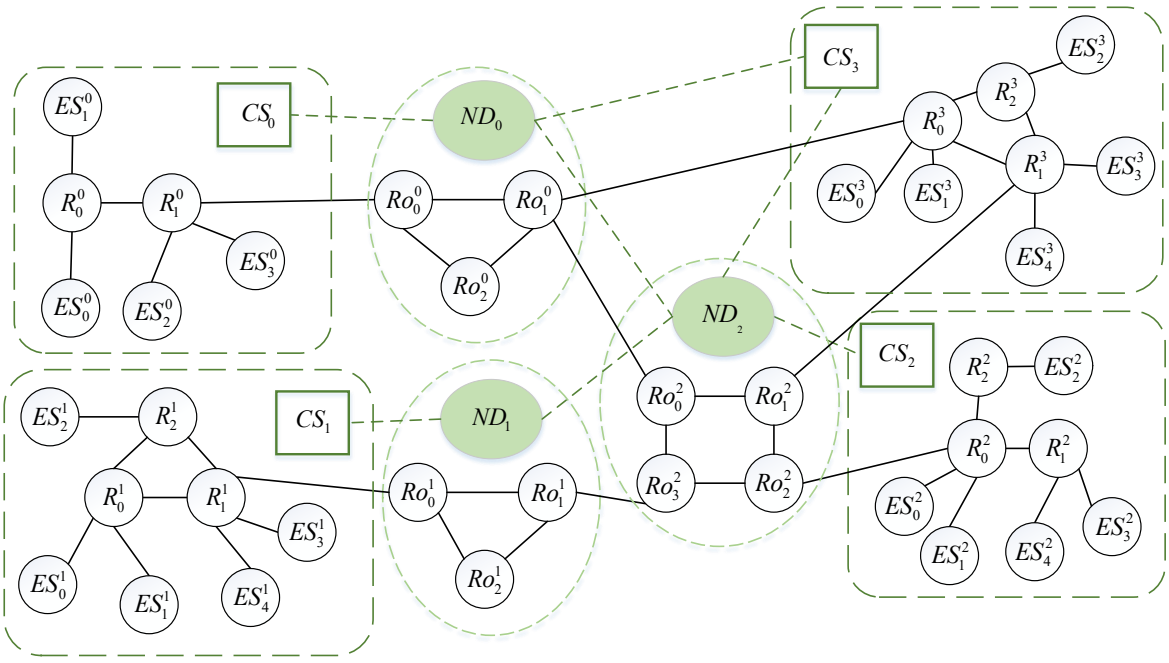


Figure 7.3: An example of an SoS network

## 7.3 The Base Scheduling Heuristic Results

In this section, we compare the possible schedules obtained from the our two-level GA-based scheduler with the GLS-based ones. Figure 7.4 shows one of the SoS-level schedules for the application model with 5 services and 6 messages. This schedule includes the information about mapping each service to a compatible constituent system and the assigned time budgets for executing the services and the SoS-messages. As the figure shows, the

*Service*0 and *Service*4 are assigned to the CS1 with different time budgets (TB). The execution time interval (TI) shows the earliest possible start time and the last finish time of execution for each service on their assigned constituent system. Table 7.2 presents the details of the final schedule for this example. It reports the injection time of each SoS-message and the shortest route between the constituent systems through the network domains as well as the allocated constituent system to each service, the start time and the average makespan for providing each service. For example, the services 0, 1, 2, 3, and 4 are allocated on constituent systems 1, 3, 2, 0, and 1, respectively. In this experiment, there is assumed a constant maximum transmission time for the messages dedicated to the links within each constituent system and to the SoS links between constituent systems. The comparison between the overall completion time of the application from theses schedulers shows the superiority of GA.

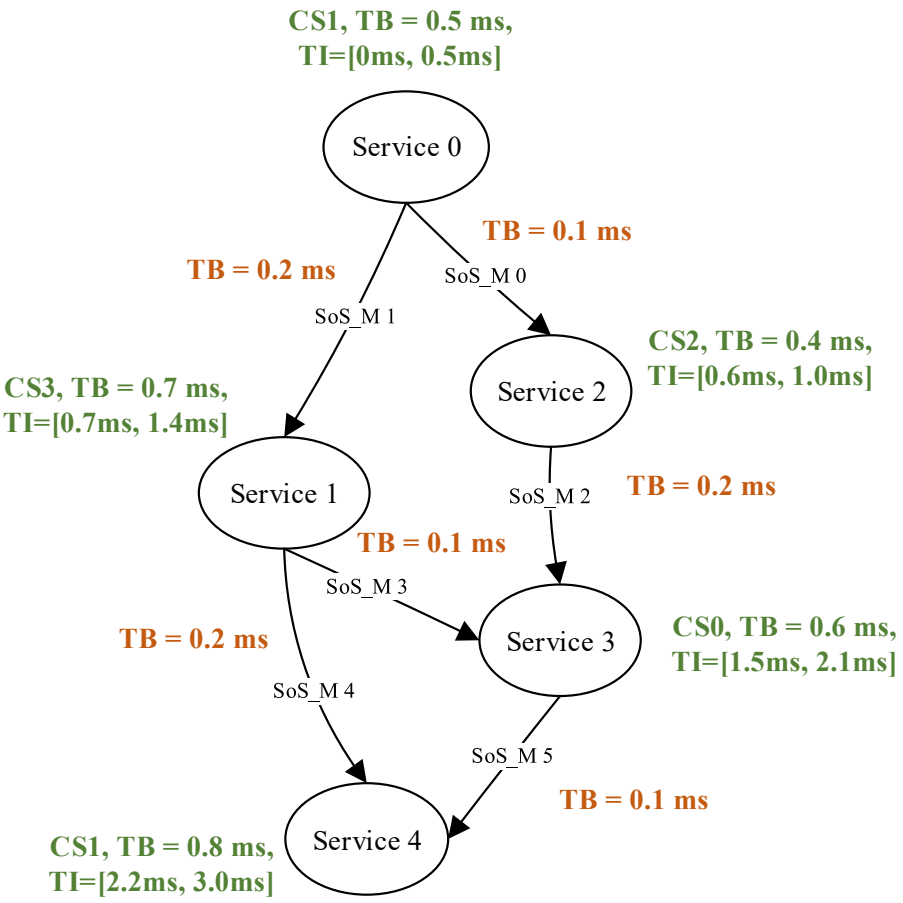


Figure 7.4: An example of an SoS application

Table 7.2: Experimental results comparing the performance of GA and GLS scheduling approaches

| **Two-level GA-based scheduler:** | | | |
|---|---|---|---|
| **Msg.** | **Injection time** | **Route** | |
| 0 | 200 | $ES_4^1, R_1^1, Ro_0^1, Ro_1^1, Ro_3^2, Ro_2^2, R_0^2, ES_0^2$ | |
| 1 | 220 | $ES_3^1, R_1^1, Ro_0^1, Ro_1^1, Ro_3^2, Ro_2^2, Ro_1^2, R_1^3, ES_4^3$ | |
| 2 | 840 | $ES_2^2, R_2^2, R_0^2, Ro_2^2, Ro_1^2, Ro_0^2, Ro_1^0, Ro_0^0, R_1^0, ES_2^0$ | |
| 3 | 680 | $ES_0^3, R_0^3, Ro_1^0, Ro_0^0, R_1^0, ES_3^0$ | |
| 4 | 660 | $ES_4^3, R_1^3, Ro_1^2, Ro_0^2, Ro_3^2, Ro_1^1, Ro_0^1, R_1^1, ES_3^1$ | |
| 5 | 1320 | $ES_2^0, R_1^0, Ro_0^0, Ro_1^0, Ro_0^2, Ro_3^2, Ro_1^1, Ro_0^1, R_1^1, ES_4^1$ | |
| **Service id** | **Allocated CS** | **Start Time** | **Avg. local makespan ($\mu$s)** |
| 0 | 1 | 0 | 200 |
| 1 | 3 | 500 | 179.83 |
| 2 | 2 | 520 | 319,7 |
| 3 | 0 | 1160 | 159.91 |
| 4 | 1 | 1640 | 200 |
| **Total makespan by GA = 1840 ($\mu$s)** | | | |
| **Two-level GLS-based scheduler:** | | | |
| **Service id** | **Allocated CS** | **Start Time** | **Avg. local makespan ($\mu$s)** |
| 0 | 0 | 0 | 180 |
| 1 | 2 | 510 | 198.4 |
| 2 | 1 | 500 | 340.8 |
| 3 | 3 | 1235 | 138 |
| 4 | 0 | 1732 | 200 |
| **Total makespan by GLS = 1932 ($\mu$s)** | | | |

Moreover, we generated 10 different types of scenarios with the SNAP library to analyze the schedulability of our schedulers. Table 7.3 shows the configuration of these scenarios. The SoS includes different SoS applications with the number of services ranging from 4 to 7 running on different networks. The SoS size denotes the total number of constituent systems and network domains and the CS size refers to the average number of nodes in the constituent systems (i.e., including end-systems and switches). The fourth column in this table refers to the size of the SoS applications including the number of services and SoS-messages and the service size refers to the average number of jobs in each service.

Table 7.4 compares the required time for providing all the SoS services from these two schedulers. The results show that GA improves the makespan on average by 19% in

comparison to GLS. The GA results in a better resource utilization and load balancing in both global and local schedules. Table 7.5 lists the average execution time of GA and GLS for each scenario. As the results show, the GLS solves the scheduling problem faster than the GA scheduler, because of employing a faster search strategy. However, the chance of getting stuck in a local minimum for this scheduler is more than the GA. Figure 7.5 shows the average improvement in the makespan for different siye of problems.

Table 7.3: SoS Scenario configuration

| **Scenario id** | **SoS size** | **CS size** | **App. size** | **Service size** |
|---|---|---|---|---|
| 1 | 7 | 6 | 4 | 4 |
| 2 | 7 | 8 | 5 | 6 |
| 3 | 7 | 10 | 6 | 8 |
| 4 | 8 | 6 | 4 | 4 |
| 5 | 8 | 8 | 5 | 6 |
| 6 | 8 | 12 | 6 | 8 |
| 7 | 9 | 6 | 4 | 4 |
| 8 | 9 | 8 | 5 | 6 |
| 9 | 9 | 9 | 6 | 8 |
| 10 | 9 | 12 | 7 | 9 |

Table 7.4: Comparing the average transmission makespan from GLS and GA schedulers

| **SoS Scenario** | **GA Avg. makespan** ($ms$) | **GLS Avg. makespan** ($ms$) | **Improvement ratio** |
|---|---|---|---|
| 1 | 381 | 418 | 0.09 |
| 2 | 414 | 470 | 0.12 |
| 3 | 458 | 572 | 0.19 |
| 4 | 523 | 637 | 0.18 |
| 5 | 580 | 753 | 0.23 |
| 6 | 599 | 798 | 0.25 |
| 7 | 618 | 846 | 0.27 |
| 8 | 632 | 902 | 0.30 |
| 9 | 683 | 962 | 0.29 |
| 10 | 728 | 1070 | 0.32 |

Table 7.5: Comparing Execution time of GA and GLS scheduling algorithm for different scenarios

| SoS Scenario | GA Avg. Exec Time (min) | GLS Avg. Exec Time (min) |
|---|---|---|
| 1 | 0.16 | 0.005 |
| 2 | 0.58 | 0.0902 |
| 3 | 0.95 | 0.156 |
| 4 | 1.54 | 0.234 |
| 5 | 2.3 | 0.256 |
| 6 | 3.7 | 0.343 |
| 7 | 4.5 | 0.210 |
| 8 | 5.9 | 0.214 |
| 9 | 8.3 | 1.04 |
| 10 | 10.9 | 1.3 |

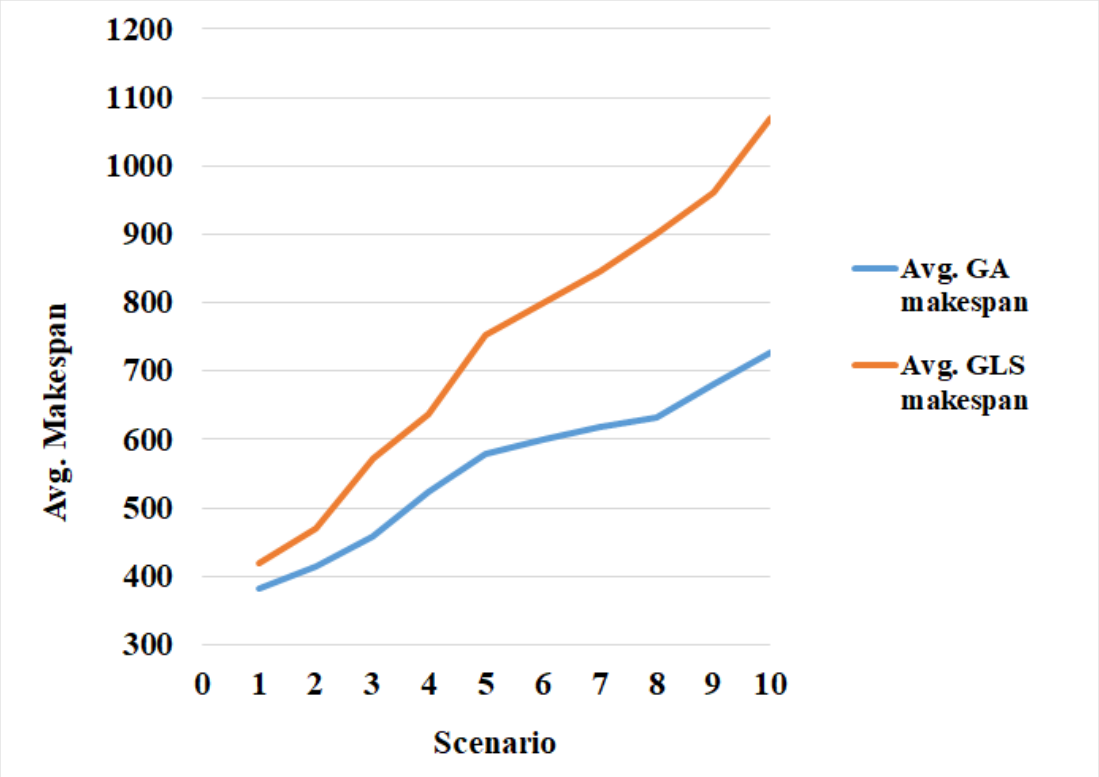


Figure 7.5: Comparing the average makespan from GA and GLS schedulers

## 7.4 Results of Incremental Scheduling Algorithm

In this section, we examine the performance of the proposed MBT-aware incremental scheduling approach in the chapter 5 by running a set of real-time applications on an SoS network. These applications are released incrementally in a chronological order and executed over time. The assumed SoS network includes 4 CSs and 3 NDs. To evaluate the effectiveness of the proposed MBT method in our incremental scheduler, we compare its schedule results from with the ones obtained from the non MBT-aware incremental scheduler. Table 7.6 compares the average makespan of the 10 applications obtained from both incremental schedulers as well as their shortest makespan when all the resources are free. As can be seen in the results, our new incremental scheduling algorithm outperforms the conventional one. Although the conventional scheduler found better schedules (shorter makespan) for the first and second applications, its performance was getting worse by releasing more applications and it was not able to meet the deadline of the applications after finish executing the 5th application (The deadline of each application is assumed 2 $ms$). The new scheduler can cope better with the upcoming applications.

Table 7.6: Comparing the makespan of the SoS applications from both incremental schedulers

| **App. id** | **Release time** (*sec.*) | **non MBT-aware scheduler Avg. makespan** ($\mu s$) | **MBT-aware scheduler Avg. makespan** ($\mu s$) |
|---|---|---|---|
| 1 | 0 | 350 | 480 |
| 2 | 5 | 454 | 500 |
| 3 | 12 | 1116 | 675 |
| 4 | 19 | 1529 | 930 |
| 5 | 23 | 1730 | 1131 |
| 6 | 30 | 2086 | 1262 |
| 7 | 45 | 2569 | 1436 |
| 8 | 60 | 2953 | 1609 |
| 9 | 75 | 3236 | 1813 |
| 10 | 100 | 3420 | 1955 |

One of the demanding parameters in our new scheduler is $\Delta$, which affects the performance of the allocation approach and consequently the makespan of schedules. We examined the new scheduler by assigning different values to this parameter to find an op-

timized value. Figure 7.6 shows the results from the new scheduler for different values of $\Delta$. As Figure 7.6 shows, by increasing the value of $\Delta$ from 0 to 20, the schedule results enhanced significantly, but increasing this value up to e.g. 25 or 30, has a reverse effect on the performance of our scheduler.

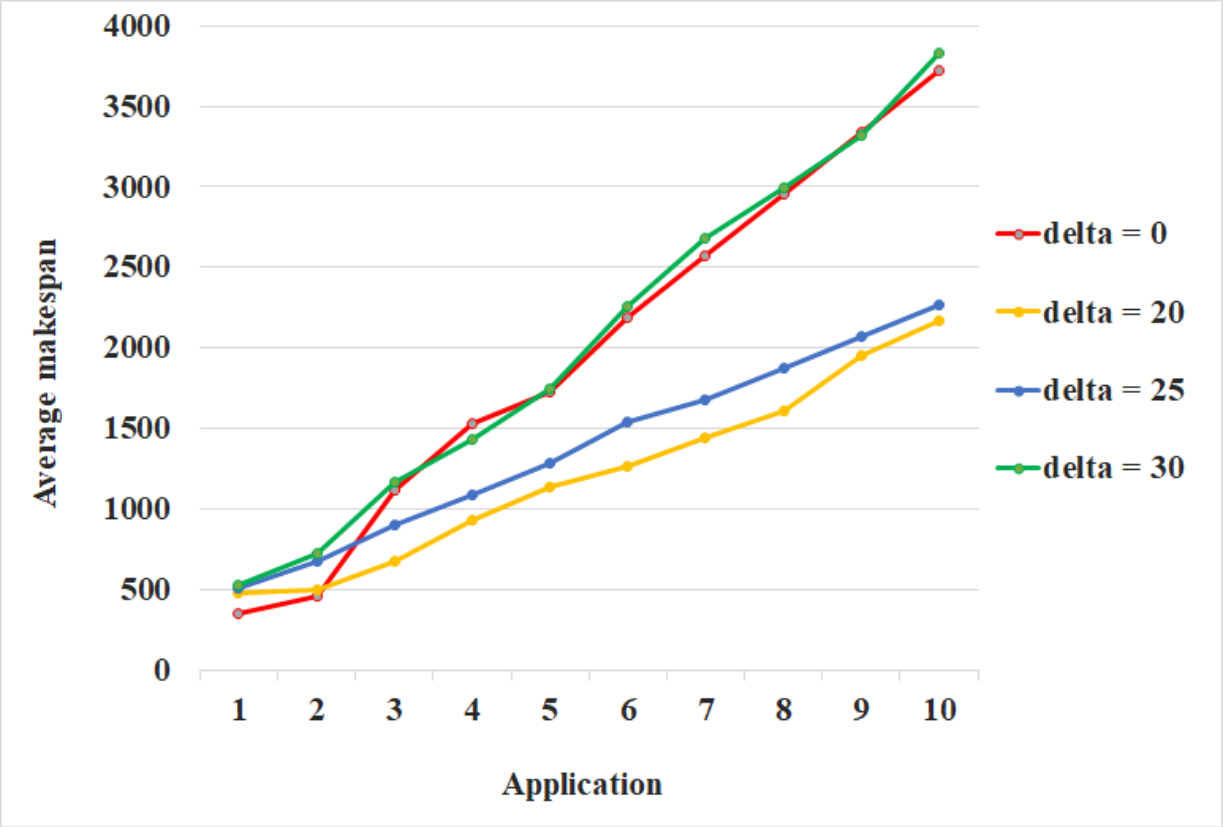


Figure 7.6: Examining the effect of parameter $\Delta$ on performance of the MBT-aware scheduler

Another important parameter in this algorithm is the weight of MBT in the fitness function of the local schedulers ($W0$). The effect of this parameter is studied in the MBT-aware incremental scheduler by testing different values ranging from 0.1 to 1. As Figure 7.7 shows, the best value for the parameter $W0$ is 0.6, which has the highest rank of success. The success ratio is defined as the number of SoS applications found schedulable by the new incremental scheduler divided by the total number of applications.

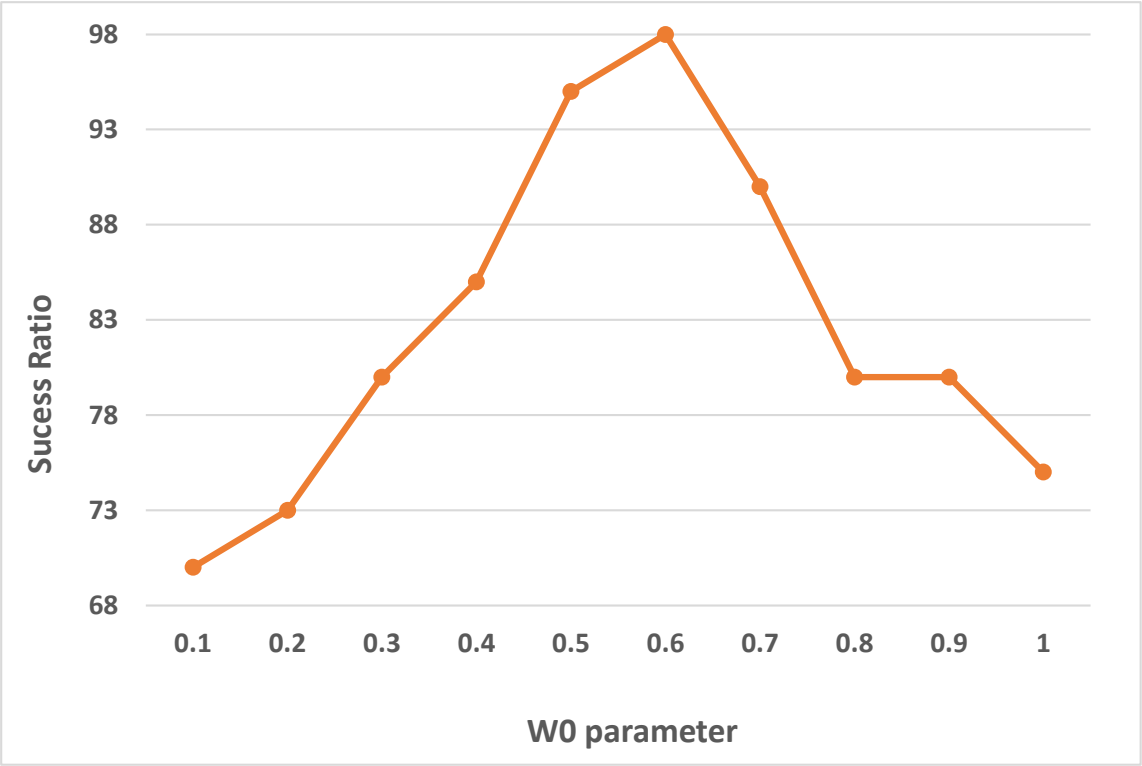


Figure 7.7: Effect of $W0$ parameter

## 7.5 Results of Fault-Tolerant Scheduling Algorithm

Table 7.7 shows different SoS instances used to examine the performance of the fault-tolerant scheduling model. There are some assumptions for the parameters of the fault-tolerant GA scheduler which includes the failure rates for the nodes (end-systems and switches) and links as well as the number of generations and populations (see Table 7.8). Table 7.9 shows the best results of our fault-tolerant scheduler for different scenarios in case of replicating the jobs and services and compares them with the results assuming no replication. The results show the capability of fault-tolerant techniques in our scheduler.

Table 7.7: The generated scenarios

| Scenario | Number of services | Number of SoS msgs |
|---|---|---|
| N01 | 4 | 3 |
| N02 | 4 | 4 |
| N03 | 4 | 5 |
| N04 | 5 | 4 |
| N05 | 5 | 6 |
| N06 | 5 | 7 |
| N07 | 6 | 4 |
| N08 | 6 | 5 |
| N09 | 6 | 6 |
| N10 | 7 | 4 |
| N11 | 7 | 5 |
| N12 | 7 | 6 |
| N13 | 8 | 6 |
| N14 | 8 | 7 |
| N15 | 8 | 8 |
| N16 | 9 | 4 |
| N17 | 9 | 5 |
| N18 | 9 | 6 |
| N19 | 10 | 6 |
| N20 | 10 | 7 |

Table 7.8: Assumptions of the fault-tolerant scheduling algorithm

| Avg. failure rate of nodes | Avg. failure rate of links | Gen. | Pop. size |
|---|---|---|---|
| 1.00E-06 | 1.00E-07 | 50 | 20 |

Table 7.9: Reliability of system scheduler for different scenarios with and without replicating jobs

| Scenario | makespan | Reliability with replication | Reliability without replication |
|---|---|---|---|
| N01 | 2967 | 0.9978 | 0.969 |
| N02 | 2441 | 0.989 | 0.985 |
| N03 | 2229 | 0.993 | 0.987 |
| N04 | 3573 | 0.998 | 0.929 |
| N05 | 2680 | 0.991 | 0.921 |
| N06 | 3235 | 0.979 | 0.9887 |
| N07 | 3089 | 0.994 | 0.897 |
| N08 | 3021 | 0.993 | 0.9083 |
| N09 | 4210 | 0.997 | 0.9089 |
| N10 | 3405 | 0.9791 | 0.9438 |
| N11 | 5601 | 0.9714 | 0.898 |
| N12 | 6123 | 0.9329 | 0.8973 |
| N13 | 6284 | 0.9869 | 0.8897 |
| N14 | 6134 | 0.9974 | 0.9866 |
| N15 | 7032 | 0.923 | 0.9370 |
| N16 | 7134 | 0.911 | 0.8709 |
| N17 | 6990 | 0.9364 | 0.832 |
| N18 | 7012 | 0.9834 | 0.89 |
| N19 | 7222 | 0.91302 | 0.870 |
| N20 | 7234 | 0.9039 | 0.834 |

# 8 Conclusion

## 8.1 Summary

The increasing importance of SoSs in safety-critical domains requires customized techniques for process optimization and resource allocation to suit the requirements of this type of systems. The challenge is exacerbated in time-aware applications with real-time requirements. This dissertation developed a heuristic approach based on GA to schedule the time-triggered messages-based communication in collaborative SoSs. Since there is no management hierarchy in our SoS, the scheduling algorithm is initiated by one of the involved constituent systems to coordinate the operations of other constitutes. Therefore, the scheduling activities are defined at two levels of SoS and CS, which iteratively interact with each other. The SoS level optimizes the global schedules for the services and their related messages and allocates the common resources among the involved constituent systems, while the CS level refers to independent schedulers inside the constituent systems which provide the local schedules for the jobs and the messages of each service. These local schedules are rated based on their lateness values to meet the global deadlines by the SoS level. Consequently, the global schedules are updated for the better fitness to the local resources.

To investigate the capability of our scheduling algorithm, we made a comparison with the results from a GLS-based scheduling algorithm run on different sets of SoS applications. The results on different examples showed the schedulability of our algorithm and the ability for improving the average transmission makespan up to 32% in comparison to the other scheduler.

Moreover, we extended our heuristic scheduling algorithm to schedule time-triggered applications which are incrementally released and added to the SoS. In order to share the limited resources efficiently between these applications, the resource allocation should be done by focusing not only on the current application requirements but also sparing resources for the upcoming applications. This goal was achieved through developing a heuristic method which creates a balance in utilizing resources and reserve free time slots for the future applications. Examining different sets of examples showed that employing this method is able to improve the schedulability of our GA scheduler up to 50 percent.

Furthermore, we integrated the spatial redundancy technique in our proposed scheduling process and design a fault-tolerant scheduling algorithm for SoS applications to guaran-

tee timely delivery of their operations in the presence of physical (i.e., links and nodes) faults. The implemented model maximizes the reliability of operations within and between the constituent systems by replicating time-triggered messages and considering redundant paths for all the real-time communication messages as well re-executing jobs and services. Testing our fault-tolerant scheduler on 20 scenarios with different network and application scales showed the functionality of our scheduling algorithm in terms of reliability and schedulability.

## 8.2 Future Work

In this work, it is assumed that the scheduling process and resource allocation for each SoS application will be finished before starting the process for the next application. As a future work, scheduling algorithms can be proposed to process concurrently the newly arriving applications. Additionally, we can use machine learning techniques to observe the behavior patterns of an SoS and predict its future states. For example, we can apply it in the incremental scheduling algorithm to attempt predicting the sequence of new arriving applications and to improve the resource allocation.

# Bibliography


[1] C. D. Locke. “Best-effort decision making for real-time scheduling”. PhD thesis. Computer Science Department, CMU, 1986.

[2] Hermann Kopetz. *Real-time systems: design principles for distributed embedded applications*. Springer Science & Business Media, 2011.

[3] Mo Jamshidi. *Systems of systems engineering: principles and applications*. CRC press, 2008.

[4] Charles B Keating and Polinpapilinho F Katina. “Systems of systems engineering: prospects and challenges for the emerging field”. In: *International Journal of System of Systems Engineering* 2.2-3 (2011), pp. 234–256.

[5] Judith S Dahmann and Kristen J Baldwin. “Understanding the current state of US defense systems of systems and the implications for systems engineering”. In: *2008 2nd Annual IEEE Systems Conference*. IEEE. 2008, pp. 1–7.

[6] Yaneer Bar-Yam et al. “The characteristics and emerging behaviors of system of systems”. In: *NECSI: Complex Physical, Biological and Social Systems Project* (2004), pp. 1–16.

[7] James M Parker. “Applying a system of systems approach for improved transportation”. In: *SAPI EN. S. Surveys and Perspectives Integrating Environment and Society* 3.2 (2010).

[8] Roman Obermaisser, Mohammed Abuteir, Ala Khalifeh, and Dhiah el Diehn Abou-Tair. “Systems-of-Systems Framework for Providing Real-Time Patient Monitoring and Care: Challenges and Solutions”. In: *ICTs for Improving Patients Rehabilitation Research Techniques*. Springer Berlin Heidelberg, 2015, pp. 129–142.

[9] Roman Obermaisser and Ayman Murshed. “Incremental, Distributed and Concurrent Scheduling in Systems-of-Systems with Real-Time Requirements”. In: *Proc. of the 13th IEEE International Conference on Dependable, Autonomic and Secure Computing (DASC-2015)*. 2015.

[10] Amos Albert et al. “Comparison of event-triggered and time-triggered concepts with regard to distributed control systems”. In: *Embedded world* 2004 (2004), pp. 235–252.

[11] Lucia Lo Bello and Wilfried Steiner. “A perspective on IEEE time-sensitive networking for industrial communication and automation systems”. In: *Proceedings of the IEEE* 107.6 (2019), pp. 1094–1120.

[12] Wilfried Steiner, Günther Bauer, Brendan Hall, Michael Paulitsch, and Srivatsan Varadarajan. “TTEthernet dataflow concept”. In: *8th IEEE International Symposium on Network Computing and Applications*. IEEE. 2009, pp. 319–322.

[13] Robert Kaiser and Stephan Wagner. “Evolution of the PikeOS microkernel”. In: *First International Workshop on Microkernels for Embedded Systems*. Vol. 50. 2007.

[14] Ethan Grossman et al. “Deterministic networking use cases”. In: *IETF draft* (2018).

[15] Grace Lewis, Ed Morris, Pat Place, Soumya Simanta, Dennis Smith, and Lutz Wrage. “Engineering systems of systems”. In: *2008 2nd Annual IEEE Systems Conference*. IEEE. 2008, pp. 1–6.

[16] Sascha Einspieler, Benjamin Steinwender, and Wilfried Elmenreich. “Integrating time-triggered and event-triggered traffic in a hard real-time system”. In: *2018 IEEE Industrial Cyber-Physical Systems (ICPS)*. IEEE. 2018, pp. 122–128.

[17] H. Kopetz, A. Ademaj, P. Grillinger, and K. Steinhammer. “The time-triggered Ethernet (TTE) design”. In: *Eighth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC’05)*. 2005, pp. 22–33. DOI: 10.1109/ISORC.2005.56.

[18] Michael J Pont. *Patterns for time-triggered embedded systems*. TTE System, Ltd, 2008.

[19] Andrea Bondavalli, Sara Bouchenak, and Hermann Kopetz. *Cyber-physical systems of systems: foundations–a conceptual model and some derivations: the AMADEOS legacy*. Vol. 10099. Springer, 2016.

[20] Amit J Lopes, R Lezama, and Ricardo Pineda. “Model based systems engineering for smart grids as systems of systems”. In: *Procedia Computer Science* 6 (2011), pp. 441–450.

[21] Mark W Maier. “Architecting principles for systems-of-systems”. In: *Systems Engineering: The Journal of the International Council on Systems Engineering* 1.4 (1998), pp. 267–284.

[22] Department of Defense (DoD) USA: *DoD Architecture Framework Version 1.5: Volume I: Definitions and guidelines*. Tech. rep. April 2007.

[23] E Douglas Jensen, C Douglas Locke, and Hideyuki Tokuda. “A time-driven scheduling model for real-time operating systems.” In: *Rtss*. Vol. 85. 1985, pp. 112–122.

[24] Arezou Mohammadi and Selim G Akl. “Scheduling algorithms for real-time systems”. In: *School of Computing Queens University, Tech. Rep* (2005).

[25] S Sarathambekai and K Umamaheswari. “Task scheduling in distributed systems using heap intelligent discrete particle swarm optimization”. In: *Computational Intelligence* 33.4 (2017), pp. 737–770.

[26] Emil Åström. *Task Scheduling in Distributed Systems: Model and prototype*. 2016.

[27] Chafik Arar, Hamoudi Kalla, Salim Kalla, and Hocine Riadh. “A Reliable Fault-Tolerant Scheduling Algorithm for Real Time Embedded Systems”. In: *SAFECOMP 2013-Workshop DECS (ERCIM/EWICS Workshop on Dependable Embedded and Cyber-physical Systems) of the 32nd International Conference on Computer Safety, Reliability and Security*. 2013, NA.

[28] Hermann Kopetz et al. “Distributed fault-tolerant real-time systems: The Mars approach”. In: *IEEE Micro* 9.1 (1989), pp. 25–40.

[29] Hermann Kopetz and Günther Bauer. “The time-triggered architecture”. In: *Proceedings of the IEEE* 91.1 (2003), pp. 112–126.

[30] Vilgot Claesson, Stefan Poledna, and Jan Soderberg. “The XBW model for dependable real-time systems”. In: *Proceedings 1998 International Conference on Parallel and Distributed Systems (Cat. No. 98TB100250)*. IEEE. 1998, pp. 130–138.

[31] Viacheslav Izosimov, Paul Pop, Petru Eles, and Zebo Peng. “Design optimization of time-and cost-constrained fault-tolerant distributed embedded systems”. In: *Design, Automation and Test in Europe*. IEEE. 2005, pp. 864–869.

[32] Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, et al. *Fundamental concepts of dependability*. University of Newcastle upon Tyne, Computing Science, 2001.

[33] C Mani Krishna. “Fault-tolerant scheduling in homogeneous real-time systems”. In: *ACM Computing Surveys (CSUR)* 46.4 (2014), pp. 1–34.

[34] Israel Koren and C Mani Krishna. *Fault-tolerant systems*. Morgan Kaufmann, 2020.

[35] Imad Sanduka. “A modelling framework for systems-of-systems with real-time and reliability requirements”. In: (2015).

[36] Yecheng Zhao and Haibo Zeng. “The concept of Maximal Unschedulable Deadline Assignment for optimization in fixed-priority scheduled real-time systems”. In: *Real-Time Systems* 55.3 (2019), pp. 667–707.

[37] Maryam Pahlevan and Roman Obermaisser. “Genetic Algorithm for Scheduling Time-Triggered Traffic in Time-Sensitive Networks”. In: *2018 IEEE 23rd International Conference on Emerging Technologies and Factory Automation (ETFA)*. Vol. 1. 2018, pp. 337–344. DOI: 10.1109/ETFA.2018.8502515.

[38] Li Bingqian and Wang Yong. “Hybrid-GA based static schedule generation for time-triggered ethernet”. In: *2016 8th IEEE International Conference on Communication Software and Networks (ICCSN)*. IEEE. 2016, pp. 423–427.

[39] Eike Schweissguth, Peter Danielis, Dirk Timmermann, Helge Parzyjegla, and Gero Mühl. "ILP-based joint routing and scheduling for time-triggered networks". In: *Proceedings of the 25th International Conference on Real-Time Networks and Systems*. 2017, pp. 8–17.

[40] Paul Pop, Petru Eles, and Zebo Peng. "Scheduling with optimized communication for time-triggered embedded systems". In: *Proceedings of the Seventh International Workshop on Hardware/Software Codesign (CODES'99)(IEEE Cat. No. 99TH8450)*. IEEE. 1999, pp. 178–182.

[41] Krzysztof Kuchcinski. "Embedded system synthesis by timing constraints solving". In: *Proceedings. Tenth International Symposium on System Synthesis (Cat. No. 97TB100114)*. IEEE. 1997, pp. 50–57.

[42] Shiv Prakash and Alice C Parker. "SOS: Synthesis of application-specific heterogeneous multiprocessor systems". In: *Journal of Parallel and Distributed computing* 16.4 (1992), pp. 338–351.

[43] Arthur L Liestman and Roy H Campbell. "A fault-tolerant scheduling problem". In: *IEEE transactions on software engineering* 11 (1986), pp. 1089–1095.

[44] Mohamed Ould Sass, Maryline Chetto, and Audrey Queudet. "The BGW model for QoS aware scheduling of real-time embedded systems". In: *Proceedings of the 11th ACM international symposium on Mobility management and wireless access*. 2013, pp. 93–100.

[45] Traian Pop, Petru Eles, and Zebo Peng. "Holistic scheduling and analysis of mixed time/event-triggered distributed embedded systems". In: *Proceedings of the tenth international symposium on Hardware/software codesign*. 2002, pp. 187–192.

[46] Manar Qamhieh and Serge Midonnet. "An experimental analysis of DAG scheduling methods in hard real-time multiprocessor systems". In: *Proceedings of the 2014 Conference on Research in Adaptive and Convergent Systems*. 2014, pp. 284–290.

[47] Manar Qamhieh, Laurent George, and Serge Midonnet. "Stretching algorithm for global scheduling of real-time DAG tasks". In: *Real-Time Systems* 55.1 (2019), pp. 32–62.

[48] Georgios L Stavrinides and Helen D Karatza. "Scheduling real-time jobs in distributed systems-simulation and performance analysis". In: (2014).

[49] Andrei Tchernykh, Juan Manuel Ramírez, Arutyun Avetisyan, Nikolai Kuzjurin, Dmitri Grushin, and Sergey Zhuk. "Two level job-scheduling strategies for a computational grid". In: *International Conference on Parallel Processing and Applied Mathematics*. Springer. 2005, pp. 774–781.

[50] Zeeshan Ahmed and Salima Hamma. “Two-level scheduling algorithm for different classes of traffic in WiMAX networks”. In: *2012 International Symposium on Performance Evaluation of Computer & Telecommunication Systems (SPECTS)*. IEEE. 2012, pp. 1–7.

[51] Mirsaeid Hosseini. “A new shuffled genetic-based task scheduling algorithm in heterogeneous distributed systems”. In: *Journal of Advances in Computer Research* 9.4 (2018), pp. 19–36.

[52] Junyan Zhou. “Real-time task scheduling and network device security for complex embedded systems based on deep learning networks”. In: *Microprocessors and Microsystems* 79 (2020), p. 103282.

[53] Alan A Bertossi and Luigi V Mancini. “Scheduling algorithms for fault-tolerance in hard-real-time systems”. In: *Responsive Computing*. Springer, 1994, pp. 3–19.

[54] Alan Burns, Robert Davis, and Sasikumar Punnekkat. “Feasibility analysis of fault-tolerant real-time task sets”. In: *Proceedings of the Eighth Euromicro Workshop on Real-Time Systems*. IEEE. 1996, pp. 29–33.

[55] Ching-Chih Han, Kang G Shin, and Jian Wu. “A fault-tolerant scheduling algorithm for real-time periodic tasks with possible software faults”. In: *IEEE Transactions on computers* 52.3 (2003), pp. 362–372.

[56] Ying Zhang and Krishnendu Chakrabarty. “Energy-aware adaptive checkpointing in embedded real-time systems”. In: *2003 Design, Automation and Test in Europe Conference and Exhibition*. IEEE. 2003, pp. 918–923.

[57] Houssine Chetto and Maryline Chetto. “Some results of the earliest deadline scheduling algorithm”. In: *IEEE Transactions on software engineering* 10 (1989), pp. 1261–1269.

[58] Ayman Murshed. “Scheduling event-triggered and time-triggered applications with optimal reliability and predictability on networked multi-core chips”. In: (2018).

[59] Paul Pop, Petru Eles, Traian Pop, and Zebo Peng. “An approach to incremental design of distributed embedded systems”. In: *Proceedings of the 38th annual Design Automation Conference*. 2001, pp. 450–455.

[60] Paul Pop, Petru Eles, and Zebo Peng. “Flexibility driven scheduling and mapping for distributed real-time systems”. In: *8th International Conference on Real-Time Computing Systems and Applications*. 2002.

[61] Christian Schöler, René Krenz-Bååth, Ayman Murshed, and Roman Obermaisser. “Computing optimal communication schedules for time-triggered networks using an SMT solver”. In: *2016 11th IEEE Symposium on Industrial Embedded Systems (SIES)*. IEEE. 2016, pp. 1–9.

[62] Naresh Ganesh Nayak, Frank Dürr, and Kurt Rothermel. "Incremental flow scheduling and routing in time-sensitive software-defined networks". In: *IEEE Transactions on Industrial Informatics* 14.5 (2017), pp. 2066–2075.

[63] P. Mejia-Alvarez, R. Melhem, and D. Mosse. "An incremental approach to scheduling during overloads in real-time systems". In: *Proceedings 21st IEEE Real-Time Systems Symposium*. 2000, pp. 283–293.

[64] Carey Douglass Locke. "Best-Effort Decision-Making for Real-Time Scheduling". PhD thesis. USA: Carnegie Mellon University, 1986.

[65] Roman Obermaisser and Ayman Murshed. "Incremental, distributed, and concurrent scheduling in systems-of-systems with real-time requirements". In: *2015 IEEE International Conference on Computer and Information Technology; Ubiquitous Computing and Communications; Dependable, Autonomic and Secure Computing; Pervasive Intelligence and Computing*. IEEE. 2015, pp. 1918–1927.

[66] Michael Vierhauser, Rick Rabiser, and Paul Grünbacher. "Requirements monitoring frameworks: A systematic review". In: *Information and Software Technology* 80 (2016), pp. 89–109.

[67] Larry B Rainey, Andreas Tolk, et al. *Modeling and simulation support for system of systems engineering applications*. Wiley Online Library, 2015.

[68] Claus Ballegaard Nielsen, Peter Gorm Larsen, John Fitzgerald, Jim Woodcock, and Jan Peleska. "Systems of systems engineering: basic concepts, model-based techniques, and research directions". In: *ACM Computing Surveys (CSUR)* 48.2 (2015), pp. 1–41.

[69] Nil Kilicay and Cihan H Dagli. "Methodologies for understanding behavior of system of systems". In: *CD Proceedings of Conference on System Engineering Research*. 2007, pp. 14–16.

[70] Paulette Acheson, Cihan Dagli, and Nil Kilicay-Ergin. "Model based systems engineering for system of systems using agent-based modeling". In: *Procedia Computer Science* 16 (2013), pp. 11–19.

[71] Vadim Kotov. *Systems of systems as communicating structures*. Vol. 119. Hewlett Packard Laboratories, 1997.

[72] Jo Ann Lane and Richard Turner. "Improving development visibility and flow in large operational organizations". In: *International Conference on Lean Enterprise Software and Systems*. Springer. 2013, pp. 65–80.

[73] Richard Turner and Jo Ann Lane. "Goal-Question-Kanban: applying lean concepts to coordinate multi-level systems engineering in large enterprises". In: *Procedia Computer Science* 16 (2013), pp. 512–521.

[74] Flávio Oquendo. "Software architecture challenges and emerging research in software-intensive systems-of-systems". In: *European Conference on Software Architecture*. Springer. 2016, pp. 3–21.

[75] Milena Guessi, Flavio Oquendo, and Elisa Yumi Nakagawa. "Checking the architectural feasibility of systems-of-systems using formal descriptions". In: *2016 11th System of Systems Engineering Conference (SoSE)*. IEEE. 2016, pp. 1–6.

[76] FIWARE. *IoT Discovery*. Tech. rep. 2018.

[77] Jin Y Yen. "An algorithm for finding shortest routes from all source nodes to a given destination in general networks". In: *Quarterly of Applied Mathematics* 27.4 (1970), pp. 526–530.

[78] Pierre Hansen, Nenad Mladenović, Raca Todosijević, and Saïd Hanafi. "Variable neighborhood search: basics and variants". In: *EURO Journal on Computational Optimization* 5.3 (2017), pp. 423–454.

[79] Günther Bauer and Hermann Kopetz. "Transparent redundancy in the time-triggered architecture". In: *Proceeding International Conference on Dependable Systems and Networks. DSN 2000*. IEEE. 2000, pp. 5–13.

[80] Ariyan Abdulla and Erik Andersson. *Heuristiska algoritmer för schemaläggning i real-tidssystem med hänsyn till data beroenden*. 2018.

[81] Snap library 4.0. *user reference documentation*. in `https://snap.stanford.edu/snap/doc/snapuser-ref/index.html`. 2017.