Integrated Task and Message Scheduling in Time-Triggered Aeronautic Systems

Dissertation

zur Erlangung des akademischen Grades eines
Doktors der Naturwissenschaften
(Dr. rer. nat.)
durch die Fakultät Wirtschaftswissenschaften der
Universität Duisburg-Essen
(Campus Essen)

vorgelegt von
Dipl. Wirt.-Inf. Sebastian Voss
aus Geldern
München 2010

Tag der mündlichen Prüfung:	21.07.2010
Erstgutachter:	Prof. Dr. Klaus Echtle
Zweitgutachter:	Prof. Dr. Bruno Müller-Clostermann

Kurzfassung

Diese Arbeit präsentiert Techniken zur Generierung von integrierten Task- und Message Konfigurationen für zeitgesteuerte IMA Architekturen. Den präsentierten Ansätzen liegt das Problem der Erzeugung von integrierten Task und Message Schedules für zeitgesteuerte Netzwerke zugrunde. Die entwickelten Ansätze generieren dabei automatisch integrierte Task und Message Schedules, die speziellen Systemanforderungen genügen.

Unser erster Ansatz löst das Task und Message Scheduling Problem mittels Transformierung in ein Graphen - Problem. Ein Algorithmus wird entwickelt, der den Abhängigkeitsgraphen durchläuft und dabei sowohl das Task Scheduling auf Systemebene, als auch das Message Scheduling auf Kommunikationsebene einbezieht. Der präsentierte Ansatz arbeitet iterativ und enthält Überprüfungs- und Pfad-Erweiterungs-Funktionalität, die die zeitliche Konsistenz des entwickelten Schedules garantieren. Einer der Vorteile dieses Ansatzes ist die Skalierbarkeit, die es ermöglicht, auch größere aeronautische Architekturen zu untersuchen und entsprechende Konfigurationen bereitzustellen.

Ein zweiter Ansatz beschreibt und löst erstmals das gegebene Problem mit Hilfe von Model - Checking Techniken. Diese Arbeit zeigt, dass aktuelle Model - Checking und Bounded Model - Checking Techniken genutzt werden können, um integrierte Task und Message Schedules zu berechnen, die speziellen Systemeigenschaften genügen. Wir präsentieren einen Ansatz, der das Prinzip der Zustandsraumexploration zur Scheduling Synthese nutzt. Dazu entwickeln wir eine symbolische Kodierung, die es nicht nur ermöglicht gültige Konfigurationen zu finden, sondern auch solche, die optimal sind bezüglich der Minimierung der Ende-zu-Ende Latenz. Zusätzlich präsentieren wir einen heuristischen Ansatz, der es erlaubt, die Skalierbarkeit deutlich zu verbessern, indem er das Problem der erschöpfenden Suche einschränkt. Die entwickelte Heuristik führt dabei eine gesteuerte, gewichtsbasierte Erkundung des Zustandsraumes durch.

Die entwickelten Ansätze sind in einem Framework zur Scheduling Synthese implementiert. Dieses Framework erlaubt die Generierung von geeigneten System Konfigurationen als auch Komplexitätsuntersuchung für verschiedene Systemarchitektur - Szenarien. Dazu wird ein Ansatz zur Evaluierung von Komplexitätsuntersuchungen entwickelt.

Abstract

This thesis presents generation techniques for task and message configurations in time-triggered IMA architectures. The commonality among the given techniques is the problem of integrated task and message scheduling for time-triggered networks. The proposed approaches allow the automatic generation of task and message schedules, which comprises certain system requirements.

Our first approach solves the task and message scheduling problem by regarding it as a graph problem. We present an off-line scheduling algorithm that traverses the precedence graph. The approach integrates task scheduling at system level and message scheduling at communication level by iteratively traversing the given precedence graph. The algorithm incorporates backtracking and path extension functionality, guaranteeing the consistency of the developed schedule. The main advantage of the algorithm is that it scales up well even for large avionics applications.

Furthermore, this thesis extends ongoing research into task and message scheduling based on time-triggered networks by first using model checking techniques for solving this kind of problem. We demonstrate that state-of-the-art model checking and bounded model checking techniques can be used to compute a schedule that fulfills certain system requirements. Therefore, we introduce an approach that adopts the principle of symbolic state space exploration to schedule synthesis and provides a symbolic encoding which makes it possible to guarantee an optimal solution with respect to minimizing the system's end-to-end latency. A developed heuristic approach extends this approach in order to increase scalability by preventing from an exhaustive search through a guided, weight-based state-space exploration.

The developed approaches are implemented in a framework for scheduling synthesis. This framework enables the generation and investigate certain system configurations in terms of complexity. This is done by using a complexity evaluation approach, which is developed is this thesis.

Acknowledgments

First of all, I am especially grateful to Prof. Klaus Echtle for offering me the opportunity to prepare a PhD thesis under his supervision. I am very thankful for his continuous technical and personal support in all phases of this thesis. He always offered time to discuss my ideas and supported me with constant and motivating encouragement. I am also grateful to Prof. Bruno Müller-Clostermann for his constructive support and his attendance to write the second expertise.

I particularly like to thank my mentor Dr. Maria Sorea for supporting me with endless patience and guidance. It has been a pleasure to discuss with her and gave me the chance to learn a lot. This thesis would not have been possible without her.

Many thanks to my teamleader at EADS Innovation Works, Josef Schalk, who offered me the chance to prepare this thesis in the industrial context.

I wish to express sincere appreciation to my friends, who contributed by talks, visits, their friendship and many other non-research ways, making my life more beautiful.

Very special thanks go to my family for all their support and love. My parents Wilhelm and Katharina, for the wonderful education and all the opportunities they offered me. They have contributed more than a lot to make this thesis possible. Many thanks to my sister Simone for her visits, talks and enjoyable times not only in Munich. It helped me going on.

Finally, my deepest thanks to my wonderful girlfriend Nicole, for her constant and beloved support over all the recent years. Even with larger distance she supported me by trusting in me and gave me confidence during this work. Thank you for your patience and your love. Without her, this thesis would not have been possible.

Danke!

Contents

Contents **IX**

1 Introduction **1**
1.1 Avionics Systems 1
1.2 Motivation 2
1.3 Objectives and Contributions of this Thesis 3
1.4 Structure of this Thesis 7

2 Concepts and Terms **9**
2.1 Time-Triggered Architecture 9
2.2 Basic Notations for Scheduling in Distributed Systems 10
2.3 Problem Formulation 13
2.4 Model Checking 14

3 Scheduling Algorithms for (Hard) Real-Time Systems **17**
3.1 Classification of Scheduling Algorithm 17
3.1.1 Static vs. Dynamic Scheduling 18
3.1.2 Online vs. Offline Scheduling 18
3.1.3 Preemptive vs. Non-Preemptive Scheduling 19
3.2 Scheduling for Uni- and Multiprocessor Systems 19
3.2.1 Scheduling for Uni-Processor Systems 19
3.2.2 Scheduling for Multi-Processor Systems 20
3.3 Scheduling for Distributed Systems 21
3.4 Related Research 22
3.4.1 Holistic Schedulability Analysis 22
3.4.2 Combined Task and Message Scheduling using Branch-and-Bound . . . 23
3.4.3 Combined Task Message Scheduling using Satisfiability Checking 23
3.4.4 Scheduling Multi-Mode Real-Time Distributed Components 24
3.4.5 An Improved Scheduling Technique for Time-Triggered Embedded Systems 25
3.4.6 Optimal Task Graph Scheduling with Binary Decision Diagrams 26

4 Algorithm for Integrated Task and Message Scheduling **27**

4.1 Functional description of new scheduling algorithm . . . 27
4.1.1 Calculation of longest path . . . 28
4.1.2 Initial starting point . . . 28
4.1.3 Precedence Graph Traversal . . . 29
4.1.4 Backtracking and path extension . . . 33
4.2 Discussion . . . 35
4.2.1 Cases of guaranteed optimality . . . 35
4.2.2 Deviation from expected optimum in worst case scenario . . . 36

5 Symbolic Task and Message Scheduling 37
5.1 Basic idea of using model checking for solving scheduling problems . . . 37
5.2 Requirements to Symbolic Task and Message Scheduling . . . 38
5.3 The Task and Message Scheduling Model . . . 41
5.3.1 State Representation . . . 41
5.3.2 Initial State . . . 43
5.3.3 Goal State . . . 43
5.3.4 Transitions . . . 43
5.3.4.1 Start Task Transition . . . 44
5.3.4.2 Run Task Transition . . . 45
5.3.4.3 Change Task Transition . . . 45
5.3.4.4 End Task Transition . . . 46
5.3.4.5 Start Message Transition . . . 47
5.3.4.6 End Message Transition . . . 47
5.3.4.7 Wait Message Transition . . . 48
5.4 SAL Representation . . . 48
5.4.1 Representation of a state . . . 49
5.4.2 The basic module . . . 51
5.4.3 Initialization . . . 51
5.4.4 Transitions . . . 52
5.5 Transition Ordering . . . 57
5.6 The 'Time' variable . . . 58
5.7 Solving (Hard) Real-Time Scheduling Problems using SAL . . . 59
5.7.1 Task and Message Scheduling using SAL-SMC . . . 60
5.7.2 Task and Message Scheduling using SAL-INF-BMC . . . 61
5.7.3 Counterexample and Schedule . . . 62
5.8 Binary Search Algorithm for finding the optimal solution . . . 64

6 Weighted Symbolic Scheduling 67
6.1 Goal . . . 68
6.2 An approach for state space reductions . . . 69
6.3 Results and Discussion . . . 73

7 Framework for Scheduling Synthesis 75

7.1 Graph Generation . . . 75
7.1.1 Single Graph Generation . . . 76
7.1.2 Multiple Graph Generation . . . 77
7.1.3 General Parameter for Graph Generation . . . 80
7.1.4 Precedence Graph Editor . . . 81
7.2 Code Generation . . . 82
7.3 Result Generation . . . 82

8 Analysis and Results **85**
8.1 Complexity evaluation for scheduling configurations . . . 85
8.2 Design of experiments . . . 87
8.3 Results . . . 88
8.3.1 General Results of Task and Message Scheduling . . . 89
8.3.2 Relation of Task and Nodes . . . 90
8.3.3 Variation in Precedence Graph Layouts . . . 93
8.3.4 Effects of dynamic reordering . . . 97
8.3.5 Effects of an additional transition . . . 98
8.3.6 Weighted Symbolic Task and Message Scheduling . . . 99
8.4 Discussion of Results . . . 102

9 Conclusion and Future Work **105**
9.1 Accomplishments . . . 105
9.2 Perspectives . . . 106

List of Figures **109**

List of Tables **111**

A SAL Code Example **113**

B Abbreviations **119**

Bibliography **121**

Chapter 1

Introduction

1.1 Avionics Systems

There is an ongoing revolution in designing avionics systems. Current avionics systems are predominately federated. Federated avionics architectures make use of distributed avionics functions that are deployed in self-contained units. Separate Line Replaceable Units (LRUs) or Line Replaceable Modules (LRMs) contain independent collections of dedicated computing resources (computing processor, communication and I/O) for each avionics function. These federated architectures lead to increasing quantities of hardware and wiring. A further increase results from the ever-rising number of software controlled systems manifested by new functionality for performance (e.g. flight management systems), improved safety (e.g. traffic collision avoidance), improved maintenance (e.g. aircraft condition monitoring) or improved passenger comfort (e.g. cabin environment control). Furthermore, during an aircraft life cycle, costs of modifications including parts obsolescence mitigation and functional upgrades become more significant. Although federated systems provide a robust avionics architecture, they have high realization and maintenance costs and therefore modern design favors a more integrated approach, in terms of modularity and system integration.

The ongoing trend from federated to integrated modular avionics systems leads to avionics with Integrated Modular Avionics (IMA) architectures. The IMA concept is one of the major trends in building current state-of-the-art avionics systems. New developments such as the Airbus A380 or Boeing 787 already favor a migration of federated and integrated architecture.

IMA architectures are based upon a set of modular systems that share a set of computing, communication and I/O resources in order to reduce weight, volume/size, and power of hardware for more fuel-efficiency. These advances in IMA promise to increase the performance and reliability of avionics systems while simplifying the development and certification of flight software and avionics hardware. But these advantages come at a price: the IMA approach allows multiple applications of different criticality levels to share common computing resources, it is important to

keep individual applications away from potential interference. Protection of integrated applications and system resources is possible via temporal and spatial partitioning. Spatial partitioning guarantees that an application has exclusive control over its own data and state information. With spatial partitioning, an application can be protected from any erroneous behaviors of other applications while sharing the same physical resources. Temporal partitioning guarantees that an application has temporal exclusive access to its pre-allocated computing, communication and I/O resources. As a shared resource for communication, IMA replaces some of the point-to-point cabling with shared communication buses (e.g. high-speed multiplexed networks). At the point of writing this thesis, integration level reached in air transport commercial avionics is characterized by a common open system bus ARINC664 [ARI05], with interfaces to different systems including flight control, engine control and passenger entertainment. But also different (fault-tolerant) shared bus systems have a promising role in IMA architectures.

In recent years, time-triggered (TT) architectures [KB01] have gained momentum for platform-based applications. Time-triggered system architecture, such as Flexray [Con05], Time Triggered Protocol (TTP) [AG03] or SAFEbus [ARI93, HD93], are widely used in embedded systems for safety-critical applications [Pau02]. The FlexRay communication standard [Con05] for instance, has gained industry-wide acceptance as the next-generation automotive networking standard [PH08, SJ08].

As time-triggered systems provide predictable communication behavior in a timely deterministic way, it leverages their potential use for the mentioned class of IMA systems as well. For enforcing temporal partitioning, the time-triggered shared resources have to be scheduled, while guaranteeing timing constraints of the application. With guaranteed pre-scheduled temporal partitioning, applications can meet their timing requirements.

The question whether given applications meet safety-critical system requirements always depends on the proper design and configuration of these systems. The challenge is to find such a configuration.

1.2 Motivation

Safety-critical system design incorporates several fundamental functional and non-functional requirements: availability, integrity, predictability and reliability. These requirements should be reflected in the system design and configuration. The IMA architecture, including spatial and temporal partitioning, provides a concept for safety-critical system design. But this IMA concept needs to be supported by a system configuration appropriate to the timing requirements of the actual application. In order to obtain such a configuration, efficient techniques are needed.

In IMA, several applications may be integrated and communicate on a shared resource with each other or with distributed applications. Applications consist of a number of communication tasks with precedence constraints, i.e. constraints specifying their execution ordering. These tasks communicate via message passing over the shared communication resource, which can be im-

plemented as a time-triggered network. Communication in time-triggered networks is realized by a time-division multiple-access (TDMA) discipline with a fixed slotting scheme and defined communication cycles. A further characteristic is that the application is synchronous with the underlaying communication bus. For such a system composed of a number of communicating applications on a distributed network connected by a time-triggered bus, effective configuration is needed to guarantee functional and non-functional avionics system requirements [PRS08]. This configuration leads to a scheduling policy that determines the temporal behavior of a system configuration. Therefore, a scheduling policy has to take into account not only the constraints imposed by the applications (e.g. precedence relations of the application tasks) but also the characteristics and efficient usage of the underlying communication system [PEP99, TV99]. Message slot allocation cannot be accurately computed before a task schedule is computed (from which, for example, we may want to know the time to send these messages). If the task scheduling problem and the message scheduling problem are regarded independently, further timing inconsistencies can arise. A task, for instance, never receives the actual correct value, because the message slot containing that information is allocated after the starting time of the task. These inconsistencies become worse as the number of tasks in a given application increases, with the resulting increase in the complexity of the precedence graph. Thus, the problems of task scheduling and message scheduling should be regarded in an integrated way. As embedded systems become larger and more complex, this task becomes more and more difficult, especially for the system designer. As complexity rises rapidly this cannot be performed manually any more.

From an academic point of view, several solutions for task and message scheduling for time-triggered networks are known: constrained-based approaches, e.g. [MFHS05, JLL04], the usage of branch-and-bound techniques [AS99] as well as the application of different techniques for solving the task and message scheduling problem, e.g. [TC94, PEP99]. However, these approaches are not applicable for calculating a scheduling policy for IMA systems, because they mostly consider different optimization parameters. Furthermore, commercial scheduling tools are provided by a range of different suppliers (e.g. DECOMSYS, TTTech, Vector). Most of these tools do not take into account any efficient techniques for combining the task and message scheduling problem. Furthermore, the optimization of different parameters (e.g. maximum latency), is not supported either.

In order to guarantee timing requirements in a given IMA architecture, integrated and efficient techniques for system configuration are needed. These techniques need to incorporate pre-scheduled application behavior as well as a time schedule of the underlaying communication system to meet (hard) real-time requirements of the application. It is the main objective of this thesis to develop and implement such methods.

1.3 Objectives and Contributions of this Thesis

We address in this thesis the generation of task and message configurations in time-triggered IMA architectures. We propose several approaches that allow the automatic generation of task

and message configurations that are optimal with respect to given system requirements, such as end-to-end latency.

Scheduling Algorithm for Integrated Task and Message Scheduling

Our first approach regards the task and message scheduling problem as a graph problem. An off-line scheduling algorithm is developed for traversing this graph, which augments conventional scheduling rules with algorithms addressing the specific problems of scheduling messages on time-triggered communication busses. The algorithm allows to automatically compute schedules even for large aeronautic applications. Our approach integrates task scheduling at system level with message scheduling at the communication level.

At system level the task scheduling problem is represented by a directed acyclic weighted graph. This graph incorporates the task precedence relations given by the application and is therefore called a precedence graph. Tasks are represented by vertices, and the precedence relations between them are represented by edges. Thus, the precedence graph induces a partial order on the task set. At the communication level a set of messages needs to be allocated to a certain communication slot. Therefore, each edge of the precedence graph is labeled with a given weight. This weight defines the number of messages to be transferred from the sending task to the receiving task. In this context, the addressing sending and receiving task correlates according to the direction of the connected edge. The weight of the vertices is given by the task duration.

The algorithm starts by using a weighted precedence graph as an input. In a first step the algorithm calculates the longest path through the given graph with respect to the task length of vertices and the amount of messages of the edges. Here, a depth-first-search algorithm is used, slightly adapted to deal with vertices and edge weights. Starting from the sink vertex of the longest path, the algorithm incrementally traverses backwards along the calculated longest path until the source vertex is reached. The sink vertex (task) is allocated to the end of the given communication cycle, while its starting time is calculated by subtracting its computation time. In each step of this graph traversal, whenever a new task on the longest path is reached, the algorithm searches for both successor and predecessor tasks according to the task ordering. Messages are allocated to the next available time slot on the communication bus, with minimal intertask communication. This slot is known by the application due to the synchronization of the application and the time-triggered communication bus(es).

The main challenge of the task and message scheduling problem is to handle complex applications that have a highly concurrent precedence graph. The reason, for instance, is that two different tasks are trying to allocate the same computing resource. In this case a distinction is made which causes one task to wait until the favored task is processed. When optimizing for minimal end-to-end latency this might lead to suboptimal solutions. This problem is solved by the mentioned incremental successor and predecessor calculation in each step.

The main advantage of the algorithm is that it scales up well for large applications. The price to pay is that the algorithm does not always find an optimal solution. However, the algorithm

delivers its result very fast. Even precedence graphs with 100 or 1000 different (concurrent) task sets can be handled in minutes. The presented algorithm is therefore highly scalable and thus applicable for large avionics systems.

Symbolic Task and Message Scheduling

This thesis extends ongoing research in task and message scheduling based on time-triggered shared resources by first using Model Checking techniques for solving this kind of problems. This thesis proves that state-of-the-art model checking and bounded model checking techniques can be used to compute a schedule that fulfills certain system requirements.

We adopt the principle of symbolic state space exploration to scheduling synthesis and propose Symbolic Task and Message Scheduling as a novel approach to the problem of task and message scheduling for TDMA-based avionics applications. This approach allows to automatically compute schedules with minimal end-to-end latency. Here, the framework of Model Checking is used to carry out the verification performed on finite-state space. We use SAL (Symbolic Analysis Laboratory, `http://sal.csl.sri.com`) [dMOR+04] from SRI International as a framework for specification and scheduling synthesis. In particular, we use state-of-the-art model checking and bounded model checking techniques to compute the schedules. Experimental results demonstrate how the latest generation of model-checking tools meets the challenges of providing both a convenient modeling language and the performance to solve given scheduling problems.

Our approach relies on a symbolic encoding which makes it possible to guarantee an optimal solution of the given scheduling problem. The symbolic encoding is done by transferring and defining the task and message scheduling problem as a finite-state model-checking problem $\mathcal{M} \models \varphi$. Therefore, we define a basic model, containing all state variables (e.g. for tasks, messages, nodes, etc.). The initial state is specified as a situation in which no task is started. The goal states are those states where all task have finished and all messages have been sent.

The proposed approach works by gradually constructing a schedule, beginning in a state where no task has started and no messages have been sent on the bus yet, and proceeding one step at a time assigning starting times to tasks and slot positions to messages. This construction process is mapped onto a set of transitions, specified as guarded commands. Six transition relations are used to model the problem of integrated task and message scheduling. Four different transitions are needed to define task level transitions, while two more transitions are used for the message level. These transitions are encoded as a conjunction of constraints over the current state variables. The reason we use a set of transition relations instead of a single one can be seen in the complexity of the integrated task and message scheduling problem and its consequently large and complex number of constraints for a single transition. Although the transitions are in arbitrary order, they cannot be executed non-deterministically since arbitrary execution would lead to an inaccurate schedule. The ordering of transitions has to take into account aspects such as task precedence, the duration of a certain task and its amount of messages sent. Therefore, the encoding explicitly

defines and restricts how a task and message schedule for time-triggered networks can be built. The given precedence graph is traversed from source to sink. Whenever the precedence graph allows for different possible solutions, characterized by concurrent access to a shared resource (either different messages trying to allocate the next available time slot on the communication bus or several tasks waiting for access to the shared computing resource), we use the model checker's capabilities to explore all interleaved possibilities. The model checker analyzes the specified task and message scheduling model with respect to a given property, stating that there is no possible schedule fulfilling the given system requirement. By failing the property we obtain a counterexample containing a task and message schedule. This schedule might not be optimal with respect to end-to-end latency because the transition system is allowed to make timeless transitions. To obtain an optimal task and message schedule we further use a binary search for finding that solution. Thus, schedule generation that is automatically optimal is guaranteed by construction.

The major challenge is the scalability of the given approach. Increasing the number of tasks in a precedence graph would increase the state-space and thus the computation time exponentially. Thus, schedules can be computed until an upper bound of tasks in a precedence graph (depending on the model checker used and computer resources). In order to reduce the state-space and enlarge the given number of tasks in a precedence graph, we have developed a heuristic approach that reduces the state space significantly and thus the computation time for finding an optimal solution.

Using model checking techniques for solving scheduling synthesis for avionics has further advantages. Aeronautic requirements for communication systems are mainly driven by certification and non-functional needs, as manifested in recent efforts to improve system verification and design assurance. Achieving Level B or even Level A with respect to DO-178B [RTCfA] is one of the biggest cost drivers for aeronautical issues. Thus, methods, technologies and algorithms are needed to support the system designer on the one hand and implicitly provide assured awareness for supporting the process of system validation and verification on the other hand. The reduction of system verification time and design assurance time is highly cost-effective and can be supported by using efficient techniques such as model checking.

Framework for Scheduling Synthesis

We have developed a framework for integrating both methods for scheduling synthesis. This framework is composed of different elements. A graph generator is included which is able to generate precedence graphs, including all necessary parameters (e.g. number of nodes, tasks, precedence relations, etc.). This can be done explicitly (e.g. according to a given aeronautic application) or randomly. In the latter case, numerous generated graphs can be randomly generated and used for the experimental series. Experimental series lead to statements about average computation time, as well as the influence of parameter variations. Furthermore, the framework consists of an interface to import certain communication-dependent parameters (e.g. FIBEX - file including a FlexRay configuration).

The framework includes the integrated task and message scheduling algorithm, as well as a code generator for the symbolic approach. The precedence graphs are automatically translated into SAL specifications according to the symbolic encoding scheme presented in chapter 5. The framework has an interface to the SAL framework for executing the task and message scheduling problem. The calculated results are transferred back. Experiment series, composed of different precedence graphs containing certain property settings, can be computed and compared as well. For visualization, textual as well as gnuplot-based graphical elements are available.

The presented framework can be used for solving the task and message scheduling problem with the approaches presented in this thesis.

1.4 Structure of this Thesis

This thesis is structured as follows. Chapter 2 presents background information and basic notions. Therefore, we review the basic concepts of Time-Triggered Architectures (TTA) and introduce the basic notations in the domain of scheduling in distributed systems. Furthermore, the problem statement of this thesis is introduced and we briefly highlight the principles of model checking. We give an overview on the SAL framework from SRI International used in this thesis. Chapter 3 introduces algorithms used for (hard) real-time systems in the domain of scheduling. We survey a classification scheme of scheduling algorithms to better classify the algorithms presented in this thesis. An overview and discussion of related academic work is given as well.

Chapter 4 introduces the newly developed scheduling algorithm for integrated task and message scheduling. We present a detailed functional description and provide a discussion of it. Symbolic Task and Message scheduling is introduced in chapter 5. A particular and detailed formal description of the algorithm is provided as well as implementation issues. Furthermore, a heuristic approach for state space reduction is introduced in Chapter 6. Chapter 7 presents the implemented framework for scheduling synthesis, including the description of the various steps and the design and implementation of the scheduling approaches presented.

Chapter 8 presents analysis and results for the developed techniques and algorithms. An approach for complexity evaluation of different scheduling configurations is introduced. Furthermore, the design of experiments and their results are presented. These results are discussed to evaluate the new approaches of this thesis. We conclude in chapter 9.

Chapter 2

Concepts and Terms

This chapter gives an overview about time-triggered architecture (section 2.1), introduces terms and basic notations of scheduling in distributed systems (section 2.2) and defines the given scheduling problem (section 2.3) on which the objectives and contributions of this thesis are based. Finally, we introduce the general concepts and terms of model checking (section 2.4).

2.1 Time-Triggered Architecture

This section describes the basic concepts of Time-Triggered Architecture (TTA). In recent years, time-triggered (TT) architectures [KB01] have gained momentum for platform-based applications. Time-triggered system architecture, such as Flexray [Con05], Time Triggered Protocol (TTP) [AG03] or Safebus [ARI93, HD93], are widely used in embedded systems for safety-critical applications [Pau02].

As time-triggered systems provide predictable communication behavior in a timely deterministic way, it leverages their potential use for the mentioned class of IMA systems as well. For enforcing temporal partitioning, the time-triggered shared resources have to be scheduled, for guaranteeing timing constraints of the application. With guaranteed pre-scheduled temporal partitioning, applications can meet their timing requirements.

Communication in time-triggered networks is realized by a time-division multiple-access (TDMA) discipline, in which n nodes share a time-triggered bus based on a cyclic schedule γ. Each node $Node_i$ can transmit only during a predetermined time interval, denoted as TDMA (or time) slot. A slot has a fixed size, in the sense that if a node does not have a message to send in his slot, the network remains idle for this period. A sequence of slots forms a TDMA round. The time intervals associated to each slot are disjoint. Several rounds are combined into a *communication cycle* that is repeated periodically. Tasks are executed on nodes, and communicate with each other by message passing. A node $Node_i$ might send one or more messages in several slots in a TDMA round.

TDMA protocols require clock synchronization and an off-line schedule containing the allocation of slots. Not only the assignment of nodes to slots, but also the assignments of messages to time slots is done. Thus, the entire schedule is generated off-line and therefore guarantees a predictable behavior of the system.

2.2 Basic Notations for Scheduling in Distributed Systems

A real-time system consists of several components for providing various functions. These functionalities can be described as computational activities, which run in parallel. We use the term *tasks* to denote a logical unit of computation and we represent a program as a set of tasks. In the literature, terms as *task*, *job* and *process* are used interchangeable, representing a sequential computation. Therefore, a task can be described as a computation that is executed by the CPU in a sequential fashion [But04].

A computation resource may execute a set of concurrent tasks, that is, tasks that can overlap in time. In this case, the CPU needs to be assigned to the various tasks according to a predefined criterion, these *scheduling rules* are also called a *scheduling policy* [But04].

Furthermore, tasks can be classified as *periodic* or *aperiodic*. Periodic tasks are triggered regularly, in a time-triggered fashion, whereas aperiodic tasks are triggered by events, which can be completely random. *Sporadic* tasks [Mok83] can be seen as a subclass of aperiodic tasks but with a minimum time between any two activation events. Most scheduling algorithms are defined for periodic tasks and are also valid for sporadic tasks.

In general, a real-time task t_i is characterized by the following parameters [But04] (cf. figure 2.1):

- Arrival time a_i (also known as request time or release time): the time at which a task becomes ready for execution;
- Computation time c_i: time necessary to the processor for executing the task without interruption;
- Absolute deadline d_i: time before which a task should be completed to avoid damage / degradation;
- Relative Deadline D_i: difference between absolute deadline and the arrival time: $D_i = d_i - a_i$;
- Start Time s_i: time at which a task starts its execution;
- Finishing Time f_i: time at which a task finishes its execution;
- Response Time R_i: difference between the finishing time f_i and the arrival time a_i: $R_i = f_i - a_i$.

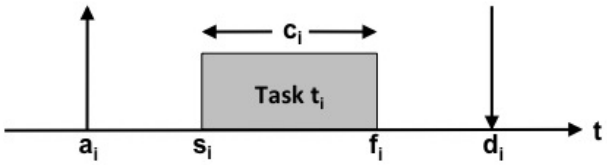


Figure 2.1: Typical Task Parameters

Typical attributes specific to a real-time task t_i are:

- Period P_i: the time interval between successive releases, in case of periodic tasks.
- Worst-case execution time $wcet_i$ or C_i: the maximum execution or computation time needed to execute the task without interruption on a particular processor. An obvious assumption is $C_i \leq P_i$.
- Phase Φ_i : the time elapsed between time 0 and the release of the first instance of the task: Φ_i = a_i. The phase is the offset at system start-up.
- Processor-utilization U_i: the fraction of the processor time used for the execution of the task (task-specific utilization): $U_i = C_i/P_i$.

In traditional real-time applications, especially in the aeronautic domain, the set of tasks does not change at run-time. Therefore, most algorithms consider the problem of scheduling at a fixed set T of n tasks: $T = \{t_1, t_2, ..., t_n\}$. The following concepts were defined for a task set and they are used in the schedulability analysis of different algorithms:

- The *overall processor-utilization* [LL73] for the set of tasks T is computed as $U = \sum_{i=1}^{n} C_i/P_i$; a necessary condition for the schedulability of a task set is $U \leq 1$.
- The *hyperperiod* $H = lcm\,{}_{i=1}^{n} P_i$ is the time after which the pattern of periodic task arrivals repeats itself, where lcm denotes the *least common multiple*.
- A task set is called *harmonic* when $\forall(P_i, P_j)$, if $P_i > P_j$ then P_i is an integer multiple of P_j.
- A task set is called *synchronous* if the phases of all tasks are equal: $\Phi_i = \Phi_j$, $\forall i, j = 1, ..., n$.

In order to describe the functionality of a real-time system, tasks have been introduced to describe the computational activities of the system. However, depending on the different system requirements and functionalities, tasks often cannot be executed in arbitrary order. Therefore, the interaction of tasks need to be described with respect to their precedence relations. These relations define the execution ordering and are captured by a directed acyclic labeled graph, called *precedence graph* $\mathcal{G}$, as illustrated in figure 2.2.

Thus, a precedence graph $\mathcal{G} = \{V, E\}$ can be seen as an abstract model for a functional system representation. Each vertex V in this graph represents a task t, while $E \subseteq V \times 2^M \times V$ represents a set of edges between the tasks, where M denotes the set of messages. A directed edge $e_{ij} = \langle t_i, \{m_1, \ldots, m_k\}, t_j \rangle$ shows the direct precedence between the sending task t_i and the receiving task t_j. This precedence relation is formally depicted as $t_i \prec t_j$ (to be read as t_i precedes t_j). Intuitively, $t_i \prec t_j$ means that t_i and t_j must be scheduled such that $f_i < s_j$, where f_i is the finishing time for the task t_i and s_j is the starting time for the task t_j. The set $M = \{m_1, \ldots, m_k\}$ denotes the messages that t_i sends to t_j.

For an edge $e_{ij} \in E$, its weight u_{ij} gives the number of messages to be transfered from task t_i to t_j, that is the cardinality of the associated message set. For instance, the edge from task t_1 to task t_2 in figure 2.2 has one message associated with it, namely, m_1. The weight w of an edge can be assumed as the duration of a sending slot. The weight of a vertex is given by the task duration of the task associated to the vertex, that is, $w_{t_i} = c_i$, where c_i is the computation time for task t_i.

Definition 1 *Given $\mathcal{G} = (V, E)$ with $V = T$, for a task $t \in T$ and a message m, we define the* destination $\mathcal{D}$ *of t under m, as the function $\mathcal{D} : T \times M \rightarrow T$, with $\mathcal{D}(t, m) = \{t' \in T | \langle t, \{m\}, t' \rangle \in E\}$.*

A scheduling algorithm provides the order in which computational activities get access to active resources such as computation (CPUs) or communication (networks) resources. *Scheduling* is the process of creating such an ordered list, called the schedule γ.

Definition 2 *A schedule γ is defined by $\gamma = \{t_i \mapsto \gamma_i | \forall t_i \in T\}$, where γ_i is described as the tuple $\gamma_i = \langle s_i, \{\sigma(m_1), \ldots, \sigma(m_j)\} \rangle$. σ is a function $\sigma : M \rightarrow SL$ that allocates slots to messages (compare section 2.3).*

As stated before, the objective is to find a task schedule which as well incorporates a message schedule and guarantees (hard) real-time system requirements while considering reliable and predictable communication on a time-triggered network. Therefore, we start by defining a *valid*, *feasible*, and an *optimal* schedule in the context of this thesis.

Definition 3 *A schedule γ is said to be* valid*, iff:*

- *Each task t_i in γ has an exclusively access to the CPU of the task's node (non-preemptive access)*
- *All precedence relations must be satisfied.*
- *Intertask communication caused by the transmission of messages according to the precedence relations must be guaranteed. For instance, a task t_i sends a message m_1 to task t_j. The duration for the intertask communication (represented by length/duration of m_1) between t_i and t_j is denoted dt_{t_i,t_j}. Hence, $f_i + dt_{t_i,t_j} <= s_j$ must be satisfied.*

Definition 4 *A valid schedule γ is said to be* feasible *for (hard) real-time systems, if it guarantees the arrival times and deadlines of each task, formally, $\forall\ t_i \in T : s_i \geq a_i$ and $f_i \leq d_i$.*

An *optimal* scheduling algorithm is the best one according to some criteria. In this thesis, we optimize the schedule with respect to the given system requirement of minimizing the end-to-end latency.

Definition 5 *A* feasible *schedule γ is called* optimal*, if the length $|\gamma| = f_{sink} - s_{source}$ is minimal for a given task precedence graph $\mathcal{G}$.*

2.3 Problem Formulation

Formally, the task and message scheduling problem is defined as follows:

Let $N = \{n_1, n_2, ..., n_m\}$ be a set of nodes, $T = \{t_1, t_2, ..., t_n\}$ a set of tasks, and $M = \{m_1, m_2, ..., m_o\}$ a set of input/output messages of the tasks. The dependencies between the tasks in T are captured by a precedence graph $\mathcal{G}$. Furthermore, let $\eta : N \rightarrow 2^T$ be a function that assigns to every node a task running on it, and $\tau : T \rightarrow 2^M$ a function that assigns a set of messages to tasks. The set of time slots is denoted by $SL = \{sl_1, \ldots, sl_k\}$. A function $\sigma : M \rightarrow SL$ allocates slots to messages. The scheduling problem consists in determining the starting time and slot(s) position of the tasks in T that is, to calculate for every task $t_i \in T$, the tuple $\gamma_i = \langle s_i, \{\sigma(m_1), \ldots, \sigma(m_j)\}\rangle$, such that the overall schedule γ is *optimal*.

Example 1 This simple example is used to describe the concepts of a precedence graph $\mathcal{G}$. Consider a set of tasks $T = \{t_0, t_1, t_2, t_3\}$, and a set of messages $M = \{m_0, m_1, m_2, m_3\}$, cf. figure 2.2. Furthermore, $\tau(t_0) = \{m_0, m_1\}$, $\tau(t_1) = \{m_2\}$, $\tau(t_2) = \{m_3\}$, $\tau(t_3) = \{\}$, $\eta(n_0) = t_0$, $\eta(n_1) = \{t_1, t_2\}$ and $\eta(n_2) = \{t_3\}$. For simplicity reasons we just use time units, and assume an equal computation time for each task with $c_i = 2$ time units. Furthermore, $\mathcal{D}(t_0, m_0) = t_1$, $\mathcal{D}(t_0, m_1) = t_2$, $\mathcal{D}(t_2, m_3) = t_3$, $\mathcal{D}(t_1, m_2) = t_3$.

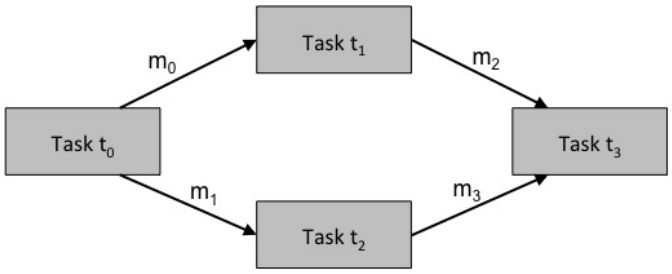


Figure 2.2: Simple Example of a Precedence Graph $\mathcal{G}$

Remark 1 We consider the hyperperiod H of a task to be equivalent to the communication cycle, thus identical for all tasks in T. Hence, the arrival times a_i, which denote the earliest time a task can be invoked is equivalent to the beginning of the communication cycle for all tasks. The deadline d_i, which is the latest time it can finish its execution corresponds to the end of the communication cycle. Preemption of tasks is not considered.

2.4 Model Checking

A very attractive alternative to simulation and testing is the approach of *formal verification*. While simulation and testing explore only some of the possible behaviors and scenarios, formal verification conducts an *exhaustive exploration* of all possibilities. In this thesis, we concentrate on the method of *model checking* as a formal verification technique by which a desired behavioral property of a defined system model is analyzed.

Compared to other approaches, the *model checking* method has two advantages [EMCGP99]:

- The completely automatic verification of finite-state concurrent systems, which needs no further user supervision or expertise in mathematical discipline.
- By failing the desired property, the model checker always produces a *counterexample* that demonstrates the behavior which falsifies the property. This characteristic is used in this thesis to generate integrated task and message schedules.

The main disadvantage of model checking is the state explosion that can occur if the system being verified has many components.

In this thesis, we use model checking techniques by adopting the principle of state space exploration to scheduling synthesis. In the following the basic principles of model checking are explained.

Principles in Model Checking

Consider a set of variables $V = \{v_1, ..., v_n\}$ interpreted over nonempty domains $\mathcal{P}_1$ through $\mathcal{P}_n$ together with a type assignment ϕ such that $\phi\ (v_i) = \mathcal{P}_i$. For a set of typed variables V, a *variable assignment* is a function ν from variables $v \in V$ to an element of $\phi(v)$. The variables in $V = \{v_1, ..., v_n\}$ are also called *state variables*, and a *program state* is a variable assignment over V.

Definition 6 ***($\mathcal{T}$-Programs)*** A quadrupel $M = \langle V, I, G, T\rangle$ is a $\mathcal{T}$- *program* (or a model M) over V, where interpretations of the typed variables V describe the set of states. For a given program $\mathcal{T}$, the set of *boolean constraints* $\mathrm{Bool}(\mathcal{T})$ includes all constraints in $\mathcal{T}$and it is closed under conjunction $\wedge$, disjunction $\vee$ and negation $\neg$. A *state* of a concurrent system can be described by giving values for all the variables $v_i \in V$. In other words, a state s is a function that associates a value in $\mathcal{P}$ with each variable $v \in V$, thus $state : V \rightarrow \mathcal{P}_n$. The predicate $I \in \mathrm{Bool}(\mathcal{T})$ describes the initial states, and $G \in \mathrm{Bool}(\mathcal{T})$ describes the final goal states. $T \in \mathrm{Bool}(\mathcal{T}(\mathrm{V} \cup V'))$ specifies the transition relation between current states and their successor states. V' is a primed, disjoint copy of V. V denotes the current state variables, while V' specifies the next state variables. The set of $\mathcal{T}$- programs over V is denotes by $Prog(\mathcal{T})$.

For a program $M = \langle V, I, G, T\rangle$ a sequence of states $\pi(s_0, s_1, ..., s_n)$ forms a *path* through M, if $\bigwedge_{0 \leq i < n} T(s_i, s_{i+1})$. A state s is *reachable* in M if there is a path $\pi(s_0, s_1, ..., s_{n-1}, s)$ through

M and $I(s_0)$. A state property $\varphi \in \mathrm{Bool}(\mathcal{T}(V))$ is *invariant* in M, if $\varphi(s)$ holds for every reachable state s in M. A counterexample for a property φ is a path $\pi(s_0, ..., s_n)$ such that $I(s_0)$ and $\neg\varphi(s_n)$, and the length $len(\pi)$ of such a counterexample is given by the number of steps in this path ($len(\pi) = n$).

Let $\mathcal{M}= \langle V, I, G, T\rangle$ be a $\mathcal{T}$- program. The model checking problem is to decide whether $\mathcal{M} \models \varphi$ holds or not. If not, the model checker should provide a counterexample. In accordance with the two parameters of the model checking problem ($\mathcal{M}$ and φ), there are two basic strategies when designing a model checking algorithm: "Global" algorithms recurse on the structure of φ and evaluate each of its subformulas over all of $\mathcal{M}$. "Local" algorithms, in contrast, explore only parts of the state space of $\mathcal{M}$, but check all subformulas of φ in the process. Traditionally, *propositional tree logic* (PTL) [EMCGP99] model checking has been based on the local approach, while model checkers for *computational temporal logic* (CTL) [CE82] and other branching-time logics have used global algorithms [Mer01]. In this thesis, we assume φ to be a CTL formula.

Any finite-state system can be encoded as a program $\mathcal{M}$. As these programs are called *symbolic*, model checking algorithms that work on symbolic representations are called *symbolic model checking* (SMC) techniques [BCM$^+$92]. *Binary decision diagrams* (BDD) [And97] are a data structure for the symbolic representation.

Further details on model checking algorithms, theorems and proofs may be found in textbooks such as *Model Checking* from Clarke, Grumberg and Peled [EMCGP99].

An Overview of SAL

SAL stands for Symboblic Analysis Laboratory (see `http://sal.csl.sri.com`). SAL provides a framework for combining different tools for abstraction, program analysis, theorem proving, and model checking toward the calculation of properties (symbolic analysis) of transition systems [dMOS03]. The key part of the SAL framework is the language [BGL$^+$00] for describing transition systems, which provide notations for specifying state machines and their properties. Thus, the framework can be used for the specification and analysis of concurrent systems. SAL provides model checkers and several other tools for analyzing properties of state machines specifications. These model checkers are of interest for our analysis. These include a symbolic model checker (SMC), a witness and counterexample generating model checker (WMC) for finite-state systems, a SAT-based bounded model checker for finite-state systems (BMC), and a bounded model checker for infinite systems (infBMC).

The symbolic model checker `sal-smc` [dMOR$^+$04] uses the CUDD BDD package and provides access to many options for variable ordering, and for clustering and partitioning the transition relation. Experiments in this thesis mainly uses the symbolic model checker `sal-smc` of the SAL framework as well as SAL's infitite-state bounded model checker `sal-inf-bmc`.

Model Checking in the Aeronautic Context

System verification and design assurance in the aeronautic industry is a big issue. Aeronautic requirements for communication systems are mainly driven by certification and non-functional needs, as manifests in recent efforts of involving a system design process in the development of aeronautic systems. Customer needs for additional or modified system functionality, as well as unpredictable obsolescence policies request for a proper system design process. Achieving certification (according to Level B or even Level A with respect to DO-178B) is one of the biggest cost driver for aeronautical developments. Thus, methods, technologies and algorithms are needed which support on the one hand the system designer on a technical level, and on the other hand implicitly provide assured awarenesses for supporting the process of system validation and verification. Model checking is a promising technology for dealing with these issues.

Chapter 3

Scheduling Algorithms for (Hard) Real-Time Systems

In concurrent distributed systems, several computational activities compete for the set of resources, such as CPUs and networks. The scheduling discipline tries to solve the problem of efficient use of these shared resources. This chapter presents a classification (section 3.1) of scheduling algorithms and a brief overview of scheduling algorithms for uni- and multiprocessor systems (section 3.2). This provides a basis for introducing scheduling algorithms for distributed systems that incorporate both task scheduling at system level and message scheduling at communication level. We present the problem for scheduling in distributed systems in section 3.3 and introduce related research in this domain in 3.4).

Further details on scheduling algorithms, theorems, and proofs may be found in textbooks such as Buttazzo's *Scheduling Hard Real-Time Computing Systems* [But04] or Liu's *Real-Time Systems* [Liu00].

3.1 Classification of Scheduling Algorithm

Real-time systems consist of several computational activities that describe the functionality of the given system. A precedence graph highlights the interaction of these activities, as they cannot be executed in arbitrary order. Thus, scheduling algorithms refer to the problem of assigning computational activities to shared resources such as CPUs and networks in an effective way. The scheduling literature provides a variety of algorithms, depending on the restrictions enforced, ranging from independent periodic tasks in uniprocessor systems to periodic and aperiodic dependent tasks running on distributed systems.

In the literature, scheduling algorithms can be classified as follows (compare [But04]).

3.1.1 Static vs. Dynamic Scheduling

In static scheduling, all scheduling decisions are based on fixed parameters, assigned to tasks before their activation; whereas in dynamic scheduling, all decisions are based on dynamic parameters that might change at run-time (compare Buttazzo [But04]). Static scheduling needs a priori knowledge of all task attributes; therefore it is less flexible. Dynamic scheduling can provide a better processor utilization and supports non-predicted events, but it has a higher runtime overhead than static scheduling.

Static and dynamic scheduling have a further meaning in multiprocessor and distributed systems, depending on how these systems are configured. In a static system, the tasks are partitioned into subsystems and they are statically allocated to processors. The tasks on one processor are scheduled independently, except the cases when they must be synchronized. In a dynamic system, the tasks are dynamically dispatched to processors. There is a common queue for all processors and the task in the head of the queue is dispatched to the first processor that becomes idle. If preempted, a task can migrate from one processor to another in order to resume its execution. This thesis focuses on static scheduling, because the architecture and allocation of task to computing resources is (in most cases) given in avionic real-time systems.

3.1.2 Online vs. Offline Scheduling

Off-line scheduling computes all decisions at compile time and stores them in a dispatcher table. At run-time no scheduler is needed, but only a dispatcher which takes the next entry from the table. Off-line scheduling is also called *table-driven scheduling*, that incorporates a table determining which tasks to execute at which points in time. Thus, feasibility is proven constructively [BS05]. Off-line scheduling methods are capable of managing distributed applications with complex constraints (e.g. precedence, end-to-end deadlines, etc.). On the other hand, the a-priori knowledge about all system activities may be hard or impossible to obtain. Its rigidity enables deterministic behavior, but limits flexibility drastically. The off-line scheduling approach is the one usually associated with time-triggered architectures [BS05].

On-line scheduling, however, takes all decisions at run-time, meaning that the scheduler decides when a new task is released or when a task terminates its execution. Nevertheless, on-line scheduling anomalies have to be handled. Consider for instance tasks sharing resources in a nested way. Due to on-line priority rules, a higher priority task t_i can be blocked by a lower priority task t_j holding a resource. Assume that an independent task t_k is released with an intermediate priority level. Then, task t_k is executed although there exists an available higher priority task. This is known as the *priority inversion phenomenon*.

Thus, off-line scheduling needs a priori knowledge of all task attributes, including release times and precedence relations; whereas in on-line scheduling the attributes of tasks become available only when the tasks are released. In this thesis, we focus on off-line scheduling in order to provide a deterministic and predictable time schedule, which is an on avionic system requirement.

3.1.3 Preemptive vs. Non-Preemptive Scheduling

A preemptive scheduler can interrupt a task execution, when some higher-priority task needs to be executed, and resume it later, when the higher-priority task terminates. The executions of tasks are interleaved. In contrast, a non-preemptive scheduler will execute a task until the task is completed, regardless of which requests became enabled in the meantime.

Some tasks require explicitly non-preemptive scheduling because of their functionality, for example an interrupt handler that saves the state of the processor. If the application has no restrictions on preemption, it is not trivial to answer which alternative is better. Typically, the maximum response time from all tasks is smaller for an optimal preemptive algorithm than for an optimal non-preemptive one, but the cost of preemption is most of the times ignored in the analysis. The question remains whether the benefit of preemption compensates for the context-switch overhead. However, these scheduling analysis is focused on task scheduling, without including constraints from intertask communication. For simplicity reasons, we focus on non-preemtive scheduling in this thesis. A preemptive scheduling policy can be estimated as well.

3.2 Scheduling for Uni- and Multiprocessor Systems

Scheduling for uni- and multiprocessor systems have been intensively studied for the past decades. This section provides an overview about scheduling algorithms for both, uni-processor and multi-processor systems. These commonly used approaches to real-time scheduling may be used for scheduling tasks on a processor or messages on a network (i.e., traffic or communications scheduling). In the following, we briefly classify the scheduling domains for uni- and multi-processor systems.

3.2.1 Scheduling for Uni-Processor Systems

The scheduling problem for uni-precossor systems can be defined as a set of tasks $T = \{t_1, t_2, ...t_n\}$ that need to be allocated to a uni-processor system such that all constraints are satisfied. An implicit assumption is that a processor can execute at most one task at a time, and vice versa, a task can be processed by at most one processor at a time. In the following, we distinguish between clock-driven and priority-driven approaches.

Clock-driven approaches

Clock-driven approaches can be used, when all tasks are periodic and their release times are known. In this case, a static off-line schedule can be build. Task dependencies are constraints to the scheduler, which have to be met. If all tasks are released synchronously, the schedule is build for the length of a hyperperiod. This concludes in a periodic schedule, which is called *cyclic*

schedule. Generally, the schedules can be constructed with any algorithm, including those from priority-driven scheduling, when the tasks are independent. In case there are any precedence or exclusion constraints, the scheduling problem becomes NP-complete.

Priority-driven approaches

In priority-driven scheduling approaches the scheduler assigns properties to task, which may be fixed or dynamic. This assignment is done at compile time to each task in the task set. The online scheduler maintains an ordered queue of ready tasks and executes the tasks in the order defined by their priority. Priority-driven task scheduling is classified into *fixed-priority* and *dynamic-priority* scheduling. In this section, we briefly describe both domains and name two exemplary algorithms, one for each class.

The *fixed-priority* scheduling (FPS) is also called *strict-priority* scheduling. The *Rate-Monotonic* (RM) algorithm, proposed by [LL73], is an example for a priority-driven algorithm with static-priority assignment in the sense that the priorities of all instances are known before their arrival. The RM-algorithm requires a preemptive scheduler that assigns priorities inverse proportional to the task periods. Thus, a task with a shorter period (i.e., higher request rate) gets a higher priority. The RM algorithm can be used for task sets with any relation between deadlines and periods. However, when $D_i = P_i, \forall\ i = 1, ..., n$, meaning that the relative deadlines of all tasks are equal to their periods, RM is optimal among all fixed-priority algorithms [LL73]. The least upper bound of the processor utilization can be computed by $U_{lub}(n) = n(2^{1/n-1})$, with the limit $\lim_{n\to\infty} = \ln 2 \simeq 0.693$, where n is the number of given tasks [LL73]. The optimality is maintained if the relative deadlines of tasks are less than this limit but proportional to the periods.

Another domain in priority-driven approaches is the *dynamic-priority scheduling* (DPS), where individual instances of the same task may have different priorities at run-time. The *Earliest-Deadline-First* (EDF) approach, proposed by [LL73], is such a priority-driven algorithm. This algorithm chooses at each instant in time that task of the currently active tasks, which has the smallest deadline. Thus, a task instance with an earlier deadline will get a higher priority and is scheduled first. EDF requires a preemptive scheduler and it is called the *deadline-monotonic* (DM) scheduling algorithm. EDF is optimal (compare [Der74]) among all algorithms; therefore, any feasible task set complaining with the given requirements, can be scheduled by EDF. The EDF algorithm remains optimal also in case of hybrid systems, which have both periodic and sporadic tasks [Mok83]. Optimality gets lost, if tasks cannot be preempted.

3.2.2 Scheduling for Multi-Processor Systems

Another interesting line of research in the scheduling context focuses on scheduling task sets for more than one processor. Scheduling multi-processor systems can be defined by a set of tasks $T = \{t_1, t_2, ..., t_n\}$ and a set of processors $P = \{p_1, p_2, ...p_n\}$. In that context, scheduling algorithms define the assignment of tasks from T to processors from P, such that all tasks meet their

constraints. Dhall and Liu [SL78] distinguish between two general approaches to scheduling periodic task sets on multiprocessor system: global scheduling vs. partitioning. In the following the different approaches are briefly described.

Global Scheduling

Global scheduling algorithms store the tasks waiting for execution in one queue, which is shared among all processors of P. Practically, the problem can be described as follows: In a system consisting of n processors, the processors select the n highest priority tasks of the queue at every moment. As a consequence, each processor has to maintain the tables necessary for multi-processor scheduling algorithms. However, special algorithms need to be developed, because Mok and Dertouzos[MD78] has proven that scheduling algorithms that are optimal for uniprocessor systems are no longer optimal for multiprocessor systems.

Partitioning

Partitioning paradigm provides a diversification of tasks, such that all tasks in a partition are assigned to the same processor. Tasks are not allowed to migrate. Thus, partitioning has the advantage of reusing the well known results from uniprocessor scheduling. However, partitioning has negative consequences. Finding a minimal schedule for a given set of tasks in a multiprocessor system has proven to be a NP-hard problem [LW82]. Therefore, heuristic algorithms are used. However, these algorithms do not guarantee an optimal allocation. The *First Fit* (FF) and *Best Fit* (BF) are examples for heuristic algorithms.

3.3 Scheduling for Distributed Systems

Distributed systems are uniprocessor systems that exchange data over networks. There is no shared memory. Usually, a task gets its input message at its beginning and sends its output messages at its end. In a systems view, applications consist of a number of communication tasks with precedence constraints, that is, constraints specifying their execution ordering. A task is activated when all its input messages have been delivered by the communication interface. The set of tasks is allocated to a set of computing resources and communicate via message passing over the shared communication resource.

Therefore, a scheduling policy has to take into account not only the constraints imposed by the applications (e.g. precedence relations of the application tasks) but also the characteristics and efficient usage of the underlying communication system. Message to slot allocation cannot be accurately computed before a task schedule is computed. If the task scheduling problem and the message scheduling problem are regarded independently, further timing inconsistencies can arise. A task, for instance, never receives the actual correct value, because the message slot

containing that information is allocated after the starting time of the task. These inconsistencies are getting worse by increasing the number of tasks in a given application and thus the complexity of the precedence graph.

In this thesis, the problems of task scheduling and message scheduling are regarded in an integrated way. We focus on a set of processors P, which are mapped on a set of distributed communication resources N, communicating over a shared resource (communication medium). We focus only on clock-driven protocols (e.g. FlexRay [Con05]), in contrast to fixed-priority protocols (e.g. CAN [fS98]) or token-passing protocols (e.g. FDDI [FDD89], or PROFIBUS [Com99]). Therefore, the shared communication resource is implemented as a time-triggered network. For such a system composed of a number of communicating applications on a distributed network connected by a time-triggered bus, effective configuration is needed to guarantee functional and non-functional avionics system requirements [PRS08]. Several related research approaches that deal with this problem of combined task and message scheduling, are presented in section 3.4.

3.4 Related Research

Task scheduling for various kinds of systems has been intensively studied in literature. Well-known preemptive and non-preemptive task scheduling approaches do not take into consideration bus-related communication aspects. Specific issues, such as communication protocols, assignment of messages to slots, etc. are not addressed. These aspects are, however, highly necessary in (hard) real-time systems configuration. Therefore, an effective scheduling policy for TDMA-based avionics applications needs to consider an integrated task and message scheduling. Related approaches considering this integrated view are reviewed in the following.

3.4.1 Holistic Schedulability Analysis

Tindell and Clark [TBW94] provide a holistic scheduling technique. Based upon a distributed real-time system, where fixed-priority tasks with arbitrary deadlines communicate by message passing with a simple TDMA protocol. Tindell et al. assumes that all CPUs and networks are scheduled according to a fixed-priority policy for tasks with arbitrary deadlines. Furthermore, if a task t_2 needs the output of task t_1 and they are on different processors, then a delay is identified, because task t_1 will produce and send a message m_1, that produces a release jitter for task t_2. The message m_1 itself has a release jitter resulting from time variations in the producer task. Thus, the receiver task t_2 inherits the release jitter from the message, because it is released when the message is delivered. Thus, Tindell and Clark use the schedulability analysis for fixed-priority task with arbitrary deadlines to determine the worst-case response times of messages sent between related task sets.

The holistic approach uses the response-time analysis and end-to-end deadlines to compute the worst-case response time of the distributed task set and messages communicating between these

tasks. As the equations for task and message scheduling depend on each other, this approach is called holistic and is solved by forming recurrence relations. For further details, compare Tindell et al. [TBW94].

3.4.2 Combined Task and Message Scheduling using Branch-and-Bound

The algorithm provided by Abdelzaher and Shin [AS99] uses a branch-and-bound technique a combined task and message schedule in distributed real-time systems. The algorithm provides an off-line scheduling of communication tasks with precedence and exclusion constraints. Tasks are assumed to communicate via message passing based on a time-triggered real-time channel.

As a basis, the algorithms use a given task set, which includes the allocation of task to computing resources, the computation times of each task, the deadline and arrival times, as well as the period. The algorithm analyzes the system within an interval of time equal to the least common multiple (LCM) of all task periods, called the *planning cycle*. This planning cycle is derived by the definition of a *module set*, where a *module* corresponds to a task invocation of a certain task. These modules may have synchronization or mutual exclusion constraints. The approach generates a complete schedule at each vertex in the search tree. Generally, this is done in conceptually two orthogonal dimensions: The first searches the message-priority space, the second searches the space of all possible task schedules for a schedule that minimizes the maximum task lateness. A *message-priority order*, based on the relative deadline calculated for each message is used for message scheduling, and the *Earliest Deadline First* (EDF) algorithm is used for task scheduling. The branch-and-bound technique can be viewed as traversing a search tree, where all vertices are pruned whose lateness of a schedule is higher than an already found schedule. Thus, the algorithm yields a feasible schedule or alternatively proceedes the search tree until an optimal task schedule is found.

Furthermore, a *greedy heuristic* can be used, which performs depth-first search with no backtracking. This would expand each vertex by generating all its children, then branches to the minimum-cost child, ignoring all others. Child vertices, whose schedule lateness is more than that of the parent are pruned.

For a more detailed description, e.g. concerning the branching functionality, compare [AS99].

3.4.3 Combined Task Message Scheduling using Satisfiability Checking

Metzner et al. [MFHS05] introduces a SAT-based approach to the task and message scheduling problem of distributed real-time systems. The approach proposes an optimal strategy to assign task and messages to computing resources (ECUs) and communication resources (network busses) and delivers an optimal allocation respectively.

As a basis, abstract models of the system architecture and the task model, including timing constraints are defined. In order to find an optimal allocation scenario, timing and resource

restrictions are modeled as a set of integer formulae. This approach uses the deadline monotonic algorithm with preemption (a *preemptive, fixed-priority algorithm*) to schedule the set of tasks. A cost function is added, which reflects the runtime overhead of a certain task allocation for both, memory allocation of certain tasks set to a computing resource, as well as worst-case execution time including possible preemption cost on a processor. The given approach is based on the transformation of this problem into nonlinear integer optimization problems, solved by an appropriate propositional SAT checker (compare [FH03]). For each given allocation, scheduling analysis evaluates whether all tasks can meet their timing constraints by calculating the response times of all task chains. Afterwards, a binary search is used in order to find the optimal solution.

3.4.4 Scheduling Multi-Mode Real-Time Distributed Components

Farcas et al. [Far06] introduces algorithms for automatic schedule generation for task and message scheduling. This approach is based on the so-called Logical Execution Time (LET) abstraction (cf. figure 3.1), which abstracts from the physical execution time on a particular platform and thereby abstracts from both the underlying execution platform and the communication topology. The LET forms the basis for component-oriented development of real-time systems. Languages such as Giotto [HHM+, HHK01b, HHK01a] and the Timing Definition Language (TDL) harness the LET abstractions.

The *Logical Execution Time* (LET) conforms to a timed model, where all computational activities and communications logically consume a fixed amount of time, regardless of whether they actually need less time to execute. The basic idea is, that a given program will produce the output exactly at the required response time, even if the task execution is completed earlier. Logically, the previous value remains unchanged, until it is updated with the new one. This concept provides determinism and predictability as the behavior of the program does not depend on the platform but only on the task properties and the environment. Pree and Templ [PT08] describe this abstraction from a time-triggered platform, using the FlexRay protocol.

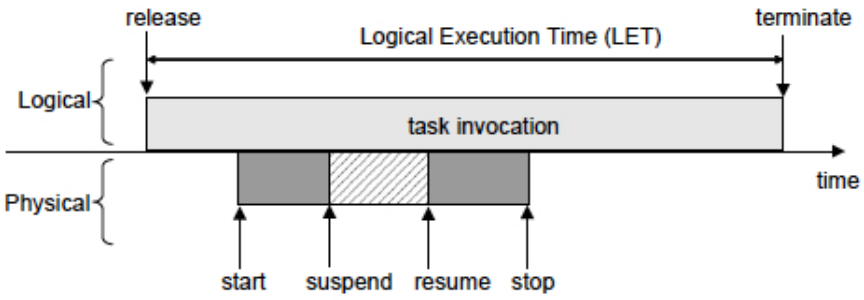


Figure 3.1: Logical Execution Time (LET)

This approach is based on the TDL component model, which supports the decomposition of hard real-time applications into modules that are executed logically in parallel. The scheduling is the

backbone to maintain the concepts of LET and transparent distribution. For processor scheduling algorithms are introduced for generating clock-driven schedules. For communication scheduling algorithms are introduced, which identify the required messages, assign timing constraints, map messages to frames, and schedule them. This approach is based on traditional hyperperiod scheduling.

The task and message schedules are generated in two steps. First of all, the messages and afterwards the tasks are scheduled with deadline constraints from the bus schedule. Then, a deadline corresponds to the situation where a producer task have to finish before the corresponding message is sent on the bus. The strategy is to schedule the message as late as possible. For task scheduling the Earliest Deadline First (EDF) with precedence constraint algorithm is used [But04]. For message scheduling, an heuristic algorithm, adapted from Reversed EDF, also called Latest Release Time (LRT) [Liu00] is used. Further details can be found in [FFPT05].

However, using the LET abstraction and the provided concepts for component-oriented development of real-time systems, several drawbacks have to be taken into account. The basic assumption of the LET concept is that the platform, run-time system, and the scheduling mechanisms used for the physical execution allow each task to complete before the end of its LET. This, however, introduces a so-called *unit-delay*, because dependent tasks exchange information only at LET boundaries. In the context of scheduling for (hard) real-time aeronautic system, as it is investigated in this thesis, this means that a calculated communication period has at least the length of the *least common multiples* of all all task periods. Given a certain precedence graph $\mathcal{G}$, the end to end latency of this precedence graph could be even longer because the *logical execution time* is equal to the task period. Thus, the LET concept does not appear to be a feasible solution for the given scheduling problem.

3.4.5 An Improved Scheduling Technique for Time-Triggered Embedded Systems

Pop, Eles and Peng [PEP99] provide an improved scheduling technique for scheduling synthesis. Based on developed older approaches ([DE98] and [EKP+98]), where the worst case delay for a system execution was derived by a function depending only on the amount of date exchanged by the processes, the new approach is based on the concrete underlaying TDMA bus system (TTP Protocol). The given approach produces a schedule table that contains both the flow of data (communication on the TDMA bus system) and that of control (task execution times of the application).

Therefore, the *abstract system model* is given by a directed, acyclic graph with conditional edges. It captures both the process graph description, highlighting the processes (tasks) and their dependencies (represented by a graph node), as well as communication messages, which are represented by a conditional edge. Furthermore, the notation of *disjunction node* and *conjunction node* is introduced. Further details can be found in [PEP99].

A priority based schema is used to decide which processes or tasks are extracted in order to be scheduled at a given time. This priority scheme depends on the *Partial Critical Path* function, which includes knowledge of the bus access scheme into the priority function. Based on this first schedule, the schedule is then improved by using a proposed heuristic to determine an ordering of the slots an the slot lengths so that the execution delay of the application is as small as possible. Starting from the first slot, the given schedule is improved, by successively trying every not yet allocated node to that slot, in order to minimize the overall system worst case delay. Thus, for each candidate node, the schedule length is calculated. This is done, as well, for every possible slot length, to obtain even better results.

3.4.6 Optimal Task Graph Scheduling with Binary Decision Diagrams

Precedence task graph scheduling is considered by Jensen, Lauritzen and Laursen [JLL04]. The given approach solves the task graph scheduling problem with uniform processors and arbitrary task execution times by using BDDs for representing the task graph scheduling problem. A breadth-first search and an A*-based algorithm is employed for finding optimal schedules.

This approach is based on the task graph scheduling problem, which can be given by a directed acyclic graph of dependencies between tasks with arbitrary execution times and an arbitrary number of available homogenous processors. The task graph scheduling problem is represented as a BDD state space exploration problem and approached by a breadth-first and an A* search algorithm for finding optimal schedules. The BDD algorithm used can be described by three major phases: The first phase generates the states with possible combinations of tasks that can be started, the second phase runs them one step and the last phase stops them if they have finished [JLL04]. Thus, the given task precedence graph is traversed according to their given dependencies.

Furthermore, Jensen, Lauritzen and Laursen provide a guided search, in order to prevent from searching exhaustively for an optimal schedule configuration. Therefore, a cost function is provided, which estimates the number of free time slots at the next step of the schedule as an estimate of the remaining free slots. A free slot can be seen as a time interval of a processor, which is not used during this interval. As the approach tries to find the optimal schedule, which is the schedule with the fewest possible free time slots, this cost weight estimation is used as a lower bound on the total cost of the set of states, which means that promising states with a lower cost can be examined first.

The representation of tasks is similar to the presented *Symbolic Task and Message Scheduling* approach, however, they do not consider underlying network communication aspects, and message scheduling.

Chapter 4

Algorithm for Integrated Task and Message Scheduling

This chapter presents an off-line scheduling algorithm for the task and message scheduling problem based on precedence graph traversal. We propose an approach that integrates task scheduling at system level with message scheduling at communication level. This algorithm augments conventional scheduling rules with algorithms addressing the specific problems of scheduling messages on time-triggered communication busses. The algorithm allows to automatically compute schedules even for large aeronautic applications.

4.1 Functional description of new scheduling algorithm

Our approach relies on an off-line scheduling algorithm for traversing a precedence graph $\mathcal{G}$that computes an integrated task and message schedule. Thus, on the one hand, the order of tasks, defined by the precedence graph relations has to be considered. On the other hand, a message schedule has to be defined, which assigns messages to specific time slots that are used to transmit the information over the time-triggered bus. On system level, the task scheduling problem is represented by a directed acyclic weighted graph, as described in section 2.3.

In general, the algorithm starts by using a weighted precedence graph as an input. Starting at the sink vertex the precedence graph is traversed backwards along the longest path through the precedence graph. In each step of this graph traversal, whenever a new task on the longest path is reached, the algorithm searches for both successor and predecessor tasks according to the task ordering. In order to minimize end-to-end latency, messages are allocated to the next available time slot on the communication bus, with the minimal intertask communication. This slot is known by the application due to the synchronization of application and communication bus. By reaching the sink vertex of the precedence graph an integrated task schedule has been found.

4.1.1 Calculation of longest path

As a first step the longest path through the precedence graph $\mathcal{G}$ is calculated. This is done with respect to the task length or computation duration $comp(t_i)$ of a certain task t_i and the duration of a message send by task t_i to the successor task t_j. Using these parameters an off-line calculation of the longest path lP can be done. This calculation is done by using the depth-first search (DFS) algorithm [CLRS01] on the given precedence graph $\mathcal{G}$ for the calculation of the longest path $|lP|$.

The longest path lP is stored in a linked list of task elements $t_i \in T$, storing the task sequence that incorporates all tasks on the longest path length lP. In the following example we introduce a precedence graph highlighting the benefits of a linked list in that context.

Example 2 We consider a set of three tasks $T = \{t_1, t_2, t_3\}$ and a set of messages $M = \{m_1, m_2, m_3\}$. Furthermore, $\tau(t_1) = \{m_1, m_3\}$, $\tau(t_2) = \{m_2\}$ and $\eta(n_i) = t_i$, $\forall\ i \in \{1, 2, 3\}$. The function $\mathcal{D}$ is given as $\mathcal{D}(t_1,m_1) = t_2$, $\mathcal{D}(t_1,m_3) = t_3$, $\mathcal{D}(t_2,m_2) = t_3$. Consider the longest path to be determined as the task sequence: lP= [t_1, t_2, t_3]. In this case, both task t_1 and task t_2 are predecessors of task t_3 and are both on the longest path lP, as illustrated in figure 4.1. Because the algorithm traverses backwards along the longest path, the selection of the next task to be considered remains on the linked list. In this case t_2 would be the next task to be considered, since this task is the direct predecessor of t_3 according to the longest path list.

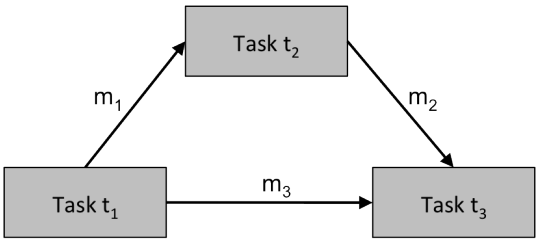


Figure 4.1: Simple Precedence Graph $\mathcal{G}$

4.1.2 Initial starting point

The initial starting point of the algorithm is given by the sink vertex (task t_{sink}). In case the precedence graph $\mathcal{G}$ consists of multiple sink vertices, e.g an aeronautic system with a server sending its information to multiple actuators, we introduce a sink dummy task t_{dummy}. This dummy task t_{dummy} serves as a successor task for all sink vertices. Thus, task t_{dummy} takes a virtual input message m_v from all sink vertices as an input. A virtual input message m_v can be described by a message that consumes no time. The dummy task itself consumes no time as well. In case there are precedence graphs with multiple source vertices, the same introduction of task t_{dummy} is done for these source tasks, respectively. The following example illustrates the scenario of multiple sink vertices.

Example 3 We consider a set of four tasks $T = \{t_1, t_2, t_3, t_4\}$ and a set of messages $M = \{m_1, m_2, m_3\}$. Furthermore, $\tau(t_1) = \{m_1, m_2, m_3\}$ and $\eta(n_i) = t_i$, $\forall\ i \in \{1, 2, 3, 4\}$. The function $\mathcal{D}$ is given as $\mathcal{D}(t_1,m_1) = t_2$, $\mathcal{D}(t_1,m_2) = t_3$, $\mathcal{D}(t_1,m_3) = t_4$. All sink vertices, namely task t_2, task t_3 and task t_4 are connected to the dummy task, t_{dummy}. The corresponding edges are allocated with a weight of 0. In this case, we consider task $t_{sink} = t_{dummy}$. For instance, the longest path lP can be considered as the task sequence: $lP = [t_1, t_2, t_{dummy}]$.

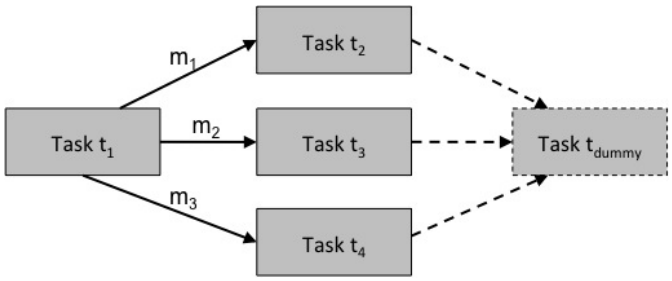


Figure 4.2: Integration of a dummy task t_{dummy} in a precedence graph

The integration of dummy tasks, either as a source or a sink task, enables the calculation of the longest path lP for a given precedence graph $\mathcal{G}$, that might have multiple source or sink tasks.

4.1.3 Precedence Graph Traversal

Starting from the sink vertex (task t_{sink}), the algorithm incrementally traverses backwards along the calculated longest path lP until the source vertex (task t_{source}) is reached. This precedence graph traversal is done iteratively.

CALCULATE_TASK_SCHEDULE ($\mathcal{G}$)

The basic algorithm, called `Calculate_Task_Schedule`, takes a weighted precedence graph $\mathcal{G}$ as input. Several global variables are accessible: the set of messages M, tasks T, as well as the functions τ, η, σ and the corresponding destination functions $\mathcal{D}$. The current time ct is a global variable as well.

In a first step (line 1) the longest path lP through the precedence graph $\mathcal{G}$ is calculated. The sink vertex t_{sink} in the precedence graph ordering is allocated to the end of the communication cycle and its starting time is calculated by subtracting its computation time $c(t_{sink})$ as $s(t_{sink}) = end\_of\_cycle - c(t_{sink})$ (cf. line (2)). The traversal of the precedence graph starts by calling a simple for-loop (line 3). Then the backward traversal starts by following the calculated longest path lP until the source vertex (task t_{source}) is reached. Whenever a new task t_i on the longest path $lP$is reached, the algorithm searches for both output precedence relations (lines (4)-(6)) and for input precedence relations (lines (7)-(9)).

Algorithm 1 Calculate Task Schedule ($\mathcal{G}$)

1: $lP \leftarrow CALCULATE\_LONGEST\_PATH(\mathcal{G})$
2: $s(t_{sink}) = ct$ = *end_of_cycle* - $c(t_{sink})$
3: **for** each vertex $t_i \in \mathcal{G}$, starting from t_{sink} to t_{source} along lP **do**
4: **if** there exists $t_j \in \mathcal{G}$ where $t_i \prec t_j$ **then**
5: $SET\_OUTPUT\_TASK(t_i, t_i)$
6: **end if**
7: **if** there exists $t_j \in \mathcal{G}$ where $t_j \prec t_i$ **then**
8: $SET\_INPUT\_TASK(ct, t_i)$
9: **end if**
10: **end for**

Depending on whether output or input precedence relations are found the following two functions are called: For input precedence relations (SET_INPUT_TASK) and for output precedence relations (SET_OUTPUT_TASK), respectively. These functions are described in the following.

SET_INPUT_TASK

Starting from a certain task $t_{receive}$, in a first step, all input precedence relations are investigated to $t_{receive}$ are computed (cf. line (2)). These tasks are called the set A. As long as the certain task $t_{receive}$ is not the sink tasks there exists a set of input tasks relations A. At least one task of the task set A corresponds to the longest path lP. This task is called task t_{send} and selected first (line (3)). In case there are multiple longest paths in a given precedence graph, the predecessor of task $t_{receive}$ is chosen non-deterministically. Then, all messages M_i that comply to $\mathcal{D}(t_{send},M_i)$ = $t_{receive}$ are schedules, using the MSG_TO_SLOT function (cf. line (4)). After scheduling the predecessors on the longest path, all remaining predecessors in A are scheduled (line (5)-(7)).

1: **function** $SET\_INPUT\_TASK(ct, t_{receive})$
2: $A = \{t_1, ..., t_n \mid t_i \prec t_{receive}, \forall i \in \{1, ..., n\}\}$
3: $t_{send} = t_i$ with $t_i \in A \cap lP$
4: $MSG\_TO\_SLOT(t_{receive}, t_{send})$
5: **for** each $t_i \in A \setminus t_{send}$ **do**
6: $MSG\_TO\_SLOT(t_{receive}, t_i)$
7: **end for**
8: $ct = s(t_{send})$
9: **end function**

The MSG_TO_SLOT function assigns messages send from a given task t_i to $t_{receive}$. This function is described in the following.

MSG_TO_SLOT

The function MSG_TO_SLOTS($t_{receive}$, t_{send}) is responsible for the allocation of all messages $m_i \in M_i$, that are sent from task t_{send} to task $t_{receive}$, to slots $sl_i \in SL$. These messages are computed by the destination function $\mathcal{D}(t_{send},M_i) = t_{receive}$. In the following this set of messages is called B (cf. line (3)). In order to schedule the message, the next available time slot sl_i needs to be identified. Therefore, the current time basis ct is updated by assigning it to the starting time of the receiving task $t_{receive}$. As all slots are allocated to time slots, using the minimal distance $dt_{t_{send},t_{receive}}$ to their corresponding sending and receiving tasks, the next available slot sl_i can easily be chosen (line (5)). In a next step the assignment of message m_i to the slot sl_i is done, cf. line (6). $start\_slot(sl_i)$ indicated the start time of the assigned time slot, which in turn enables for calculating the starting time of the sending task $s(t_{send})$, as depited in line (7).

1: **function** $MSG\_TO\_SLOT(t_{receive}, t_{send})$
2: $ct = s(t_{receive})$
3: $B \subseteq M = \{m_1, ..., m_o \mid m_i \in \tau(t)$ and $\mathcal{D}(t, m_i) = t_{receive}\}$
4: **for** each $m \in B$ **do**
5: Choose $sl_i \in SL$ such that $[ct - (start\_slot(sl_i) + duration\_slot(sl_i))]$ minimal
6: $\sigma(m_i) = sl_i$
7: $s(t_{send}) = start\_slot(sl_i) - c(t_{send})$
8: **end for**
9: **end function**

SET_OUTPUT_TASK

After allocating all incoming messages by the function SET_INPUT_TASK, all outgoing messages need to be scheduled as well. Therefore, all direct successors of a task t_{send} (corresponds to the receiving task $t_{receive}$ in MSG_TO_SLOT-function) are investigated as well. A direct successor of a certain task t_{send} corresponds to a partial precedence graph and is therefore called a *branch*.

First, all branches from a certain task t_{send} are determined (cf. line (2)). This set of tasks is called C. In a next step, for each task t_i in C, we check, whether the successor task t_i has already an assigned starting time, namely has already been scheduled or not (cf. line (4)). In case the starting time has not been calculated, the function MSG_TO_SLOTS_OUT is called to schedule all messages M_j, corresponding to $\mathcal{D}(t_{send},M_j) = t_i$.

When allocating all messages M_j from $\mathcal{D}(t_{send}, M_j) = t_i$, time consistency needs to be checked, because the algorithm might generate a schedule that does not fulfill the timing given by the precedence relations. In order to guarantee that all precedence relations, defined by the precedence graph, are in a consistent chronology, the CONSISTENCY_CHECK function is called. This function checks, whether all successor tasks in C fit according to the precedence relation in $\mathcal{G}$(cf. line (6)). In case of inconsistencies, a recalculation is done by the RECALCULATE_SRC_TASK function (line (7)). This recalculation is limited to that branch, where timing inconsistencies are detected (a detailed description is given in section 4.1.4). A branch starts always from a task on

the longest path. This task is called task t_{fix}. Thus, the entry point to that certain branch, namely the starting time s(t_{fix}) of that task on the longest path t_{fix}, is recalculated.

```
 1: function SET_OUTPUT_TASK(t_fix, t_send)
 2:     C = {t_i | t_send ≺ t_i, ∀i ∈ {1, ..., n}}
 3:     for each task t_i ∈ C do
 4:         if s(t_i) exists then
 5:             MSG_TO_SLOT_OUT(t_i, t_send)
 6:             if (CONSISTENCY_CHECK(t_receive) == false) then
 7:                 RECALCULATE_SRC_TASK(t_fix)
 8:             end if
 9:         else
10:             MSG_TO_SLOT_OUT(t_i, t_send)
11:         end if
12:     end for
13: end function
```

In case the receiving task t_i is not set (its starting time is not yet calculated, cf. line (10)), the function `MSG_TO_SLOTS_OUT` is called. This function schedules all messages sent by task t_{send} to task t_i. In this case the `SET_OUTPUT_TASK` function is called, with the new successor task t_i, until a task is reached, whose starting time is already calculated(cf. line (10)).

MSG_TO_SLOT_OUT

```
 1: function MSG_TO_SLOTS_OUT(t_receive, t_send, M, σ)
 2:     ct = f(t_send)
 3:     B ⊆ M = {m_1, ..., m_o | m_i ∈ τ(t_send) and D(t_send, M_k) = t_receive}
 4:     for each m ∈ B do
 5:         if σ(m_i) = ∅ then
 6:             Choose sl_i ∈ SL such that
 7:               [(start_slot(sl_i) − ct)] minimal
 8:             σ(m_i) = sl_i
 9:         end if
10:         if s(t_receive) not exists then
11:             s(t_receive) = end_slot(σ(m_i))
12:         end if
13:     end for
14: end function
```

The `MSG_TO_SLOT_OUT` function identifies all messages M_k send by the sending task t_{send}, corresponding to $\mathcal{D}(t_{send}, M_k) = t_{receive}$. This set of messages is called B (cf. line (3)). For each message $m_i \in B$ is checked, whether the message is already allocated to a certain slot ($\sigma(m_i) = \{sl_i\}$) (cf. line (5)). If this is not the case, the allocation is done on the next available slot with a minimal distance $dt_{t_{send},t_{receive}}$ after $f(t_{send})$. This is done for all messages in B, as illustrated in lines (5) - (9). The `MSG_TO_SLOT_OUT` function furthermore checks, whether task $t_{receive}$ has already an assigned staring time (line (10)). In case there is no starting time

allocated to the task $t_{receive}$, the task can be scheduled right after the dedicated slot, namely $end\_slot(\sigma(m_i))$.

CONSISTENCY_CHECK

Time consistency is analyzed, after allocating all messages in SET_OUTPUT_TASK (cf. line (2)). This is done, because the SET_OUTPUT_TASK function allocates all successor tasks that have not been scheduled yet, until an already scheduled task is found on that branch. Time consistency, in this context, checks whether all precedence relations are satisfied. This is done by comparing the finishing time of the last allocated slot ($end\_slot(sl_i)$) with the starting time of the already calculated receiving task s($t_{receive}$). If $end\_slot(sl_i) \leq s(t_{receive})$ is true, a consistent chronology is guaranteed. Otherwise, the SET_OUTPUT_TASK function needs to recalculate the starting time of that task, which initializes the branch in the precedence graph $\mathcal{G}$, that fails the consistency check. This recalculation and path extension is described in the following.

1: **function** $CONSISTENCY\_CHECK(t_{receive})$
2: **if** $end\_slot(sl_i) \leq s(t_{receive})$ **then return** true;
3: **elsereturn** false;
4: **end if**
5: **end function**

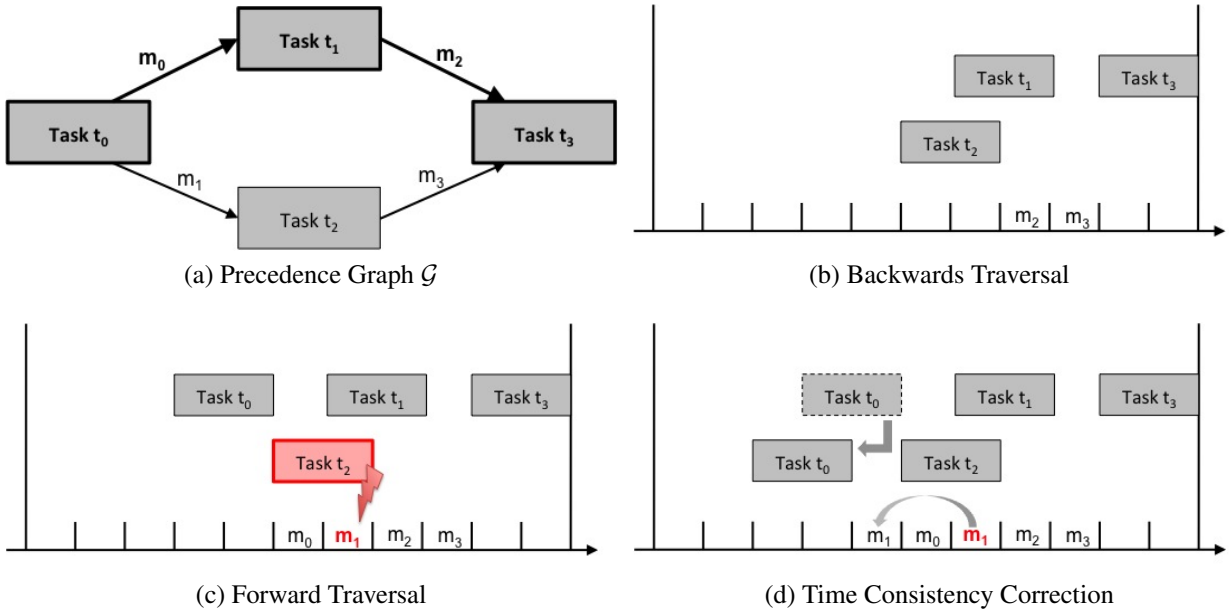
4.1.4 Backtracking and path extension

The presented algorithm is designed to start at the sink vertex (task t_{sink}) of a given precedence graph $\mathcal{G}$. While incrementally traversing backwards along the calculated longest path lP, starting times are allocated to tasks and messages to slots. This is done iteratively using the described graph traversal. As described in the previous sections, timing inconsistencies might occur. Therefore, the CONSISTENCY_CHECK function checks, whether time consistency can be guaranteed. In case of inconsistencies, a recalculation of precedence graph branches becomes necessary.

In order to highlight the process of time consistency check and potential recalculation of already calculated starting times and message slots, we consider the following example:

Example 4 Given a set of four tasks $T = \{t_0, t_1, t_2, t_3\}$, and a set of messages $M = \{m_0, m_1, m_2, m_3\}$. Furthermore, $\tau(t_0) = \{m_0, m_3\}$, $\tau(t_1) = \{m_1\}$, $\tau(t_2) = \{m_2\}$ and $\tau(t_3) = \{\}$. The task to node allocation set is given by $\eta(n_i) = t_i$, forall $i \in \{0, 1, 2, 3\}$. The destination function is given as $\mathcal{D}(t_0,m_0) = t_1$, $\mathcal{D}(t_0,m_3) = t_2$, $\mathcal{D}(t_1,m_1) = t_3$ and $\mathcal{D}(t_2,m_2) = t_3$. We consider the longest path lP as the task sequence: lP= [t_0, t_1, t_3]. The precedence graph is depicted in figure 4.3a.

Figure 4.3b highlights the situation in which - starting from the sink task t_3 - all incoming messages, namely message m_1 and m_2 are scheduled with a prioritization of messages on the longest path. Furthermore, the corresponding sending tasks (t_1 and t_2) are scheduled. According to the



(a) Precedence Graph $\mathcal{G}$

(b) Backwards Traversal

(c) Forward Traversal

(d) Time Consistency Correction

Figure 4.3: Precedence Graph and Graphical Representation of the Algorithm

described algorithm, the precedence graph is traversed along the longest path lP. Figure 4.3c shows that all tasks on the longest path have already been scheduled. As task t_0 is described by $\mathcal{D}(t_0,m_3) = t_2$ the algorithm uses the `SET_OUTPUT_TASK` function to schedule message m_3 as described in section 4.1.3.

By using the `SET_OUTPUT_TASK`-function, a consistent chronology needs to be checked and guaranteed. Time consistency is checked by comparing starting time of tasks, which are set by backwards traversing with message slots allocated by a branch forward traversal. In this simple example, figure 4.3c, highlights the time inconsistency. By comparing the finishing time of message m_3 to the starting time of task s(t_2), according to $\mathcal{D}(t_0,m_3) = t_2$, message m_3 should be finished before task t_2 is started. This check fails.

In order to resolve this situation a recalculation of starting times becomes necessary. This recalculation is bounded to the precedence graph branch, which fails the consistency check function. The entry point, namely the task on the longest path lP that serves as a starting point for that branch is identified. In the given example, task t_0 is the calculated entry point. The recalculation is performed on this entry point task: $s(t_0)$ is incrementally decreased by one time unit, as shown in figure 4.3d. As a consequence, each successor task and message on that branch, that has already been scheduled, is rejected. A recalculation starting with the new starting time of t_0 (entry point task) is done. This recalculation might be repeated several times, until a consistent chronology is guaranteed. In the given example, a decrease of one single time unit is sufficient to guarantee consistent chronology.

Such a recalculation becomes necessary for every branch that fails the CONSISTENCY_CHECK-function.

4.2 Discussion

4.2.1 Cases of guaranteed optimality

The algorithm calculates a schedule γ with respect to the minimal inter-task communication time. This schedule, described by $\gamma = \{t_i \mapsto \gamma_i | \forall t_i \in T\}$, is the solution to the task and message scheduling problem. As a consequence, the length of a schedule $|\gamma|$ can be defined by the following equation (4.1).

$$|\gamma| = \max_{\forall\, t_i \in T} f(t_i) - \min_{\forall\, t_i \in T} s(t_i) \tag{4.1}$$

In each step of the algorithms' graph traversal, whenever a new task on the longest path is reached, the algorithm searches for both successor and predecessor tasks according to the task ordering. Messages are allocated to the next available time slot on the communication bus, with the minimal inter-task communication time. From this characteristic and the fact of lP being the longest path in $\mathcal{G}$, it follows:

The length of a feasible schedule $|\gamma|$ has to be at least the length of $|lP|$:

$$|\gamma| \geq |lP| \tag{4.2}$$

In order to obtain a schedule which is optimal in terms of minimizing the end-to-end latency, we are able to define:

A schedule γ is an optimal (minimal end-to-end latency) schedule, iff $|\gamma| = |lP|$, where $|\gamma|$ is the length of the schedule, as defined in equation 4.1, and $|lP|$ is the length of the longest path. However, this optimal schedule might not always exist, e.g. systems, that are characterized by a highly concurrent precedence graph and numerous resource constraints (several tasks are allocated to a single computing resource). Thus, an optimal schedule of these systems might be longer than the longest path $|lP|$. The presented algorithm might calculate such an optimal schedule, because the algorithm is not designed to always find an optimal schedule. Only in one case, namely that the length of a calculated schedule γ corresponds to the length of the longest path lP, it is assured that the algorithm has found an optimal schedule. Other scenarios are considered in the following.

4.2.2 Deviation from expected optimum in worst case scenario

The algorithm returns a feasible schedule γ. If the length of the calculated schedule is larger than the length of the longest path, namely $|\gamma| > |lP|$, the obtained solution might differ from the optimal solution under consideration.

The reason for finding schedules with longer end-to-end latency than the longest path is that the algorithm fails the time consistency check. In case of inconsistencies, a recalculation of precedence graph branches becomes necessary. A recalculation always induces a path extension, as described in section 4.1.4. This incremental increase of schedule length results in the discrepancy between $|\gamma|$ and $|lP|$. However, as the designed algorithm not always delivers a schedule that equals the length of the longest path, the calculated schedule γ, with a length greater than $|lP|$, might be an optimal schedule for the given precedence graph $\mathcal{G}$. Therefore, in case of $|\gamma| > |lP|$, it seems impossible to identify whether the calculated schedule γ is an optimal schedule with respect to minimal end-to-end latency.

However, the maximum deviation can be specified by determining the worst case scenario. In the worst case scenario an optimal solution complies to a schedule length of $|lP|$. The calculated schedule length $|\gamma|$ is greater than the optimal schedule length $|lP|$. Therefore, we define the maximum possible deviation wcd (worst case deviation) as:

$$wcd = |\gamma| - |lP| \tag{4.3}$$

The given scheduling approach scales up very well, even for large aeronautic systems. As stated above, the optimal solution cannot be guaranteed. Nevertheless, if the optimal schedule length is unknown, each calculated schedule $|\gamma|$ might be the optimal solution under consideration. However, the maximum deviation, given by wcd, indicates the worst deviation from a potential optimum.

Chapter 5

Symbolic Task and Message Scheduling

This chapter describes how we adopt the principle of symbolic state space exploration to the task and message scheduling problem. We propose Symbolic Task and Message Scheduling as a novel approach to the problem of task and message scheduling for TDMA-based avionics applications. This approach allows to automatically compute schedulers with minimal end-to-end latency.

5.1 Basic idea of using model checking for solving scheduling problems

The framework of model checking is used to carry out verification, which is performed on finite-state space. In particular, we use state-of-the art model checking and bounded model checking techniques to compute the schedules. This approach extends ongoing research in task and message scheduling based on time-triggered shared resources by first using model checking techniques for solving this kind of problems. This thesis proves that state-of-the-art model checking and bounded model checking techniques can be used to compute a schedule that fulfills certain system requirements.

Our approach relies on a symbolic encoding of the task and message scheduling problem which guarantees an optimal solution of the given scheduling problem. The symbolic encoding is done by transferring and defining the task and message scheduling problem as a finite-state model checking problem $\mathcal{M} \models \varphi$, where $\mathcal{M}$ is the transition system representing the scheduling problem, while the temporal logic formula φ expresses the desired scheduling property, such as minimal end-to-end latency. We define a basic model $\mathcal{M}$, containing all state variables (e.g. for tasks, messages, nodes, etc.) and a set of transitions. The initial state is specified as a situation in which all tasks are not started and not finished. The goal states are those states where all tasks have finished and all messages have been sent.

The proposed approach is working by constructing gradually a schedule, beginning in a state where no task has started and no messages have been sent on the bus yet, and proceeding one step at a time assigning starting times to tasks and slot positions to messages. This construction process is mapped onto a set of transitions, specified as guarded commands.

The precedence graph is traversed from source to sink. Whenever the precedence graph allows for different possible solutions, characterized by concurrent access to a shared resource (either different messages trying to allocate the next available time slot on the communication bus or several tasks waiting for access to the shared computing resource), we use the model checker's capabilities to explore all interleaved possibilities.

The model checker analyzes the specified task and message scheduling model $\mathcal{M}$ with respect to a given property φ stating that there is no possible schedule that fulfills the given property (system requirement). The model checker returns either *verified* or *falsified*, depending whether the given property is fulfilled by the model or not. In the latter case, the model checkers usually outputs a counterexample from which a task and message schedule can directly be obtained. To find such a counterexample we can either use SAL's bounded model checker [dMRS03], sal-bmc, or one of SAL's symbolic model checkers, sal-smc or, sal-wmc [SS03].

The obtained schedule might not be optimal, with respect to end-to-end latency, because the transition system is allowed to take transitions at which no time is consumed. To obtain an optimal task and message schedule, we further use a binary search for finding that solution (cf. section 5.8). Thus, an automatically optimal schedule generation is guaranteed by construction.

Experimental results demonstrate how the latest generation of model checking tools meets the challenges of providing both a convenient modeling language and the performance to solve given scheduling problems. We use SAL (Symbolic Analysis Laboratory, `http://sal.csl.sri.com`) [dMOR+04] from SRI International as a framework for specification and scheduling synthesis. However, the presented approach is not limited to the specific framework of SAL. The presented task and message scheduling model (cf. section 5.3) can be used by different model checking frameworks as well. However, we illustrate a way how the presented approach is translated into SAL specifications (cf. section 5.4).

5.2 Requirements to Symbolic Task and Message Scheduling

Constructing a task and message schedule requires a set of scheduling rules. These scheduling rules are explicitly defined by the set of transitions, stating how a precedence graph is traversed. Moreover, for constructing an optimal task and message schedule, that is to minimize the end-to-end latency, these scheduling rules need to be adapted according to the optimality criteria. This enables to obtain an optimal task and message schedule.

The crucial point in construction is to deal with concurrent precedence graph scenarios. Concurrency can either be caused by a shared computing resource or a communication resource at a

certain point in time. Decisions in concurrent situations have a direct impact on optimality, especially when considering highly concurrent precedence graphs. While constructing a schedule gradually, it cannot be decided which decision might lead to an optimal solution with respect to a given optimality criterion, such as end-to-end latency. Therefore, we use the model checker's capabilities for constructing a task and message schedule to explore all specified interleaved possibilities defined in the given scenarios. In the following we specify these different scenarios:

Scenario 1

Different tasks (task t_i and task t_j) are allocated to the same computing resource ($Node_i$) and are waiting for access. In this situation both interleaved scenarios are investigated in a next state s', namely in Case 1 task t_i is handled prior to task t_j ($s' = t_i \succ t_j$) while in Case 2, task t_j is given priority over t_i ($s' = t_i \prec t_j$).

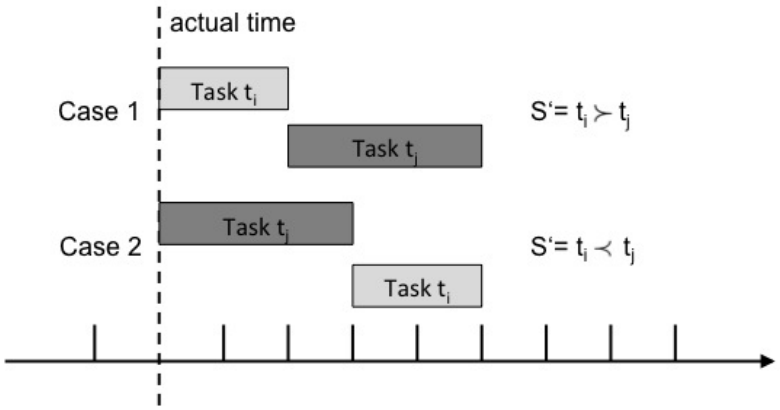


Figure 5.1: Different task trying to allocate to the same computing resource

Scenario 2

Task t_i and task t_j are allocated to the same computing resource, for instance node $Node_i$. A task t_j tries to allocate an already blocked computing resource, which is used by task t_i. Thus, two different options are possible (cf. figure 5.2). Task t_j needs to wait until the computing resource is released (Case 1). On the other hand, task t_j might interrupt the current computing task t_i on node $Node_i$ (Case 2). Subsequently, this interrupted task t_i can be restarted again, because we do not allow preemptive scheduling.

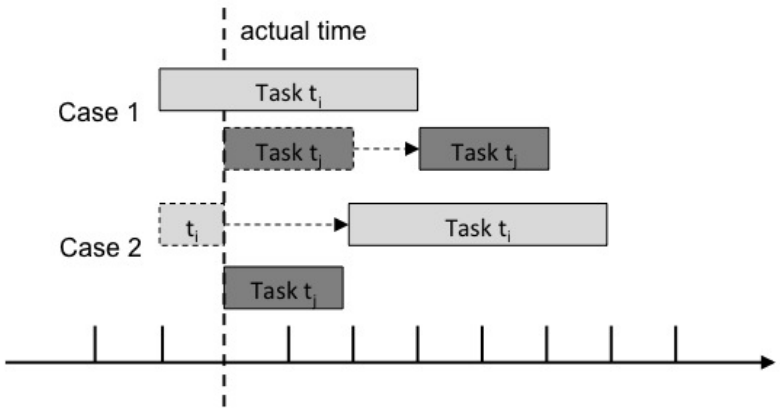


Figure 5.2: Task trying to allocated a already blocked computing resource

Scenario 3

The communication resource (in this thesis a time-triggered communication bus system) might also be the concurrent resource. Scenario 3 describes a situation in which different messages, namely m_i and m_j, are trying to allocate the same sending slot sl_k. It is irrelevant, whether these messages are allocated to identical sending task or to different ones. Obviously, two different options are possible: On the one hand message m_i is scheduled prior to message m_j (Case 1), using the communication slot sl_k ($s' = m_i \prec m_j$). On the other hand message m_j is allocated to the first available communication slot sl_k ($s' = m_i \succ m_j$) (Case 2). This scenario is depicted in figure 5.3.

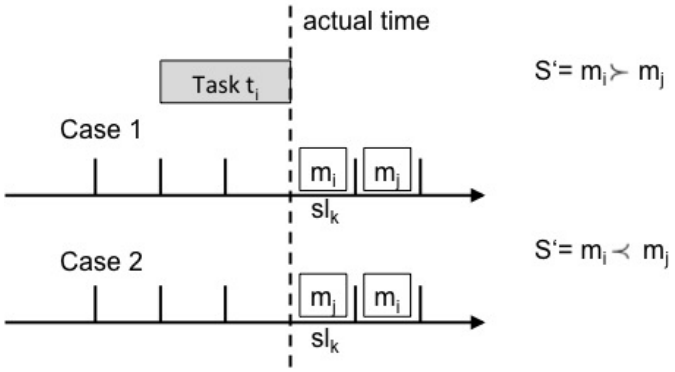


Figure 5.3: Different messages trying to allocate the same communication resource

Scenario 4

The fourth scenario is rather related to the usage of SAL's bounded model checker than to a situation of concurrency. The technique of bounded model checking determines if there is a counterexample of length k to the hypothesis that $\mathcal{M}$ satisfies the property φ. In order to be able

to calculate a counterexamples with the given amount of steps k, the accurate schedule might vary in parameter length l as well. Therefore, we define a scenario, where a message m_i might not be sent in the actual available slot sl_k, but wait for the next slot sl_l. This enables to obtain schedules with variable length l for any given length of the desired counterexample k.

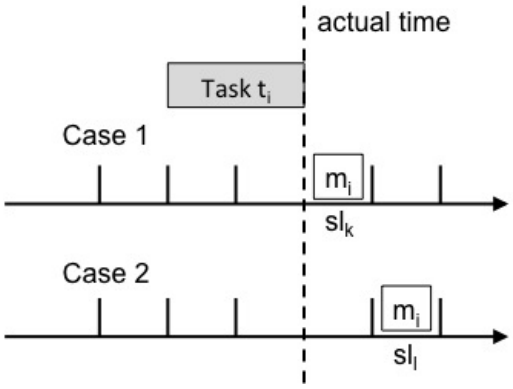


Figure 5.4: Message m_i can be sent in both of the next available slots

5.3 The Task and Message Scheduling Model

We describe the task and message scheduling problem as a $\mathcal{T}$- Program. Therefore, we specify the model $\mathcal{M}$ by the following definition:

Definition 7 (Task and Message Scheduling Model) *A task and message scheduling model is defined as a tuple* $\mathcal{M} = \langle \boldsymbol{V}, \boldsymbol{I}, \boldsymbol{G}, \boldsymbol{T} \rangle$*, where interpretations of the typed variables* $\boldsymbol{V}$ *describe the set of states.* $\boldsymbol{I} \subseteq \boldsymbol{V}$ *is a predicate that describes the initial states,* $\boldsymbol{G}$ *describes the goal states and* $\boldsymbol{T}$ *describes the transition relation between current states and their successor states.*

The model´s variables and parameters **V** are described in section 5.3.1. Interpretations of these typed variables **V** describe the set of states S. The initial state(s) **I** are specified in 5.3.2. A set of transitions **T** is introduced in section 5.3.4.

The SAL framework is used for specification of the task and message scheduling problem. In section 5.4 we will illustrate a way of translating the state space - based on the representation of the scheduling problem to SAL specifications. However, the presented approach is not limited to this specific framework and can be applied to different frameworks as well.

5.3.1 State Representation

The given precedence graph $\mathcal{G}$, as specified in section 2.2 (cf. example 1), comprises several elements needed for the definition of a system state. These elements are a set of tasks T, a set of

messages M, a set of nodes N and a bus B. Thus, we start by defining the system state $\mathbf{V}$ as a conjunction of these elements:

$$\mathbf{V} = T \cup M \cup N \cup B \tag{5.1a}$$

Furthermore, a type declaration for these precedence graph elements is given, (5.2a) - (5.2d). To specify tasks, messages, nodes and the bus, several parameters are needed for the definition of all state variables $\mathbf{V}$ in a system state s_i.

$$T = \{t_{i,started}, t_{i,finished}, t_{i,start}, t_{i,clock}, t_{i,comp}, t_{i,node}\} \tag{5.2a}$$
$$M = \{m_{i,started}, m_{i,set}, m_{i,slot}\} \tag{5.2b}$$
$$N = \{Node_{i,free}, Node_{i,task}\} \tag{5.2c}$$
$$B = \{Bus_{free}\} \tag{5.2d}$$

Each task t_i is represented by two boolean variables $t_{i,started}$ and $t_{i,finished}$ and three integer variables $t_{i,clock}$, $t_{i,comp}$, and $t_{i,start}$ interpreted over $\mathbb{N} \cup \{0\}$, meaning:

$$t_{i,clock} = \{0, 1, 2, ...\} \tag{5.3a}$$
$$t_{i,comp} = \{0, 1, 2, ...\} \tag{5.3b}$$
$$t_{i,start} = \{0, 1, 2, ...\} \tag{5.3c}$$
$$\tag{5.3d}$$

When $t_{i,started}$ is true, task t_i is running and monitors that its execution time $t_{i,clock}$ does not increase the computation time $t_{i,comp}$. The task is finished, when its execution time equals the given computation time. Upon termination the variable $t_{i,finished}$ is set to true. $t_{i,start}$ incorporates the actual starting time of task t_i. $t_{i,node}$, interpreted over the set of nodes N, is defined to hold the node n_i, which the current task t_i is allocated to, e.g, $\eta(n_i) = t_i$.

Each message is represented by two boolean variables $m_{i,started}$ and $m_{i,set}$ and the variable $m_{i,slot}$, which is interpreted over $\mathbb{N}$. When $m_{i,started}$ is true, message m_i is assigned to a certain slot sl_j. This slot sl_j corresponds to $\sigma(m_i)$ and is held by the variable $m_{i,slot}$. $m_{i,set}$ indicates that the message has been scheduled successfully.

Additionally, the variables $Node_{i,free}$, $Node_{i,task}$ and Bus_{free} represent the communication infrastructure. $Node_{i,free}$ denotes the state of the CPU at certain node $Node_i$. Bus_{free} indicates whether the current slot sl_i on the time-triggered bus is already allocated to a node or not. If the CPU of node n_i and the current time slot on the time-triggered bus are not used, $Node_{i,free}$ and Bus_{free} are set to true. The variable $Node_{i,task}$ holds the actual task $t_i \in T$, which currently uses the CPU of node $Node_i$. The $Time$ variable indicates a common understanding of the current time in the system.

5.3.2 Initial State

Having defined all state variables **V**, the initial state **I** (**V**) is specified as follows:

$$\mathbf{I}(\mathbf{V}) = \bigwedge_{t_i \in T,\ m_i \in M,\ Node_{i \in N}} \overline{t_{i,started}} \quad \wedge \quad \overline{t_{i,finished}} \quad \wedge \quad (t_{i,clock} = 0) \tag{5.4a}$$

$$\wedge \quad \overline{m_{i,started}} \quad \wedge \quad \overline{m_{i,set}} \quad \wedge \quad (m_{i,slot} = null) \tag{5.4b}$$

$$\wedge \quad Node_{i,free} \quad \wedge \quad Bus_{free} \quad \wedge \quad Node_{i,task} = null \tag{5.4c}$$

$$\wedge \quad (Time = 0) \tag{5.4d}$$

In the initial state all tasks are not started and not finished. The actual clock computation is set to zero (5.4a). Also all messages are not started and not allocated (5.4b). Each CPU of a node is not used and the current starting slot is free (5.4c).

5.3.3 Goal State

The goal states **G**(**V**) are those states where all tasks have finished and all messages are assigned (5.5a):

$$\mathbf{G}(\mathbf{V}) = \bigwedge_{t_i \in T, m_i \in M} t_{i,finished} \quad \wedge \quad m_{i,set} \tag{5.5a}$$

5.3.4 Transitions

A set of transitions $\mathbf{T} = \{tr_1, tr_2, ..., tr_n\}$, as part of the defined model $\mathcal{M}$, specifies the transition relations between current states s and their successor states s'. A given transition, specified as a guarded command, is invoked if and only if its guard is true. By this requirement the number of possible transition invocations depends on the actual status of a given state s in time. This depends, while constructing a schedule, on the current status of the traversal of the precedence graph. Thus, the selection of possible transitions is limited to the set of active transitions in a certain situation. Active transitions are all transitions whose guards are satisfied in a state s.

In this section we describe the set of transitions **T** specified in the context of symbolic task and message scheduling. We specify six transition relations used to model the problem of integrated task and message scheduling. Four different transitions are needed to define task level transitions: the Start Task Transition (T_{st}), the Run Task Transition (T_{rt}), the End Task Transition (T_{et}) and the Change Task Transition (T_{ct}). Two more transitions are used for the message level: Start Message Transition (T_{sm}) and the End Message Transition (T_{em}). These transitions are encoded

as a conjunction of constraints over the current state variables **V**, while **V'** specifies the next state variables, cf. (5.6a):

$$\mathbf{T}(\mathbf{V}, \mathbf{V'}) = T_{st} \ \vee \ T_{rt} \ \vee \ T_{et} \ \vee \ T_{ct} \ \vee \ T_{sm} \ \vee \ T_{em} \tag{5.6a}$$

The reason we are using a set of transition relations instead of a single one, is due to the complexity of the integrated task and message scheduling problem and its consequently large and complex number of constraints for a single transition. Although the transitions are in arbitrary order, they cannot be executed non-deterministically since an arbitrary execution would lead to an inaccurate schedule, e.g. finishing a task that has not been started. Therefore, rules for setting the execution ordering of transitions are introduced in section 5.5. The set of transitions **T** is introduced in the following.

5.3.4.1 Start Task Transition

The Start Task Transition T_{st} allows for starting all tasks, whose predecessors are already set. Predecessors of a task t_j are defined as all messages m_k, that fulfill $\mathcal{D}(t_i,m_k) = t_j$.

$$\begin{aligned} T_{st}(\mathbf{V}, \mathbf{V'}) = \bigvee_{t_i \in T} \Bigg(& \overline{t_{i,started}} \quad \wedge \quad Node_{getNode(t_i),free} \quad \wedge \bigwedge_{j \in getPrecMsg(t_i)} m_{j,set} && \text{(5.7a)} \\ & \wedge \quad t'_{i,started} \quad \wedge \quad t'_{i,start} = Time && \text{(5.7b)} \\ & \wedge \quad Node'_{getNode(t_i),task} = t_i \quad \wedge \quad (t'_{i,clock} = 0) && \text{(5.7c)} \\ & \wedge \quad \overline{Node'_{getNode(t_i),free}} \Bigg) && \text{(5.7d)} \end{aligned}$$

The Start Task Transition T_{st} has several preconditions: (5.7a) demands that task t_i has not been started yet and the CPU of the tasks should be available. Furthermore, all input messages of task t_i need to be set (5.7a).

If there is a state s that satisfies these preconditions, T_{st} can be executed. Thus, task t_i is started ($t'_{i,started}$ = *true*) and the starting time $t'_{i,start}$ is set to the current global $Time$ variable (5.7b). $t'_{i,clock}$ stores the actual counter of computation duration and is set to 0. Furthermore, the tasks' node CPU is allocated with the current task t_i (5.7c) and therefore the resource is not free anymore (5.7d).

5.3.4.2 Run Task Transition

The Run Task Transition T_{rt} ensures that either the task has not started, it has finished or the clock of the task is incremented by one (5.8a).

$$T_{rt}(\mathbf{V}, \mathbf{V'}) = \bigvee_{t_i \in T} \Big(t_{i,started} \quad \wedge \quad \overline{t_{i,finished}} \quad \wedge \quad (t'_{i,clock} = t_{i,clock} + 1) \Big) \tag{5.8a}$$

Thus, the Run Task Transition T_{rt} ensures that a started task can be executed on its allocated node. However, the execution of the run task transition is bounded by several conditions. This is caused by the fact that at a given point in time several transitions need to be handled prior, caused by the characteristic of the given precedence graph $\mathcal{G}$ (compare section 5.5). This prioritization is necessary, because the run task transitions will increase the global time variable $Time$ and the variable $t'_{i,clock}$ by one if and only if the following conditions are fulfilled:

There exists a task t_i with:

- $t_{i,started} \quad \wedge \quad (t_{i,clock} < t_{i,comp})$
- No other task t_j is able to start (condition of T_{st} is not fulfilled)
- No other task t_k is able to be finished (condition of T_{et} is not fulfilled)
- No message m_l is able to start (condition of T_{sm} is not fulfilled)
- The change task transition T_{ct} can not be executed

The Run Task Transition T_{rt} allows tasks to run on their allocated node (nodes' CPU) by incrementing the time variable $Time$, respectively the task clock variable $t'_{i,clock}$ by one ($t'_{i,clock}$ = $t_{i,clock}$ + 1). Basically, the run task transition controls the progress of time. Each task, which has already started and whose computation duration is not actually reached, can proceed its execution time, controlled by the $t_{i,clock}$ variable.

These pre-conditions describe the exact situation in which the Run Task Transition T_{rt} is used, that is, the situation in which tasks execute and time is allowed to pass.

5.3.4.3 Change Task Transition

The Change Task Transition T_{ct} models the stopping of an already started task t_j by another task t_i allocated to the same resource $Node_i$. This is necessary, if the precedence graph $\mathcal{G}$ consists of concurrent task precedence relations which might originate two competitively tasks (share the same resource) at the same point in time. In order to find an optimal solution in terms of total length of the final schedule, it might be necessary to abort such an already started task t_j and to enable to start the concurrent task t_i instead. However, the Change Task Transition does

not support preemption, meaning that the stopped task t_j needs to start again (cf. scenario 2 in section 5.2).

$$T_{ct}(\mathbf{V}, \mathbf{V'}) = \bigvee_{t_i \in T} \Bigg(\overline{t_{i,started}} \quad \wedge \quad \overline{Node_{getNode(t_i),free}} \quad \wedge \quad \Bigg(\bigwedge_{j \in getPrecMsg(t_i)} m_{j,set} \Bigg) \tag{5.9a}$$

$$\wedge \quad t'_{i,started} \quad \wedge \quad t'_{i,start} = Time \quad \wedge \quad t'_{i,clock} = 0 \tag{5.9b}$$

$$\wedge \quad \overline{t'_{j,started}} \quad \wedge \quad t'_{j,start} = 0 \quad \wedge \quad t'_{j,clock} = 0 \tag{5.9c}$$

$$\wedge \quad Node'_{getNode(t_i),task} = t_i \quad \Bigg) \tag{5.9d}$$

The condition of T_{ct} are nearly the same as the condition for the Start Task Transition T_{st}, except of the variable $Node_{getNode(t_i),free}$, which denotes that the resource is already used, (5.9a), and thus set to *false*. For instance, task t_i is willing to start at a current point in time. But the CPU of the node is already allocated by the concurrent task t_j. The task t_j is obtained as $t_j = Node_{getNode(t_i),task}$, which stores the actual task currently using the resource.

If there is a $state$ satisfying these preconditions, the waiting task t_i is able to stop task t_j. Thus, task t_i is started, the starting time $t'_{i,started}$ is set to the current $Time$ variable (5.9b) and the computation time counter $t'_{i,clock}$ is set to 0. The variables for task t_j, which previously used the nodes' CPU, are reset (5.9c) and the resource is handed over to task t_i (5.9d).

5.3.4.4 End Task Transition

The End Task Transition T_{et} ensures that a task t_i can be finished and the resource can be released for another task execution.

$$T_{et}(\mathbf{V}, \mathbf{V'}) = \bigvee_{t_i \in T} \Bigg(t_{i,started} \quad \wedge \quad \overline{t_{i,finished}} \quad \wedge \quad t_{i,clock} = t_{i,comp} \tag{5.10a}$$

$$\wedge \quad t'_{i,finished} \quad \wedge \quad Node'_{getNode(t_i),free} \tag{5.10b}$$

$$\wedge \quad Node'_{getNode(t_i),task} = 0 \quad \Bigg) \tag{5.10c}$$

The preconditions in (5.10a) demand that a task t_i has already started and has not yet finished. Furthermore, the execution time, measured by the variable $t_{i,clock}$, should be equal to the computation duration stored in $t_{i,comp}$, which implies that the end of the tasks execution has been reached. If these conditions are satisfied, the variable $t'_{i,finished}$ can be set to true. Furthermore, the node's resource can be released for the next task (5.10b), (5.10c). In that case the task

execution is finished and the node's resource is released for the possible execution of the next task.

5.3.4.5 Start Message Transition

The Start Message Transition T_{sm} is one of two transitions that schedules the message on the bus level. This transition allocates the messages to time slots of the underlying time-triggered protocol.

$$T_{sm}(\mathbf{V}, \mathbf{V'}) = \bigvee_{m_i \in M} \Bigg(\overline{m_{i,started}} \quad \wedge \quad Bus_{free} \tag{5.11a}$$

$$\wedge \quad \Bigg(\bigwedge_{j \in getPrecTask(m_i)} t_{j,finished} \Bigg) \tag{5.11b}$$

$$\wedge \quad m'_{i,started} \quad \wedge \quad m'_{i,slot} = Time \quad \wedge \quad \overline{Bus'_{free}} \quad \Bigg) \tag{5.11c}$$

(5.11a) demands that the message m_i has not started and the next time slot is not allocated yet by another message. The predecessor task, which sends this message, is calculated by the function $getPrecTask(m_i)$. In this context the predecessor task t_i to message m_i is that task, that fulfills $\mathcal{D}(t_i,m_i) = t_j$. If this calculated task t_i is finished ($t_{i,finished} = true$), the transition is used (5.11b). The message variable $m'_{i,started}$ is set to true in the next state s', the slot allocation ($\sigma(m_i)$) uses the current time reference (5.11c), the bus is allocated Bus_{free} = *false* and the time variable $Time$ is incremented by one. (5.11c).

Furthermore, under certain conditions, e.g if no other task has been started or finished, T_{sm} (like T_{rt}) is able to progress time by one time unit (for a more detailed description compare section 5.5 and 5.6).

5.3.4.6 End Message Transition

The End Message Transition T_{em} ensures that a message m_i can be finished after its allocation to a time slot sl_i.

$$T_{em}(\mathbf{V}, \mathbf{V'}) = \bigvee_{m_i \in M} \Bigg(\overline{m_{i,set}} \quad \wedge \quad m_{i,started} \tag{5.12a}$$

$$\wedge \quad Time = m_{i,slot} + 1 \tag{5.12b}$$

$$\wedge \quad m'_{i,set} \quad \wedge \quad Bus'_{free} \Bigg) \tag{5.12c}$$

(5.12a) demands that the message m_i is not set yet, but already allocated to a certain slot, namely the one already started. Furthermore, the communication slot $m_{i,slot}$, respectively the global time has been incremented by one, that is the next time slot (5.12a).

If there is a state s that satisfies these conditions, the message m_i is set, $m'_{i,set}$ = *true*, and the communication resource is released (Bus'_{free} = true) (5.12c).

5.3.4.7 Wait Message Transition

The Wait Message Transition T_{wm} is rather related to the usage of SAL's bounded model checker, as described in scenario 4 in section 5.2. In order to calculate counterexamples with the given length k, we define that a message m_i might not be sent in the actual available slot sl_k, but wait for the next slot sl_{k+1}. This allows for schedules with variable length l.

$$T_{wm}(\mathbf{V}, \mathbf{V'}) = \bigvee_{m_i \in M} \Bigg(\overline{m_{i,started}} \quad \wedge \quad Bus_{free} \tag{5.13a}$$

$$\wedge \left(\bigwedge_{j \in getPrecTask(m_i)} t_{j,finished} \right) \tag{5.13b}$$

$$\wedge \quad \overline{m'_{i,started}} \quad \wedge \quad Bus_{free} \Bigg) \tag{5.13c}$$

The conditions of the Wait Message Transition T_{wm} are the same as for the Start Message Transitions T_{sm}. (5.13b) demands that the message m_i has not started and the next time slot is not yet allocated by another message. The predecessor task, which sends this message is calculated by the function $getPrecTask(m_i)$. The predecessor task t_i to message m_i is the task, that fulfills $\mathcal{D}(t_i,m_i) = t_j$. If this calculated task t_i is finished ($t_{i,finished} = true$), the transition can be fired (5.13c). The message variable $\overline{m'_{i,started}}$ is kept false in the next state s', because the message is waiting for the next available slot. The communication resource remains free as well (Bus_{free} = *true*).

In certain conditions, e.g if no other task may be started of finished, the start message transition T_{sm}, like the run task transition T_{rt}, is able to progress time by one time unit. If the given conditions are fulfilled the time variable $Time$ is incremented by one.

5.4 SAL Representation

We use SAL (Symbolic Analysis Laboratory, `http://sal.csl.sri.com`) [dMOR+04] from SRI International as a framework for specification and scheduling synthesis. Thus, we

take the formal model, as specified in section 5.3, as a basis for translating scheduling synthesis problem to SAL specifications, as published in [VSE09].

5.4.1 Representation of a state

In order to implement a state representation in SAL, we use the different elements of the precedence graph, specified in section 5.3.1, namely a set of tasks, a set of messages and a set of nodes.

As an example, we consider a precedence graph $\mathcal{G}$ consisting of four tasks deployed on three different nodes and connected with 4 edges. We begin by defining all given tasks via the enumeration type `TASKS`. Furthermore, the enumeration types `MESSAGES` and `NODES` specify, respectively, all messages and all nodes of the given precedence graph.

```
TASKS : TYPE = {Task0, Task1, Task2, Task3};
MESSAGES : TYPE = {Edge0, Edge1, Edge2, Edge3};
NODES : TYPE = {Node0, Node1, Node2};
```

We assume that time is discrete. Thus, we define the `Time` - variable in this manner. Possible clock values are nonnegative integers and events can only occur at integer time values.

```
TIME : TYPE = [0..100];
```

As a next step, the type declaration for these precedence graph elements are specified in SAL. Therefore, several parameters are defined in dedicated data structures. In SAL, tasks, messages and nodes are defined as a task, a message, and a node record, respectively (cf. figure 5.5).

Task Record	Message Record	Node Record
`taskrecord:TYPE=`	`msgrecord:TYPE=`	`noderecord:TYPE=`
`[#`	`[#`	`[#`
`t_started:BOOLEAN,`	`m_started:BOOLEAN,`	`node_free:BOOLEAN,`
`t_finished:BOOLEAN,`	`m_set:BOOLEAN,`	`node_task:TASKS`
`t_start:NATURAL,`	`m_slot:NATURAL`	`#];`
`t_clock:NATURAL,`	`#];`	
`t_comp:NATURAL,`		
`t_node:NODES`		
`#];`		

Figure 5.5: Declaration of task, message and node records

The defined task record `taskrecord`, for instance, stores the parameter information of a single task t_i. The variable `t_started` indicates whether t_i has already started or not. Because a tasks' computation time can last for more than one time step, the variable `t_finished` indicates whether task t_i has finished or not. In combination with the variable `t_clock`, which holds the actual task

computation duration, the current status of a task can be exactly described. To obtain a time schedule, each task needs to store its calculated starting time. This is done by the variable `t_start`. Furthermore, `t_comp` stores the given computation duration of task t_i, and `t_node` represents the node a task is allocated to. We specify and store the parameters of all tasks defined for the given precedence graph $\mathcal{G}$ as an array of records:

```
TASKARRAY : TYPE = ARRAY TASKS OF taskrecord;
MSGARRAY : TYPE = ARRAY MESSAGES OF msgrecord;
NODEARRAY : TYPE = ARRAY NODE OF noderecord;
```

Precedence Relations in SAL

The current state definition does not include any precedence relations. By modeling precedence relations for a given precedence graph in SAL, a distinction between task and message precedence relation needs to be done. For each given task t_i, and message m_i respectively, the direct predecessor is stored to cope with all kinds of precedence graph characteristics. A predecessor of a task is defined as a message, except when the task is the starting task (source of the precedence graph). In that case the source task has no predecessors. For instance, a task t_i may receive two messages. These messages are stored as predecessors of that task. A predecessor of a message is always a single task. These precedence relations are represented via an array structure, as

```
PREC_TASK : ARRAY INDEX OF MESSAGES
PREC_MSG : ARRAY INDEX OF TASKS;
```

The allocation of tasks to nodes, defined as $\tau : T \rightarrow 2^M$ (cf. section 2.3), is specified by an array structure as:

```
NODETASKS: TYPE = ARRAY INDEX OF NODES
```

The unique precedence relation of each single task is modeled as follows:

```
prec_TASK1 : PREC_TASK = [[i:INDEX]Edge0];
prec_TASK2 : PREC_TASK = [[i:INDEX]Edge1];
prec_TASK3 : PREC_TASK = [[i:INDEX]];
```

The precedence relations for all messages are modeled in the same way:

```
prec_Edge1: PREC_MSG = [[i:INDEX]Task0];
prec_Edge2: PREC_MSG = [[i:INDEX]Task1];
...
```

Precedence functions

Two functions `getPrecTask` and `getPrecMsg` are implemented and used as a condition request about a task's or a message's predecessors. The `getPrecTask` function is used by a message to check whether its predecessor (namely the task who sends this message) is already finished

(t_finished = TRUE). Hence, getPrecTask has two parameters: the message m, whose predecessors' condition is requested and the task record array taskarry. If the task, which is predecessor (or sender) of message m is found, the current status (condition) can be detected. If and only if the predecessor task is already finished the function returns TRUE. This scenario is specified in SAL as follows:

```
getPrecTask(m : MESSAGES, taskarray : TASKARRAY) : BOOLEAN =
IF(m=Edge0 AND(EXISTS(j:INDEX1):taskarray[prec_Edge0[j]]
       .t_finished=FALSE)) THEN FALSE
ELSIF (m=Edge1 AND(EXISTS(j:INDEX1):taskarray[prec_Edge1[j]]
       .t_finished=FALSE)) THEN FALSE
ELSIF ...
ELSE TRUE
ENDIF;
```

The function getPrecMsg(t:TASKS, msgarray:MSGARRAY) : BOOLEAN detects for a task t_i if all predecessors, namely all messages m_j with $\mathcal{D}(t_x, m_j) = t_i$ are already set. If all predecessors messages are already scheduled, the function returns TRUE. Initially, only the source task fulfills this request. Thus, it can be said, that the schedule is build according to the precedence graph from source to sink. The function getNodeTask(t:TASKS):NODES is modeled to detect and return the node n, a task t_i is allocated to. Accordingly, the function getPrecTask(m:MESSAGES, taskarray:TASKARRAY):BOOLEAN detects for a message m_i if its predecessor, namely the task t_i with $\mathcal{D}(t_i, m_i) = t_j$ is already set.

5.4.2 The basic module

The basic construct in SAL is a module. A module contains the definition of variables (including, local, global, input and output), the initial state, and the transition relations. We specify this discrete model in the language of SAL as follows. The discrete-time variable Time is used for specifying the global system time.

```
scheduler : MODULE =
BEGIN
  GLOBAL currenttaskarray : TASKARRAY
  GLOBAL currentmessagearray : MSGARRAY
  GLOBAL currentnodearray : NODEARRAY
  GLOBAL Time : TIME
  GLOBAL Bus_free:BOOLEAN
```

5.4.3 Initialization

The initial state is specified in the language of SAL as follows.

```
INITIATLIZATION
  currenttaskarray[Task0].t_comp = 2.0;
  currenttaskarray[Task1].t_comp = 2.0;
  ...
  currenttaskarray[Task0].t_node = Node0;
  currenttaskarray[Task1].t_node = Node1;
  ...
  Time = 0;
  Bus_free = TRUE;
  ...
  (FORALL (i:TASKS): currenttaskarray[i].t_started = FALSE);
  (FORALL (i:TASKS): currenttaskarray[i].t_finished = FALSE);
  (FORALL (i:TASKS): currenttaskarray[i].t_start = 0);
  (FORALL (i:TASKS): currenttaskarray[i].t_clock = 0);
  (FORALL (i:MESSAGES): currentmessagearray[i].m_started = FALSE);
  (FORALL (i:MESSAGES): currentmessagearray[i].m_set = FALSE);
  (FORALL (i:MESSAGES): currentmessagearray[i].m_slot = 0);
  (FORALL (i:NODES): currentnodearray[i].node_free = TRUE);
  (FORALL (i:NODES): currentnodearray[i].node_task = Task3);
```

Initially, the given computation time for each task `cureenttaskarray[Task0].t_comp` is set. This is done for each task individually. In this case all computation times corresponds to the length of two time units. Furthermore, the allocation of tasks to nodes is initialized, e.g. `currenttaskarray[Task0].t_node=Node0`. The global variable `Time` is set to 0. The time-triggered communication bus is represented by the variable `Bus_free`, which indicates whether, at a certain point in time, the bus, respectively the current time slot, is allocated by a node. Initially, the bus is not used (`Bus_free=TRUE`). Initialization of the record arrays for tasks, messages and nodes can be described as follows: None of the tasks has started, thus the variable `t_started` is set to `FALSE`. No task has finished (`t_finished = FALSE`) and the starting time variable (`t_start`) is set to 0. The clock variable `t_clock` is also set to 0. The initialization for all given messages is done in the same way. All messages are not started (`m_started=FALSE`). The variable `m_set=FALSE` points out that no messages has been scheduled yet, which implies that also no slot has been allocated by a certain message (`m_slot=0`). For the node record the parameter `node_free` is initially set to the sink task in the given precedence graph $\mathcal{G}$.

5.4.4 Transitions

State transitions are specified in SAL via guarded commands. Here, the [character introduces a set of guarded commands, which are separated by the [ ] symbol. A SAL guarded command is eligible for execution in the current state if its guard (i.e., the part before the `-->` arrow) is true. The SAL model checker nondeterministically selects one of the enabled commands for execution at each step. In case no command is eligible, the system is deadlocked. State variables are unprimed before execution of a command and primed in the new state, that is after the execution of the command.

The scheduling algorithm is encoded in SAL using six different transition relations. Four transition relations (`start_task`, `run_task`, `change_task`, `end_task`) are used to model task scheduling, while `start_message` and `end_message` encodes message scheduling on bus level.

Start Task Transition

In order to illustrate the Start Task Transition T_{st}, the implementation of this transition in SAL is depicted below. The following transition is executed for all tasks t_i, for which the precondition holds Thus, if any task t_i, represented by the SAL code fragment `currenttaskarray WITH [i]`, fulfilling the three conditions defined in (5.7a), is allowed to be started (`[i].t_started := TRUE`). The clock is set to 0 and the starting time can be calculated by the actual time (`[i].t_start := Time`). Furthermore, the node's resource is allocated by the current task t_i, therefore, the `.node_free` variable is set to `FALSE` as specified below.

```
[([](i:TASKS): start_transition:
currentnodearray[getNodeTask(i)].node_free = TRUE AND
currenttaskarray[i].t_started = FALSE AND
getPrecMsg(i,currentmessagearray) = TRUE
-->
currenttaskarray'= currenttaskarray WITH [i].t_started:=TRUE
                   WITH [i].t_clock:=0
                   WITH [i].t_start:=Time;
currentnodearray'= currentnodearray
                   WITH [getNodeTask(i)].node_free:=FALSE
                   WITH [getNodeTask(i)].node_task:=i)
]
```

Run Task Transition

The Run Task Transition T_{rt} allows tasks to run on their allocated node (using the nodes' CPU resource) by incrementing the time variable `Time`, respectively the task's clock variable `clock` by one. Basically, the run transition controls the progress of time. Each task, which has already started and whose computation duration is not actually reached, can progress its execution time, controlled by the `t_clock` variable. The first part of the pre-condition (i.e., first disjunct) states that the current task t_i has already started. The value of the task's clock variable $t_{i,clock}$ has to be less than the value of the computation time variable $t_{i,comp}$, otherwise the task would be a candidate for the end transition. Additional conditions, for excluding the possibility of executing other transitions, have to be satisfied for enabling the Run Task Transition. These conditions are described as follows.

There exists a task t_i with:

- $t_{i,started} \quad \wedge \quad (t_{i,clock} < t_{i,comp})$
- No other task t_j is able to start (condition of T_{st} is not fulfilled)

- No other task t_k is able to be finished (condition of T_{et} is not fulfilled)
- No message m_l is able to start (condition of T_{sm} is not fulfilled)
- The change task transition T_{ct} can not be executed

```
[]
run_transition:
  (EXISTS (i:TASKS): currenttaskarray[i].t_started = TRUE AND
        currenttaskarray[i].t_clock < currenttaskarray[i].t_comp AND
  (NOT (EXISTS (j:TASKS): j /= i AND
       (currentnodearray[getNodeTask(j)].node_free=TRUE AND
        currenttaskarray[j].t_started = FALSE AND
        getPrecMsg(j,currentmessagearray) = TRUE))) AND
  (NOT (EXISTS (k:TASKS): k /= i AND
       (currenttaskarray[k].t_comp=currenttaskarray[k].t_clock AND
        currenttaskarray[k].t_finished=FALSE))) AND
  (NOT (EXISTS (l:MESSAGES): (currentmessagearray[l].m_set = FALSE AND
        currentmessagearray[l].m_started = FALSE AND
        Bus_free = TRUE AND
        getPrecTask(l,currenttaskarray)=TRUE )))
  OR
  (EXISTS (m:TASKS): m /= i AND
        currentnodearray[getNodeTask(m)].node_free = FALSE AND
        getNodeTask(m) = getNodeTask(i) AND
        currenttaskarray[m].t_started = FALSE AND
        getPrecMsg(m,currentmessagearray)=TRUE) AND
        currenttaskarray[i].t_started = TRUE AND
        currenttaskarray[i].t_clock < currenttaskarray[i].t_comp AND
  (NOT (EXISTS (j:TASKS): j /= i AND
       (currentnodearray[getNodeTask(j)].node_free=TRUE AND
        currenttaskarray[j].t_started = FALSE AND
        getPrecMsg(j,currentmessagearray)=TRUE) )) AND
  (NOT (EXISTS (k:TASKS): k /= i AND
       (currenttaskarray[k].t_comp=currenttaskarray[k].t_clock AND
        currenttaskarray[k].t_finished=FALSE))) AND
  (NOT (EXISTS (l:MESSAGES): (currentmessagearray[l].m_set = FALSE AND
        currentmessagearray[l].m_started = FALSE AND
        Bus_free = TRUE AND
        getPrecTask(l,currenttaskarray)=TRUE )))
-->
currenttaskarray'[Task0].t_clock =
  IF currenttaskarray[Task0].t_started = TRUE AND
     currenttaskarray[Task0].t_clock < currenttaskarray[Task0].t_comp
    THEN currenttaskarray[Task0].t_clock+1
    ELSE currenttaskarray[Task0].t_clock ENDIF;
currenttaskarray'[Task1].t_clock =
  IF currenttaskarray[Task1].t_started = TRUE AND
  ...
Time'=Time+1;)
```

The first part of the pre-condition (i.e., first disjunct) states that the current task t_i has already started. The value of task clock variable t_{clock} has to be less than the value of the computation time variable t_{comp}, otherwise the task would be a candidate for the end transition. Three additional conditions, for excluding the possibility of executing other transitions, have to be satisfied for enabling the run transition:

- No other task is able to start on another node (handled by the `start_transition`),
- No other task is able to be finished (handled by the `end_transition`) and
- No message is able to start (handled by the `start_message_transition`).

The second part of the disjunction in the precondition specifies the reaction, in case of interruption of the current running task by another one (this scenario is explained in section cf. 5.3.4.3). These conditions state that:

- Another task can be started on the same node,
- No other task can be finished and
- No message is able to start.

These pre-conditions describe the exact situation in which the Run Task Transition T_{rt} is used, that is, the situation in which tasks executes and time is allowed to pass.

End Task Transition

The End Task Transition ensures that a task t_i can be finished and the resource released. The guarded command detects whether the task execution time (represented by `currenttaskarray[i].t_clock`) equals the task's computation time (`currenttaskarray[i].t_comp`).

```
([](i:TASKS): endtransition:
 currenttaskarray[i].t_started = TRUE AND
 currenttaskarray[i].t_clock = currenttaskarray[i].t_comp AND
 currenttaskarray[i].t_finished = FALSE
 -->
 currenttaskarray'[i].t_finished = TRUE;
 currentnodearray'= (currentnodearray WITH [getNodeTask(i)].node_free:=TRUE
                          WITH [getNodeTask(i)].node_task:=Task3)
)
```

In case these preconditions are fulfilled the task's execution is finished and the node's resource is released for the possible execution of the next task.

Start Message Transition

The Start Message Transition enables sending of a message on the bus in the next available time slot of the underlaying time-triggered protocol. As a precondition the next possible slot needs to be free (not already allocated by another message). This is checked by the variable `Bus_free=TRUE`. Furthermore, the message should not be started and not finished yet, (`currentmessagearry[i].m_started=FALSE, currentmessagearry[i].m_set = FALSE`). However, in certain conditions the `start_message_transition`, like the `run_task_transition`, is able to progress time by one time unit. This, however, prerequires a check of these conditions (e.g. if no other task may be started of finished) Thus, it needs to be checked, whether these other transitions are able to be handled first.

Therefore, two more conditions, introduced by `NOT (EXISTS...)`, are implemented in SAL to cover these constrains:

- No task is able to start (handled by the `start_task_transition`)
- No task is able to be finished (handled by the `end_task_transition`)

```
([](i:MESSAGES): start_message_transition:
 currentmessagearray[i].m_set = FALSE AND
 currentmessagearray[i].m_started = FALSE AND
 Bus_free = TRUE AND
 getPrecTask(i,currenttaskarray)=TRUE AND
 (NOT(EXISTS(j:TASKS):currentnodearray[getNodeTask(j)].node_free=TRUE AND
       currenttaskarray[j].t_started = FALSE AND
       getPrecMsg(j,currentmessagearray)=TRUE)) AND
 (NOT(EXISTS(k:TASKS):currenttaskarray[k].t_started = TRUE AND
   currenttaskarray[k].t_clock = currenttaskarray[k].t_comp AND
   currenttaskarray[k].t_finished = FALSE))
 -->
 currentmessagearray'=currentmessagearray WITH [i].m_started:=TRUE
                                          WITH [i].m_slot:=Time;
 Bus_free' = FALSE;
 Time'=IF(NOT (EXISTS(j:TASKS):currenttaskarray[j].t_started=TRUE AND
                               currenttaskarray[j].t_finished=FALSE))
          THEN Time+1
       ELSE Time
       ENDIF;
)
```

Thus, if these preconditions are satisfied, the current message, (`currenttaskarray[j]`), is started and allocated to a certain slot. Hence, the bus is blocked and, if there are no further tasks which need to be started or finished, then the time variable is incremented.

End Message Transition

The End Message Transition ensures that a message m_i can be finished and the bus resource is released. The guarded command represents this requirement. Therefore it is checked, whether time has elapsed since starting the message / slot allocation (cf. `Time = currentmessagearray[i].m_slot+1`). If these conditions are satisfied the bus resource is released and the message is marked as set (`currentmessagarray'[i].m_set = TRUE`).

```
([](i:MESSAGES): endmessagetransition:
currentmessagearray[i].m_set = FALSE AND
currentmessagearray[i].m_started = TRUE AND
Time = currentmessagearray[i].m_slot+1
-->
currentmessagearray'[i].m_set = TRUE;
Bus_free' = TRUE
)
```

For being able to generate a task and message schedule, which is optimal in the sense that to minimize the maximum end-to-end latency is minimized, it is important that the order in which the transitions are executed is controlled. This is done due to the optimization criteria as well as for complying with all kinds of precedence graph characteristics (e.g., sequential or concurrent precedence graphs). In the following we describe this transition ordering.

5.5 Transition Ordering

Although the transitions are written in arbitrary order, they cannot be executed non-deterministically due to undesirable effects on the integrated schedule, meaning that for instance, that tasks need to be started and computed before they can be finished. Thus, an ordering has to be chosen such that it reflects the constraints posed on task execution and message transmission as given by the precedence graph.

For the given example 1 in section 2.2, a possible execution ordering of transitions is illustrated below:

The given precedence graph $\mathcal{G}$ is traversed from source to sink. The source task(s) t_{source} are characterized by the non-existence of any predecessors (input messages). Thus, the start task transition T_{st} for such source tasks are executed first. Whenever the given precedence graph $\mathcal{G}$ allows for different possible solutions, characterized by the concurrency to a shared resource (either the time-triggered bus or a nodes' CPU), we use the Model Checker capabilities (state space exploration) to explore all interleaved possibilities. The ordering of transitions is caused by this requirement. Figure 5.6 points out that both Edge0 (message 0) and Edge1 (message1) may access the next available time slot. Thus, both ways are handled via different next states s'. Generally, from each given state s the transitions generate a next state s' with all necessary

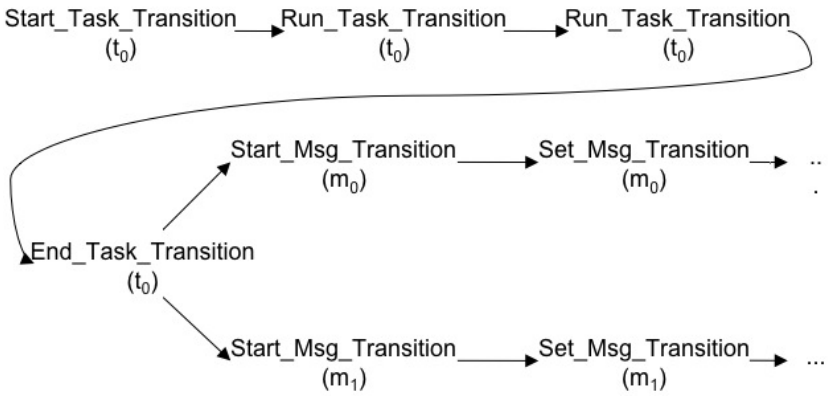


Figure 5.6: Example of possible transition execution order

scheduling combinations which are possible, according to the given precedence graph $\mathcal{G}$. Because the time variable $Time$ can also be increased in two transitions (compare section 5.6) the ordering of transitions needs to be specified (compare run task transition T_{rt} and start message transition T_{sm}).

5.6 The 'Time' variable

A discrete time variable `Time` is necessary in order to have a common understanding of the time in the system. The granularity of the time variable corresponds to a slot duration. We assume the computation time of a task ($c(t_i)$) to be equal to a slot duration or multiples of it. This includes the time consumed for reading and writing the buffer. The global time increases discretely by this time unit. The `Time` variable is then used to allocate starting times to tasks and slot numbers to messages.

Furthermore, we allow only the following three different transitions to increase the time variable by one time unit: the Run Task Transition (T_{rt}), Start Message Transition (T_{sm}) and the Wait Message Transition (T_{wm}). Before incrementing the `Time` variable in these transitions, certain conditions need to be fulfilled, that is other transitions might be executed prior. As the Wait Message Transition (T_{wm}) is rather related to the usage of SAL's bounded model checker it is not used in case of using the symbolic model checker.

Before incrementing the $Time$ variable in these three transitions, it needs to be guaranteed that all possible other actions, namely other transitions, have been executed at this point in time. This corresponds to the traversal of the precedence graph from source to sink. For instance: If there are any tasks that are able to start (their starting times can be allocated), these tasks need to start (Start Task Transition T_{st}) before the time is incremented. This would cause a non-optimal schedule in terms of minimal end-to-end latency.

Therefore, the three transitions (T_{rt}, T_{sm} and T_{wm}), implying an increase of time variable $Time$, are executed if and only if all of the following transitions are executed prior. This is done in arbitrary order.

The Run Task Transition T_{rt}

- Start Task Transition T_{st}
- Interrupt Task Transition T_{it}
- End Task Transition T_{et}
- End Message Transition T_{em}
- Start Message Transition T_{sm}

The Start Message Transition T_{sm}

- Start Task Transition T_{st}
- End Task Transition T_{et}

The Wait Message Transition

- Start Task Transition T_{st}
- End Task Transition T_{et}

5.7 Solving (Hard) Real-Time Scheduling Problems using SAL

Symbolic witnesses and counterexamples yield proofs for the validity or falsity of a CTL formula in a model. The primary function of a model checker is the analysis of the specified model $\mathcal{M}$ with respect to a given property φ. Let $\mathcal{M}= \langle \mathbf{V}, \mathbf{I}, \mathbf{G}, \mathbf{T} \rangle$ be the model for the task and message scheduling problem, as defined in section 2.4. Furthermore, let φ be a CTL formula specifying that there exists a state where all tasks have been finished (5.14a).

$$\varphi \; = \; EF \quad (\forall \, t_i \in T. \quad t_{i,finished}) \tag{5.14a}$$

The condition that all messages are sent is implicitly contained in this formula, since all task finished implies that all messages have been sent as well. This is guaranteed by the functions specified in 5.3.1.

For obtaining a schedule as a situation to the considered task and message scheduling problem, we model-check the negation of the above CTL formula, specified with `AG`, which denotes the *always globally* operator of linear temporal logic (5.15a):

$$\psi \; = \; \neg \, \varphi \; = \; AG \quad (\exists \, t_i \in T. \quad \overline{t_{i,finished}}) \tag{5.15a}$$

The model checker returns either *verified* or *falsified*, depending whether the given property ψ is fulfilled by the model or not. In the latter case, the model checker usually outputs a counterexample for the property ψ, which is a path or trace $\pi(s_0, ..., s_n)$ such that $I(s_0)$ and $\neg\varphi(s_n)$, that is, $\psi(s_n)$.

In case of finding such a state, it would suggest that there is a state s_n in which all tasks are scheduled, and implicitly the messages are. The counterexample or path $\pi(s_0, ..., s_n)$ shows how to reach this state and gives us the path to the final integrated task and message schedule. The last step s_n of the counterexample equals the complete task and message schedule, which implicitly fulfills the requested requirements. Thus, a schedule under consideration is found.

The SAL representation is depicted below:

```
th: theorem schedule |- AG ( EXISTS (i:TASKS):currenttaskarray[i]
                                                .t_finished=FALSE);
```

In the following we illustrate how the scheduling problem for a set of four tasks is solved by using symbol model checking and bounded model checking techniques. For analysis and demonstration of operation, we use the simple example explained in section 2.2.

5.7.1 Task and Message Scheduling using SAL-SMC

We invoke SAL's symbolic model checker `sal-smc` by the following line of code.

```
sal-smc -v 4 context.sal th
```

`sal-smc` search for a counterexample to the property `th`. SAL offers several options that can be used for model checking the given `context.sal` file. The `-v 1` option controls how much information is given by the model checker. The `--enable-dynamic-reordering` option enables to usage of dynamic reordering strategies for Ordered Binary Decision Diagrams (ODBBs). These strategies may accelerate the process of computation and increase its efficiency. This is investigated in section 8.3. The set of options can be displayed by the command `sal-smc --help`.

SAL outputs a counterexample, which contains several interesting properties that are listed below. The verification time of `sal-smc` used to find a counterexample for the given scheduling problem is measured with 0.953 seconds. The Model Checker explores 367 states with a total execution time of 6.985 seconds.

```
Verification Time: 0.953 secs
total execution time: 6.985 secs
number of system variables: 97
number of visited states: 367.0
BDD conversion time: 5.422 secs
```

The last state of the counterexample contains the final schedule, is depicted in section 5.7.3. This state and the extracted schedule is depicted in section 5.7.3.

5.7.2 Task and Message Scheduling using SAL-INF-BMC

We invoke SAL's infinite-state bounded model checker `sal-inf-bmc` by the following line of code:

```
sal-inf-bmc -d 1 -v 1 context.sal th
...
no counterexample between depth: [0, 1].
```

`sal-inf-bmc` search for a counterexample to the property `th`. The `-d 1` option specifies the depth of the search, in this case, only one step. This depth comprises to the bound k that decides weather the system satisfies the logic property ψ by exploring the underlying state space in a depth of k. The `-v 1` option controls how much information `sal-inf-bmc` outputs. `context.sal` is the name of the SAL context in which the property `th` is specified (compare section 5.3).

As expected, `sal-inf-bmc` does not find any counterexample at depth one. We increase the search depth.

```
sal-inf-bmc -d 24 -v 1 scheduler th
```

This invocation finds a counterexample at exactly 24 steps. A BMC Time of 16.284 seconds and a total execution time of 17.004 seconds are obtained. The last step of the given counterexample equals the result from `sal-smc` and is depicted in section 5.7.3. Thus, the same schedule is calculated.

```
BMC Time: 16.284 secs
total execution time: 17.004 secs
number of system variables: 42
```

However, it is possible to run the bounded model checker with iterative deeping. With this option, `sal-inf-bmc` searches for counterexamples of increasing length. It stops when either a counterexample is detected or the search range is covered: `sal-inf-bmc -d 26 -it scheduler`

th. SAL outputs a counterexample with the following parameters: The verification time of sal-inf-bmc while finding a counterexample is measured with 152.387 seconds. The Model Checker needs a total execution time of 153.387 seconds.

5.7.3 Counterexample and Schedule

The model checker shows that there is a state s_n, that fails the correctness property ψ. This state corresponds to the situation in which all tasks are finished. Because of the given functions getPrecTask and getPrecMsg (compare section 5.3) and the sequence of transitions, specified in section 5.5, it follows that all messages are allocated to slots as well.

By finding such a state s_n that fails the correctness property, the model checker outputs a counterexample. A counterexample is depicted in the form of computation traces that are generated when a property fails to hold. The complete integrated task and message schedule is included in the last state of this counterexample. For the given example 1 (cf. section 2.2), the model checker outputs a counterexample that contains 24 transitions steps. The last state s_n is depicted below:

```
Step 24: --- System Variables (assignments) ---
currenttaskarray[Task0].t_clock = 2
currenttaskarray[Task0].t_comp = 2
currenttaskarray[Task0].t_finished = true
currenttaskarray[Task0].t_node = Node0
currenttaskarray[Task0].t_start = 0
currenttaskarray[Task0].t_started = true
currenttaskarray[Task1].t_clock = 2
currenttaskarray[Task1].t_finished = true
currenttaskarray[Task1].t_node = Node1
currenttaskarray[Task1].t_start = 3
currenttaskarray[Task1].t_started = true
currenttaskarray[Task2].t_clock = 2
currenttaskarray[Task2].t_comp = 2
currenttaskarray[Task2].t_finished = true
currenttaskarray[Task2].t_node = Node1
currenttaskarray[Task2].t_start = 5
currenttaskarray[Task2].t_started = true
currenttaskarray[Task3].t_clock = 2
currenttaskarray[Task3].t_comp = 2
currenttaskarray[Task3].t_finished = true
currenttaskarray[Task3].t_node = Node2
currenttaskarray[Task3].t_start = 8
currenttaskarray[Task3].t_started = true
currentmessagearray[Edge0].m_set = true
currentmessagearray[Edge0].m_slot = 2
currentmessagearray[Edge0].m_started = true
currentmessagearray[Edge1].m_set = true
currentmessagearray[Edge1].m_slot = 3
currentmessagearray[Edge1].m_started = true
```

```
currentmessagearray[Edge2].m_set = true
currentmessagearray[Edge2].m_slot = 5
currentmessagearray[Edge2].m_started = true
currentmessagearray[Edge3].m_set = true
currentmessagearray[Edge3].m_slot = 7
currentmessagearray[Edge3].m_started = true
Time = 10
Bus_free = true
```

From the values of the state variables of the counterexample, a schedule $\gamma = \{t_i \mapsto \gamma_i | \forall t_i \in T\}$ is extracted. For the simple example considered here, the following schedule can be extracted:

$$\begin{aligned} \gamma \;=\; & \{t_0 \mapsto \langle 0, \{sl_2, sl_3\}\rangle, \\ & t_1 \mapsto \langle 3, \{sl_5\}\rangle, \\ & t_2 \mapsto \langle 5, \{sl_7\}\rangle, \\ & t_3 \mapsto \langle 8, \{\}\rangle\} \end{aligned}$$

The calculated schedule is illustrated in figure 5.7. Tasks t_1 and t_2 are allocated to the same node: $Node_1$. Hence, the execution ordering needs to be sequential for that node.

The global variable `Time` denotes the length of the final schedule. This variable is only influenced by the three transitions: `run_task_transition`, `start_message_transition` and `wait_message_transition`. As explained in section 5.5, these transitions guarantee that the time only proceeds, if no other transitions are able to be handled. That is why the current time reached in any certain state, represented by the variable `Time`, equals the schedule length. For the given example the schedule has a length of 10 time units.

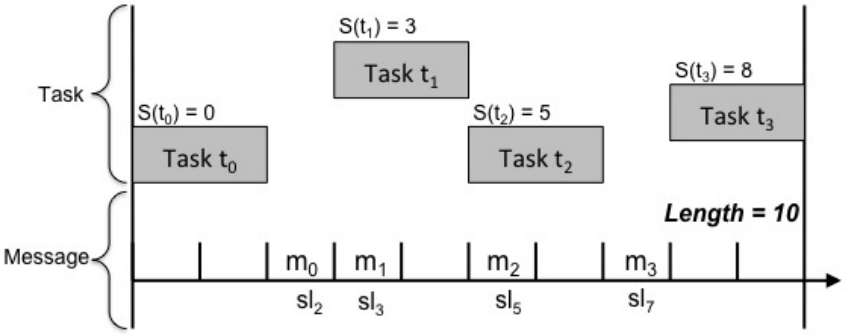


Figure 5.7: Calculated Task and Message Schedule for the given Example 1

In general, the obtained schedule might not be optimal, with respect to end-to-end latency, because the transition system is allowed to take transitions at which no time is consumed. To obtain an optimal task and message schedule, we further use a binary search for finding that solution, which is presented in the next section.

5.8 Binary Search Algorithm for finding the optimal solution

The model checker analyzes the specified model $\mathcal{M}$ with respect to a given property ψ. The model checker returns either *verified* or *falsified*, depending whether or not a given property is fulfilled by the model.

As illustrated in section 5.7, the solution to the task and message scheduling problem is formally characterized by the CTL formula $\varphi = EF \quad (\forall\ t_i \in T. \quad t_{i,finished})$. Such a solution is found by model checking the negated formula $\psi = \neg\ \varphi = AG \quad (\exists\ t_i \in T. \quad \overline{t_{i,finished}})$. If ψ is falsified, the model checker outputs a counterexample of length k. This would implicitly indicate that there are no counterexamples with k steps. However, by failing the property ψ we obtain a counterexample containing a task and message schedule which might not be optimal from a schedule length perspective. In this case we use a binary-search based model checking strategy for finding the optimal solution.

The reason for finding schedules with shorter end-to-end latency is that the transition system is allowed to take transitions at which no time is consumed. The transition `change_task_transition`, for instance, is used to interrupt an already started task by another task requesting for the same computing resource (cf. scenario 3 in section 5.2). This transition does not consume any time. However, depending on the precedence graph, this transition is necessary to compute the optimal schedule under consideration.

Therefore, a binary search strategy is introduced that is able to find the optimal schedule with respect to the minimal end-to-end latency.

We consider V as a possible solution of $AG(\varphi)$, namely a counterexample for the finite-state model checking problem $\mathcal{M} \models \varphi$. The solution V can be characterized by a task and message schedule with a dedicated $Time$ variable, indicating the length of the given schedule. In a next step a counterexample for $AG(\frac{V}{2})$ is investigated. If there exists a counterexample for $AG(\frac{V}{2})$, the *true* option is performed as a next search, otherwise the *false* option.

The binary search algorithm is specified in equation 5.16.

$$
\begin{aligned}
\psi_0 \quad &= AG\ (\exists\ t_i \in T. \quad \overline{t_{i,finished}}) \rightarrow \text{ Time = t} \quad \text{(MC Aufruf liefert Time = t)} \\
\psi_i \quad &= \begin{cases} AG\ (\exists\ t_i \in T. \quad \overline{t_{i,finished}} \text{ OR Time} \geq \frac{t}{2^i}) & \text{,if } \psi_{i-1} = \textit{true} \\ AG\ (\exists\ t_i \in T. \quad \overline{t_{i,finished}} \text{ OR Time} \geq \text{t-}\frac{t}{2^i}) & \text{,if } \psi_{i-1} = \textit{false} \end{cases}
\end{aligned} \tag{5.16}
$$

The SAL representation below depicts the usage of the presented binary search to find the optimal solution of the task and message scheduling problem. The recursive formula starts with a first calculation, ψ_0, cf. (5.17). In this exemplary case, the solution V calculates 12 time units as the length for a potential schedule. Thus, the next calculation ,ψ_1, looks like (5.18) using $\frac{t}{2^1}$ according to the specified binary search and so on (5.19).

`theorem` ⊨`AG(currenttaskarray[t5].t_finished=FALSE);` (5.17)

`theorem` ⊨`AG(currenttaskarray[t5].t_finished=FALSE OR Time` $\geq 12/2$); (5.18)

`theorem` ⊨`AG(currenttaskarray[t5].t_finished=FALSE OR Time` $\geq 12 - ((12/2) - 2)$); (5.19)

Experimental results are presented in section 8.3. These experiments illustrate that SAL' model checker `sal-smc` can be used to automatically compute optimal schedules using the presented binary search.

Chapter 6

Weighted Symbolic Scheduling

Generally, *formal verification techniques* are a very attractive and appealing alternative to simulation and testing [EMCGP99]. While simulation and testing explore some of the possible behaviors and scenarios of the system, formal verification conducts an *exhaustive exploration* of all possible behaviors.

This thesis shows that formal verification techniques, as for example *model checking*, can be used for solving scheduling synthesis problems for time-triggered networks. However, by using formal verification techniques for this kind of problems, the main disadvantage of model checking is the state explosion that can occur if the system being verified has many components. If these components perform many transitions in parallel, the number of global system states grows exponentially. As the precedence graph is used as the basis for the task and message schedule calculation, an increasing number of tasks and/or messages results in an exponential increase of the underlaying state space. In order to increase scalability, an approach for state space reduction is needed to be explored for finding a solution to the task and message scheduling problem.

State-space reduction techniques can be grouped into two classes [BH98]: automatic and semiautomatic techniques. Automatic techniques are those, in which substructures can be detected and reduced with no manual intervention (e.g. partial order reduction [GCG+99, God91], or the usage of OBDDs [Ran92]). Automatic techniques, as the usage of OBDDs, allowing for an efficient symbolic representation of the model, combined with dynamic reordering strategies may accelerate the process of computation and increase the efficiency. Effects using these automatic techniques are investigated in section 8.3. Semiautomatic techniques uses some degree of manual effort to identify substructures that can drastically reduce the state space. In this section we present a semiautomatic technique focusing on the specific problem of scheduling synthesis to effectively reduce the underlying state space.

We propose such an approach, named *Weighted Symbolic Scheduling*, which provides a heuristic that effectively decreases the state space by guiding the exploration, and which therefore increases the scalability of the symbolic task and message scheduling method.

6.1 Goal

The model checker analyzes the specified task and message scheduling model $\mathcal{M}$ (cf. section 5.3) with respect to a given property φ. Our goal is to show that φ does not hold for every reachable state s in $\mathcal{M}$, as stated in section 5.1. In this case, the model checker outputs a counterexample for the property φ, that is a path $\pi(s_0, ..., s_n)$ such that $I(s_0)$ and $\neg\varphi(s_n)$. The length $len(\pi)$ of such a counterexample is given by the number of steps in this path. The goal of the *Weighted Symbolic Scheduling* approach is to decrease the states of the explored state space, while finding the desired path π and the final state s_n. The perfect algorithm would choose the transition that continues a path leading to that desired goal state s_n, namely the optimal task and message schedule. Unfortunately, this cannot be known a priori. Therefore, a mechanism is proposed that increase the efficiency while exploring the state space.

Generally, there are two approaches on how to decrease the state space during exploration [Boe07]:

1. Execute all active transitions of a global state s. Continue by searching the most "suspicious" state of all resulting global states s' in terms of finding for the solution under consideration. Continue with this "suspicious" one and refrain from the others

2. Define a "suspicion-measure" (e.g. a weight ω) for all active transitions. This measure should be based on a criterion, which is run-time independent. Thus, the weight of a transition can be computed off-line and is valid for all explorations of the model.

For the *Weighted Symbolic Scheduling*, we follow the second approach, using a weight that gives a criteria for selecting the next transition, because the weight of a transition can be computed off-line and is valid for all explorations of the model. Unlike the first approach, transitions with low weights are not executed at all, thus saving time and memory. Figure 6.1 gives an exemplary illustration.

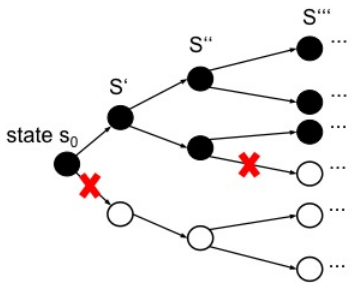


Figure 6.1: State Space Reduction Technique

States that have been explored so far are represented by black circles, unvisited by white ones. Thus, the goal is to guide the exploration and detect unnecessary branches to be "cut-off". This decreases the number of visited states, as visualized in figure 6.1.

6.2 An approach for state space reductions

The *Weighted Symbolic Scheduling* introduces a weighted approach to effectively reduce the underlying state space. It is based on a heuristic for transition selection through efficient off-line weight calculations which is used during run-time to make a selection on all active transitions. Unnecessary branches can be detected by that calculation and "cut-off" and thus, the number of global system states is decreased. This might result in an improved scalability, namely an increasing number of task and/or messages that can be handled by the presented approach.

The basic idea of detecting unnecessary branches in the explored underlying state space is based on the precedence graph traversal. As the presented approach traverses the precedence graph from source to sink, we define a heuristic for transition selection. This selection prefers the longest path through the precedence graph in a situation of concurrency. All alternative possibilities are not explored. In case of the given example 5 (compare also figure 6.2), the different weights are calculated according to the presented weight calculation formula. Thus, for this given example, the branch using $Task1$ (weight $w_1 = 7$) as the starting node is investigated. The branch, starting with $Task0$ (weight $w_0 = 5$), is cut-off and not explored anymore. Therefore the underlying state space is reduced. In the following, we describe the transition selection formally:

Consider a set of transitions $\mathbf{T} = \{tr_1, tr_2, ..., tr_n\}$. Equation 6.1 formalizes the set $\mathbf{T}_{next}$ of transitions that may be selected as the next transition tr_{next}.

$$\mathbf{T}_{next}(s_i) = \left\{ tr_i \mid tr_i \in activeTr(s_i) \right\} \tag{6.1}$$

where $activeTr(s_i)$ denotes all transitions tr_i in a state s_i that fulfill their guard and can be activated. Normally, these *active transitions* are executed non-deterministically.

In order to determine unnecessary branches, which arise from *active transitions*, we are focusing on situations of concurrency that are characterized by the access to a shared resource. As mentioned in section 5.2, we are using the model checker capabilities to explore all interleaved possibilities in situations of concurrency (e.g. the request for a nodes' CPU by several tasks at the same point in time). Because these interleaved possibilities cause state space explosion, we introduce an off-line weight calculation for choosing the next transition tr_{next} by the usage of a heuristic.

Therefore, each task t_i in the precedence graph $\mathcal{G}$ is given two more different parameters:

- *weight:* `integer`, stores the calculated weight of a task t_i
- *locked:* `boolean`, indicates whether the task is locked or not

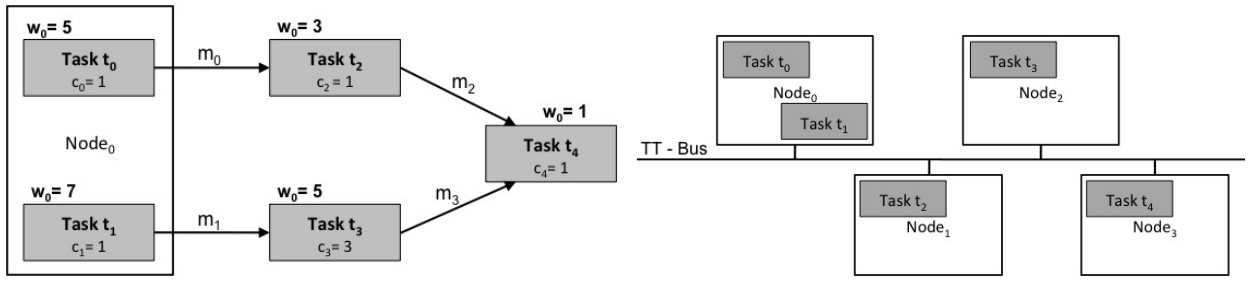
The off-line weight calculation is defined as a depth-first search (DFS) algorithm [CLRS01] using the following weight calculation formula:

$$weight(t_i) = \begin{cases} comp(t_i) & , \text{ if } t_i = \textit{leaf} \\ comp(t_i) + \left[\max_{\mathcal{D}(t_{i,m}) \wedge m_i \in \tau(t_i)} weight(t_{i,next}) + msg \right] & , \textit{otherwise} \end{cases} \tag{6.2}$$

$leaf$ denotes the sink task(s) in the given precedence graph $\mathcal{G}$, $comp(t_i)$ corresponds to the computation time of the certain task t_i. $t_{i,next}$ denotes all tasks t_x that are successor tasks from a task t_i that fulfills $\mathcal{D}(t_i, m_i) = t_x$ and msg denotes the number of messages send by the task t_i to the successor task t_j. Each message is represented as a single time unit. For each task in the given precedence graph a certain weight is calculated. This weight corresponds to the longest path taken from that certain task to a possible sink task. Thus, the heuristic weight decides for the next transition on the bases of a longest path calculation.

In order to emphasize the weight calculation, we give a short example in the following (cf. example 5).

Example 5 Consider a set of tasks $T = \{t_0, t_1, t_2, t_3, t_4\}$, and a set of messages $M = \{m_0, m_1, m_2, m_3, m_4\}$, cf. figure 6.2a. Furthermore, $\tau(t_0) = \{m_0\}$, $\tau(t_1) = \{m_2\}$, $\tau(t_2) = \{m_2\}$, $\tau(t_3) = \{m_3\}$. As illustrated in figure 6.2b, the tasks are allocated to the different nodes: $\eta(Node_0) = \{t_0,t_1\}$, $\eta(Node_1) = \{t_2\}$, $\eta(Node_2) = \{t_3\}$ and $\eta(Node_3) = \{t_4\}$. This exemplary architecture is depicted in figure 6.2b. For simplicity reasons, we just use time units, and assume the following computation time for the tasks: $c_0 = 1$, $c_1 = 1$, $c_2 = 1$, $c_3 = 3$, $c_4 = 1$. The destination function is given as $\mathcal{D}(t_0,m_0) = t_2$, $\mathcal{D}(t_1,m_1) = t_3$, $\mathcal{D}(t_2,m_2) = t_4$ and $\mathcal{D}(t_3,m_3) = t_4$, cf. figure 6.2a.



(a) Simple Example of a Precedence Graph $\mathcal{G}$ (b) Architecture of Precedence Graph $\mathcal{G}$

Figure 6.2: Simplified Architecture of the given Precedence Graph

Figure 6.2a shows a precedence graph with five different tasks. Each task in the precedence graph is equipped with a computation time c_t and its calculated weight w_i, according to the introduced formula 6.2.

The proposed approach is working by using the introduced weight calculation $weight(t_i)$ for selecting the next transition $\mathbf{T}_{next}(s_i)$:

$$\mathbf{T}_{next}(s_i) = \left\{ tr_i \in s_i \mid tr_i \in activeTr(s_i) \ \wedge \max_{tr_i \in activeTr(s_i)} weight(t_i) \right\} \tag{6.3}$$

This can be done while constructing gradually a schedule, beginning in a state where no task has started and no messages have been sent on the bus yet and proceeding one step at a time assigning starting times to tasks and slot positions to messages. In the concurrent situation where two different tasks are trying to start on the same computing resource, the task with the maximum $weight$ is preferred, while the start of the lower weighted task(s) are declined. The $locked$ parameter is used to indicate these lower weighted tasks. At runtime the $locked$ parameter of the task with the highest weight is set to `false`, while the $locked$ parameter of the other tasks are set to `true`. For tasks with equal weights ($weight(t_i)$ = $weight(t_j)$), the corresponding locked parameter are set to `false`. These tasks, whose $locked$ parameter is set to `true` are not started at all from that current state. In words of the precedence graph, the longest path through the network is preferred in concurrent situations and all alternative possibilities are not explored. Therefore the underlying state space is reduced.

SAL Representation

In SAL, the heuristic approach is defined in dedicated data structures, in particular, a heuristic record.

Heuristic Record

```
heuristicrecord:TYPE=
[#
 weight:NATURAL,
 locked:BOOLEAN
#];
HEURISTICARRAY: TYPE = ARRAY TASKS OF heuristiccalendar;
```

Figure 6.3: Declaration of a heuristic records

The defined `heuristicrecord` stores the parameter information of a single single task t_i. The `locked`-parameter is used during state space exploration, indicating whether a task should be blocked due to its lower weight calculation. This depends on the detection of a concurrency. The variable `weight` indicates the off-line calculated weight for each task. For the given example 5 the calculated weights are initialized as a global array `heuristicarray`.

```
GLOBAL heuristicarray:HEURISTICARRAY
heuristicarray[Task0].weight=5.0;
heuristicarray[Task1].weight=7.0;
heuristicarray[Task2].weight=3.0;
heuristicarray[Task3].weight=5.0;
heuristicarray[Task4].weight=1.0;
```

Furthermore, a `StartHeuristic` function is implemented. This function detects whether a current situation might occur in a certain state s. If this is the case, that is when task t_i and task t_j are trying to allocate the same computing resource, the lower weighted task is blocked for execution (e.g. `HEURISTICARRAY WITH [t].locked:=TRUE`). This `locked`-variable prohibits an execution of the previously active transition. In order to detect and potentially lock (prohibit) the start of a task, the function obtains five different parameters: the current task t, the task record array `TASKARRAY`, the node record array `NODEARRAY`, the message record array `MESSAGEARRAY`, and the heuristic array itself `HEURISTICARRAY`. If there is no situation of concurrency the `HEURISTICARRAY` remains unchanged. The function implementing the heuristic is depicted below.

```
StartHeuristic(t:TASKS, taskarray:TASKARRAY, nodearray:NODEARRAY,
           messagearray:MSGARRAY, heuristicarray:WEIGHTARRAY):WEIGHTARRAY=
IF((EXISTS (j:TASKS): j /= t AND (nodearray[getNodeTask(j)].node_free=TRUE
  AND taskarray[j].t_started = FALSE
  AND getPrecMsg(j,messagearray)=TRUE)
  AND (getNodeTask(j)=getNodeTask(t))
  AND (heuristicarray[j].weight>heuristicarray[t].weight)
))
THEN heuristicarray WITH [t].locked:=TRUE
ELSIF ((EXISTS(j:TASKS):j/=t AND (nodearray[getNodeTask(j)].node_free=FALSE
  AND taskarray[j].t_started = TRUE
  AND taskarray[j].t_finished=FALSE
  AND getPrecMsg(j,messagearray)=TRUE)
  AND (getNodeTask(j)=getNodeTask(t))
  AND (heuristicarray[j].weight>heuristicarray[t].weight)
))
THEN heuristicarray WITH[t].locked:=TRUE
ELSE heuristicarray
ENDIF;
```

This function is needed by the two different transitions that might be able to start a task and thus may cause a concurrent situation (the Start Task Transition T_{st} and the Change Task Transition T_{ct}). Thus, it is used as a condition request inside these transitions, detecting whether a current situation might occur in a certain state s.

```
([](i:TASKS):
 starttransition:
 ...
 StartHeuristic(i,currenttaskarray,currentnodearray,
        currentmessagearray,heuristicarray)[i].heuristic_locked = FALSE
 -->
 ...
)
```

In concurrent situations the task with the maximum $weight$ is detected and preferred, while the start of all other task(s) is declined. The $locked$ parameter is used to block all transitions (brances) with lower weights. Thus, these transition (corresponding to the "branches of a precedence graph") are not started at all and thus are "cut-off". This reduces the underlying state space, while calculating the integrated task and message schedule.

6.3 Results and Discussion

The efficiency of the *Weighted Symbolic Scheduling* can be identified by comparing the same example with and without using the state space reduction approach. Therefore, we use the given example 5 in section 6.2. We invoke first SAL's symbolic model checker `sal-smc` on this example without using the new heuristic-based approach. SAL outputs a counterexample including some interesting properties, as listed below. The model checker explores 1237500 states with a total execution time of 27.923 seconds.

(Symbolic Task and Message Scheduling - without heuristic approach)

```
verification time: 13.907 secs
total execution time: 27.923 secs
number of visited states: 1237500.0
```

The same example is invoked using the *Weighted Symbolic Scheduling* approach. As expected, `sal-smc` outputs a counterexample with smaller number of visited states. The model checker explores 96875 states for finding the same solution. This complies to a reduction of visited states of about 92%.

(Weighted Symbolic Scheduling - including heuristic approach)

```
verification time: 11.203 secs
total execution time: 24.422 secs
number of visited states: 96875.0
```

This enormous reduction in states can be explained as follows. As the weighted heuristic approach searches for situations of concurrency in order to "cut-off" identified unnecessary branches, the point in time of identification has a major impact: While the *Weighted Symbolic Scheduling* traverses the precedence graph from source to sink, a situation of concurrency in the first steps / states has a higher impact on the reduction of the underlying state space. This

happens, because the branches that are identified as unnecessary, reducing a lot more states, compared to a concurrent situation in the last steps of the precedence graph traversal. These reduced states comply to possible alternative schedules. In the given example 5, this effect takes place, as situations of concurrency occurs in the very first state. The reduction of 92% of visited states can be explained by that fact. Thus, in contrast, the presented approach would cause no improvements for a sequential precedence graph, as the obtained schedule has no concurrent resource constraints.

The usage of the introduced mechanisms for *Weighted Symbolic Scheduling* enables to define a *lower scheduling bound*, cf. equation 6.4. There is no shorter schedule possible than

$$\mid Schedule \mid \geq \max\ weight(t_i) \quad \forall\ t_i \in T \tag{6.4}$$

The notion $\mid \ Schedule \ \mid$ complies to the total length of a designed schedule. This obtained lower bound complies to the longest path in the precedence graph. This is calculated a priori, as explained above. Experimental results demonstrate that the *Weighted Symbolic Scheduling* can be effectively used to reduce the underlaying state space. Average improvement results are depicted in section 8.3.6.

However, using the *Weighted Symbolic Scheduling* approach and the presented heuristic for transitions selection, a probability of cutting-off the wrong branch still remains. Obviously, in this case, this lead to negative effects on the schedule length in terms of obtaining a longer schedule than the optimal schedule. The probability of these cases is hard to determine. Nevertheless, there is still a possibility of using the *Symbolic Task and Message Scheduling* approach, in order to check, whether the heuristic works perfect or not. However, this is only possible, in cases the *Symbolic Task and Message Scheduling* approach is still capable of finding a schedule.

Chapter 7

Framework for Scheduling Synthesis

This chapter proposes a framework for Scheduling Synthesis, which integrates the presented approaches of an *Algorithm for Integrated Task and Message Scheduling* (chapter 4), *Symbolic Task and Message Scheduling* (chapter 5) and *Weighted Symbolic Scheduling* (chapter 4) in order to find an (optimal) solution of the given task and message scheduling problem.

The framework is composed of different elements. It integrates the complexity evaluation approach (section 8.1), used to investigate various precedence graph parameter configurations and their correlation to the calculated task and message schedules. Furthermore, a precedence graph generator is included, which is able to generate precedence graphs, including all necessary parameters (e.g., number of nodes, tasks, precedence relations, etc.). This is done, in order to obtain general evidence about certain parameter variations (cf. section 8.1). The graph generation can be done explicitly (e.g. according to a given aeronautic application) or according to a given set of parameters. In the latter case, numerous graphs can be generated following the specified parameter set. Furthermore, the framework consists of an interface to import certain communication parameter sets (e.g. a FIBEX [fSoAS04] FlexRay configuration).

In order to calculate task and message schedules using the *Symbolic Task and Message Scheduling* approach, a code generator is implemented, that automatically translate the precedence graphs into SAL specifications according to the symbolic encoding scheme (presented in chapter 5). Furthermore, the framework has an interface to the SAL framework for executing the generated SAL specifications. The calculated results are transferred back. Experiment series, that are precedence graphs following a certain parameter set, can thus be computed and compared. For visualization, textual as well as gnuplot-based graphical elements are available.

7.1 Graph Generation

The frameworks' graph generation functionality generates precedence graphs, including all necessary parameters. The graph generation can be done explicit as a Single Graph Generation (cf.

subsection 7.1.1) or according to a given set of parameters, called Multiple Graph Generation (cf. subsection 7.1.2). In the latter case, numerous graphs can be generated following a specified parameter set. In the following both types of graph generation are highlighted in detail.

7.1.1 Single Graph Generation

The Single Graph Generation enables to generate a certain precedence graph according to the users needs. This can be an aeronautic use case system, with a fixed setting, namely certain number of tasks, their allocation to nodes and given precedence relations.

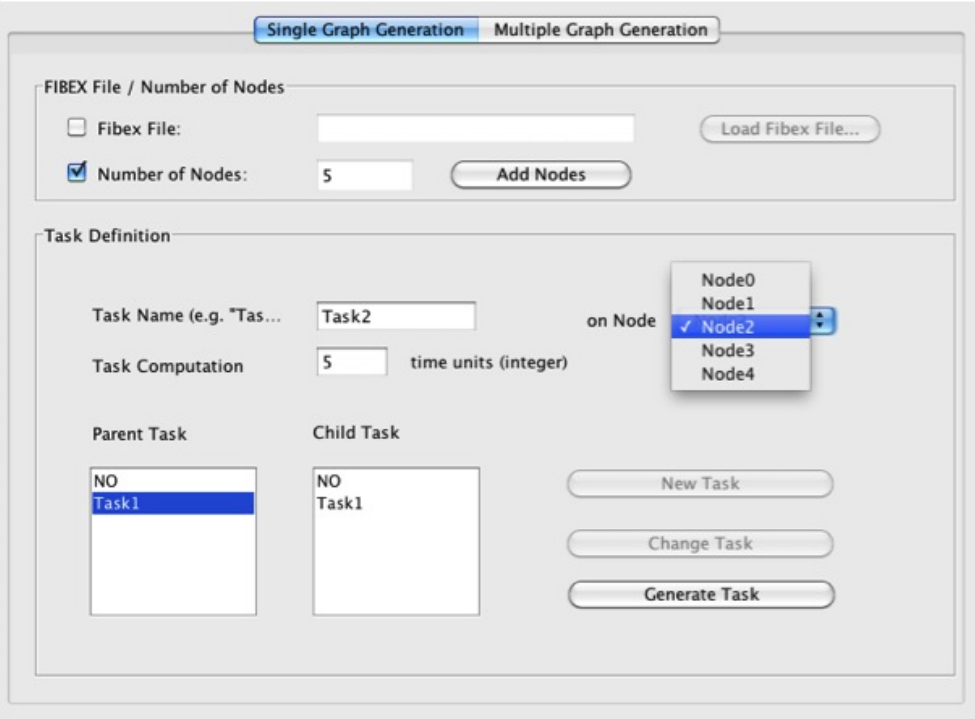


Figure 7.1: Single Precedence Graph Generation

First, a network architecture needs to be specified. Network architecture in this context means the definition of computing resources (nodes). It is implicitly assumed that the given set of nodes communicate via a shared time-triggered communication bus, meaning that all computing resources are connected to that shared communication resource.

We implemented two different ways to specify a network architecture for the Single Graph Generation. On the one hand, the number of nodes can be specified by a certain input field. Figure 7.1 illustrates a use case system with 5 different nodes. On the other hand, a configuration file can be imported, which already consists of information about the network architecture. As we are using a time-triggered communication bus, the framework supports the import of FIBEX [fSoAS04]

configuration files. This standard enables the configuration and management of a subset of different communication network architectures (e.g. CAN [fS98], LIN [LK03], FlexRay [Con05], etc.) [Ruh05].

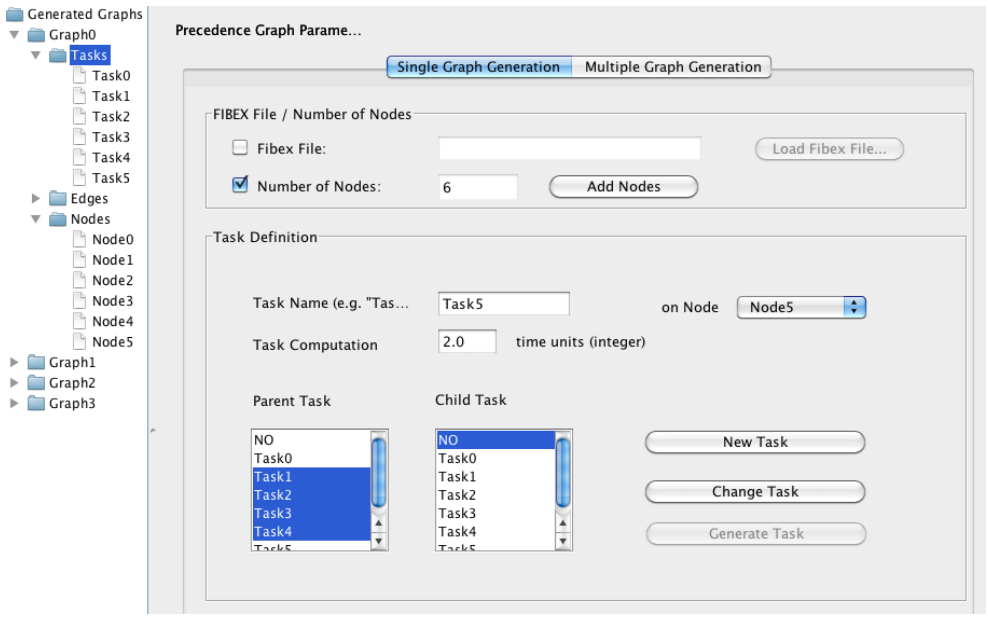


Figure 7.2: Defining the Precedence Relations for each Task

In a next step the precedence graph can be specified. This includes the definition of a set of tasks and their precedence relations. Figure 7.1 illustrates the definition of a certain task (e.g. *Task 2*). This includes the definition of the following parameters: the expected task computation and an allocation of the given task to a specific node. Furthermore, the precedence relations can be specified by defining one or many parent (predecessor) or child (successor) tasks. This is illustrated in figure 7.2. Thus, a precedence graph is built iteratively.

7.1.2 Multiple Graph Generation

The Multiple Graph Generation enables to generate a set of precedence graphs according to a specified parameter set. This set describes certain characteristics of the generated precedence graph with respect to the number of tasks, specific task parameters, as well as the precedence graph layout. The Multiple Graph Generation is part of the complexity evaluation approach, introduced in section 8.1, which can be used to evaluate and identify , whether a given parameter configuration complies with the given system requirements.

In order to specify a set of precedence graphs, we first define the number of tasks. Figure 7.3 illustrates the generation of precedence graphs consisting of 5 tasks. The number of messages (precedence relations) is calculated by:
number of messages = number of tasks - 1. This equation guarantees that the precedence graph

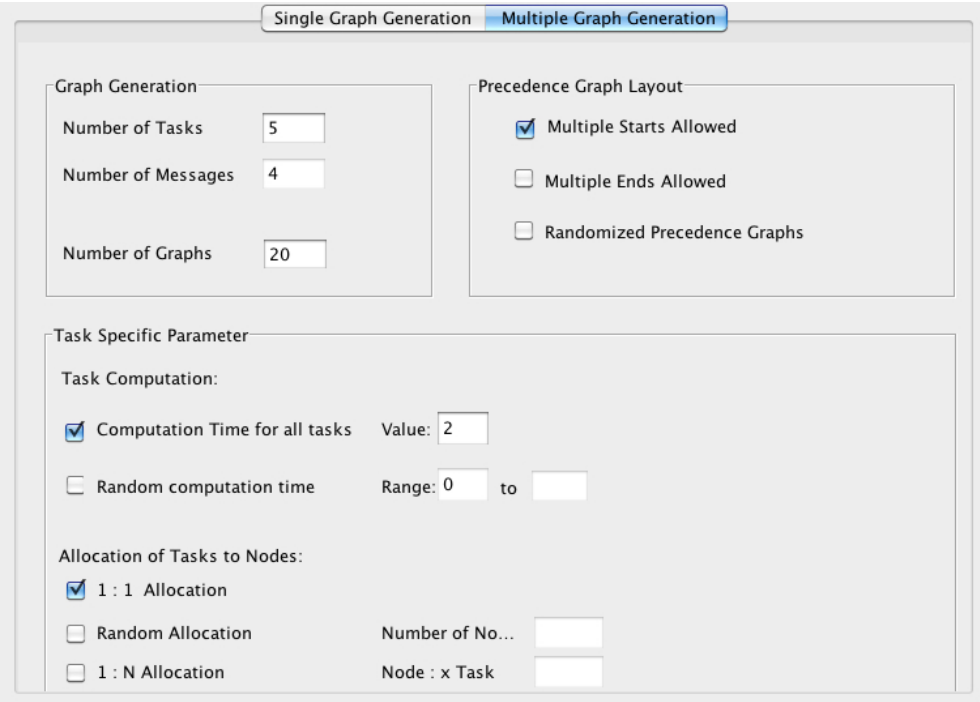


Figure 7.3: Multiple Precedence Graph Generation

is connected. Furthermore, it is prohibited to use duplicated precedence relations (two different relations from task t_i to task t_j) as well as the generation of cyclic precedence graphs. Finally, the number of generated precedence graphs using the defined characteristics is specified.

The generation of multiple precedence graphs requires a random-number generator in order to randomly generate the precedence relations. We use a generator from [LSCK01], which follows the basic design principles and methods for uniform random number generators, as described in [L'E10]. Nevertheless, the generation of precedence graphs follows a precedence graph layout, describing the general structure of the precedence graph.

The precedence graph layout is specified with respect to aeronautic use cases. In the following, the different scenarios are specified:

Precedence Graph Layout

We define three different precedence graph scenarios:

1. **Multiple Starts**: The first scenario complies to a acyclic directed precedence graph in which a number of nodes do not have any predecessors. These nodes are called source nodes. One further characteristic of this scenario is that all other tasks need to lead into a single sink task. A precedence graph with multiple source nodes is schematically shown in figure 7.4. This scenario might comply to a system situation in which different sensors are

depicted over an airplane measuring sensor values and transmit them to a server or various computing nodes.

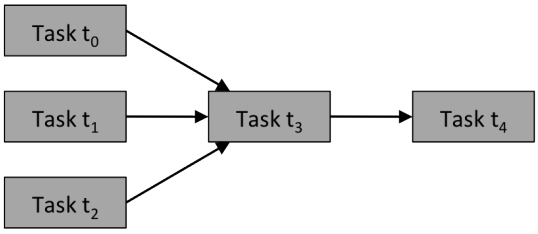


Figure 7.4: Precedence Graph with Multiple Starting Nodes

2. **Multiple Ends**: This scenario complies to an acyclic directed precedence graph in which a number of nodes do not have any successors. These nodes are called sink nodes. This scenario further requires that there exists just a single source task. A precedence graph with multiple sink nodes is schematically shown in figure 7.5. This scenario might comply to a system situation in which different actuators are depicted over an airplane which need to be controlled by one or several source nodes.

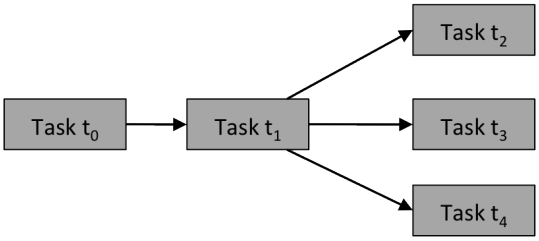


Figure 7.5: Precedence Graph with Multiple Ending Nodes

3. **Randomized**: The randomized scenario can be described by an acyclic directed precedence graph which is generated completely at random. A random precedence graph allowing both multiple ending nodes and starting nodes or reconvergences. Figure 8.8a illustrates a random precedence graph.

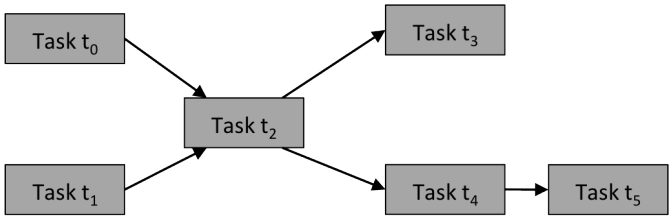


Figure 7.6: Precedence Graph with connected randomized nodes

Task Specific Parameter

The multiple graph generation requires the definition of further task specific parameters, in order to describe the set of tasks in more detail. The computation time for each task is specified by using the same time for all given tasks (as illustrated in figure 7.3) or by selecting a random computation time, which is bounded by an upper and a lower bound. The computation time is given by an integer number, representing time units.

Furthermore, an allocation of tasks to nodes needs to be defined. This is done by using one out of three different possibilities: First of all, a one to one allocation can be specified, meaning each task has its own computing resource (node). Second, a random allocation of the given set of tasks to nodes can be chosen. In this case, the number of nodes needs to be specified. The third possibility complies to a situation in which a fixed number of tasks are allocated to the set of given nodes. This ratio implies the defined number of nodes.

7.1.3 General Parameter for Graph Generation

The Graph Generation is done explicitly as a Single Graph Generation (cf. subsection 7.1.1) or as a Multiple Graph Generation (cf. section 7.1.2). Having specified the graph(s) in the desired way, several general parameters need to be specified.

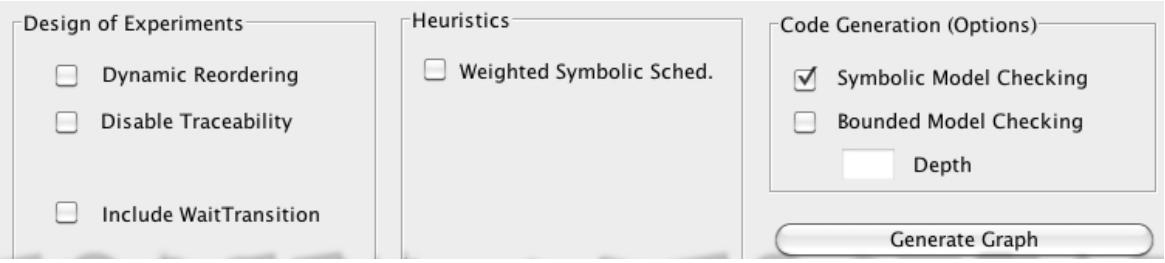


Figure 7.7: General Parameter for Graph Generation

Figure 7.7 illustrates all general parameters. The user is able to choose, whether to use the heuristic *Weighted Symbolic Scheduling* or the *Symbolic Task and Message Scheduling*, by selection. The *Weighted Symbolic Scheduling* option enables to use the heuristic approach.

As the framework is able to either use SAL's bounded model checker, `sal-inf-bmc`, or SAL's symbolic model checker, `sal-smc`, the Code Generator (cf. section 7.2) is able to generate code according to the chosen model checker.

Furthermore, the following experimental parameters can be specified: The *Dynamic Reordering* option enables the usage of dynamic reordering strategies to accelerate the process of computation and increasing its efficiency (cf. section 8.3.4). Another parameter can be used to disable the SAL's traceability. The SAL framework creates extra variables to produce detailed information about the path and the counterexamples. The *Disable Traceability* option disables the generation

of such variables, which might lead to probably faster execution. The *Include WaitTransition* option comprises to the usage of an additional transition in the specified model $\mathcal{M}$. The additional `Wait_Message_Transistion` is describend in subsection 5.3.4.7. Results of using this transtion are given in section 8.3.5.

7.1.4 Precedence Graph Editor

The proposed framework enables user interaction at any time. User changes on the precedence graph parameters (e.g. changing the precedence relations of the given task set or the allocation of task to nodes), are supported by the precedence graph editor. This enables to change precedence graph based on previous results. An iterative optimization (usage of results from the calculated schedule) according to given requirements might be able, by such an adjustment in the parameterization of a precedence graph.

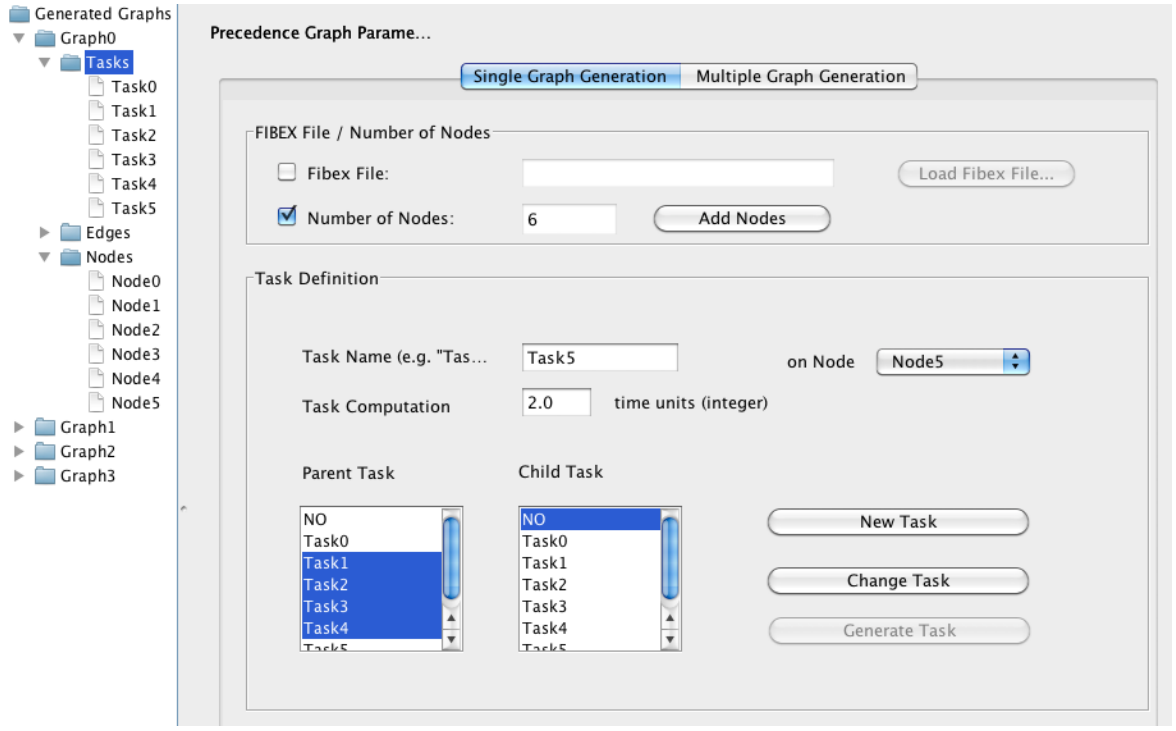


Figure 7.8: Precedence Graph Editor

Figure 7.8 illustrates the precedence graph editor for a certain graph (Graph0). As demonstrated, task *Task5* is highlighted here, consisting of 4 different predecessor tasks and no successor task. Changes in the precedence relations are done easily.

7.2 Code Generation

The frameworks' code generation functionality automatically translates the generated precedence graph(s) into SAL specifications. The code generation can be done according to the *Symbolic Task and Message Scheduling* (cf. chapter 5) or according to the heuristic *Weighted Symbolic Scheduling* (cf. chapter 6).

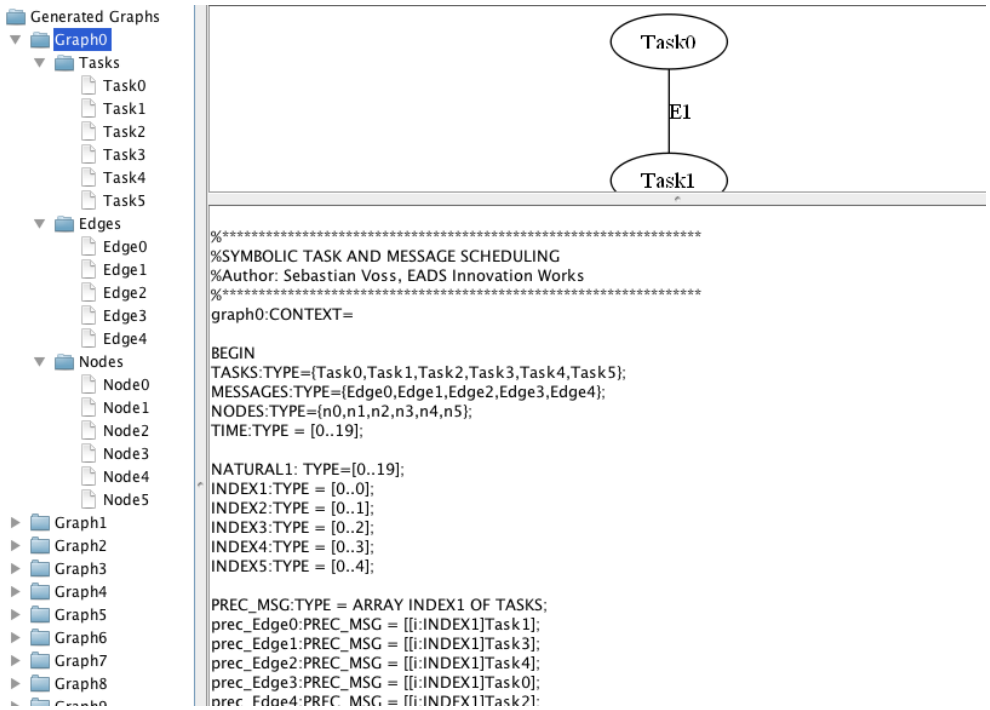


Figure 7.9: Graphical Representation and Code Generation

We take the formal model of the system, as specified in section 5.3, as a basis for translating the scheduling synthesis problem to SAL specifications. This includes declaration, initialization and transition system. The process on how to construct a SAL module, containing the definition of variables, the initial state and the transition relations is described in section 5.4. Depending on the selected model checker (cf. section 7.1.3), the property φ is generated, as specified in section 5.7. Figure 7.9 illustrates the generated code for each precedence graph. This enables to double check the relevant parameter before transferring the file to the model checker.

7.3 Result Generation

The framework has an interface to the SAL framework for analyzing the specified model $\mathcal{M}$ with respect to the given property φ. The calculated results are transferred back. This is done for a single precedence graph (e.g. aeronautic use case) as well as for experimental series.

Figure 7.10 shows how the results are illustrated and configured in the given framework. The model checker calculates the optimal task and message schedule and transfer it back the framework. All results, namely the calculated counterexamples, are stored in a folder, which needs to be chosen (cf. figure 7.10).

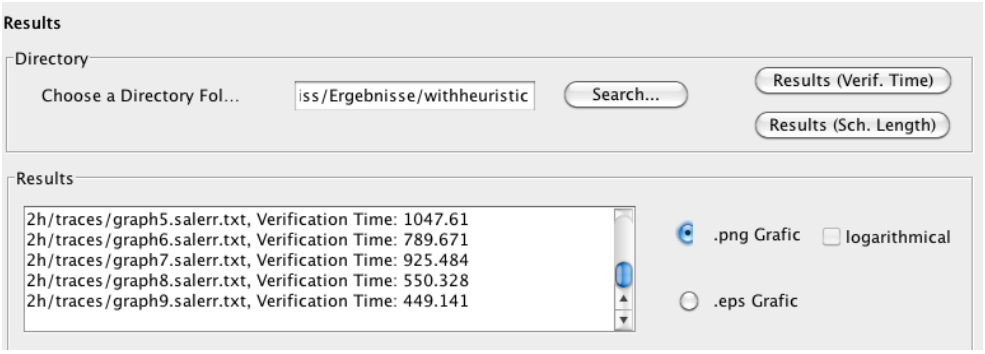
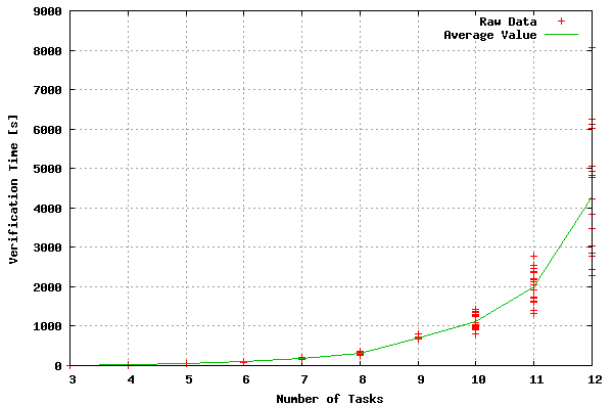


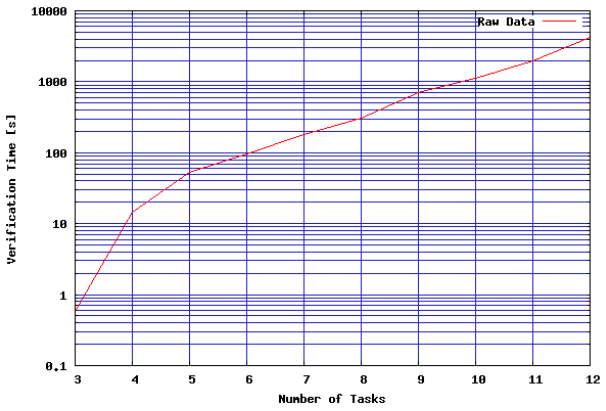
Figure 7.10: Result Generation for Symbolic Task and Message Scheduling

The framework is able to generate two different kind of results automatically: On the one hand the computation time can be extracted from all specified counterexamples. On the other hand the calculated scheduling length is extracted by the results generation capability. This is done by using regular expression running over the given counterexamples. In case the Multiple Graph Generation is used, the exact values are computed, as well as the average computation time value for the corresponding generated graph. Further parameters are calculated as well, e.g. standard deviation (cf. chapter 8). All calculated values are written in an Excel-based file as well, in order to provide a data basis for other calculations.

Textual (cf. figure 7.10) as well as gnuplot-based graphical elements (cf. figure 7.11) are provided for visual representation of the calculated results.



(a) Visual representation of experimental series



(b) Logarithmical representation of experimental series

Figure 7.11: Visual representation of calculated computation time

The calculated computation time is depicted in figures 7.11a and 7.11b, whereas the graphical scheduling length representation is illustrated in figure 7.12.

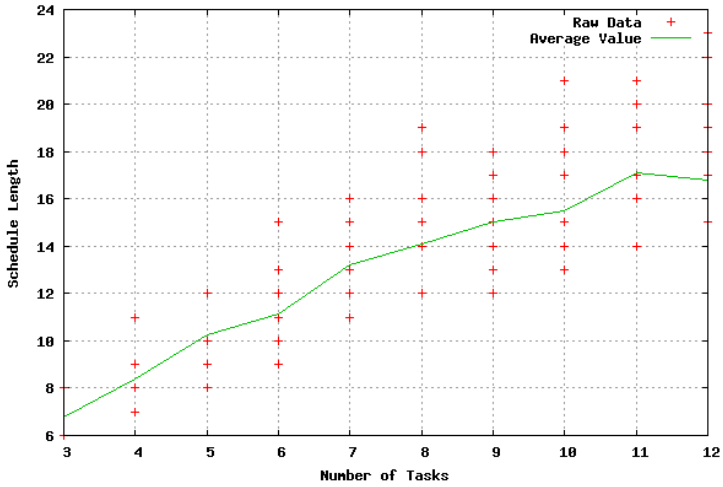


Figure 7.12: Visual representation of calculated schedule length

Chapter 8

Analysis and Results

This chapter proposes an approach for evaluating the performance and the complexity of the integrated task and message scheduling algorithms, namely the *Symbolic Task and Message Scheduling* and *Weighted Symbolic Scheduling* approaches (cf. chapter 5 and 6). This approach enables to quantify various precedence graph configurations in terms of complexity and performance evaluation. The developed evaluation approach is integrated in the framework presented in chapter 7. Furthermore, the performance and usability of the developed Weighted Symbolic Scheduling approach compared to the Symbolic Task and Message Scheduling can be quantified.

Experimental results demonstrate that Symbolic Task and Message Scheduling, as well as Weighted Symbolic Scheduling are suitable for finding an optimal task and message schedule in terms of minimizing the end-to-end latency. We have implemented both approaches using the SAL framework (cf. section 2.4). All experiments presented in this section use SAL's symbolic model checker `sal-smc`. SAL uses Yices [DdM06] as the default solver which integrates a SAT solver and decision procedures for a combination of logical theories.

8.1 Complexity evaluation for scheduling configurations

The proposed complexity evaluation approach is designed for two different evaluations: On the one hand parameter configurations can be evaluated regarding their usability in the given system design. The developed approach enables to identify, whether a given parameter configuration complies with given system requirements (e.g. end-to-end latency). On the other hand, we are able to quantify both presented approaches. This highlights the benefits of the developed Weighted Symbolic Scheduling approach compared to the Symbolic Task and Message Scheduling approach. Results demonstrate the performance of the developed approaches in terms of computation time, schedule length and scalability.

In order to give general statements about the usability, scalability and comparability of the developed approaches, we investigate different parameter configurations or variations. A parameter

configuration of a given precedence graph is characterized by the following parameters: number of nodes, number of tasks per node and precedence graph layouts or shapes. A precedence graph layout is characterized by the following scenarios: Multiple Starting Nodes, Multiple Ending Nodes or complete randomized precedence graphs. These scenarios are highlighted in section 7.1.

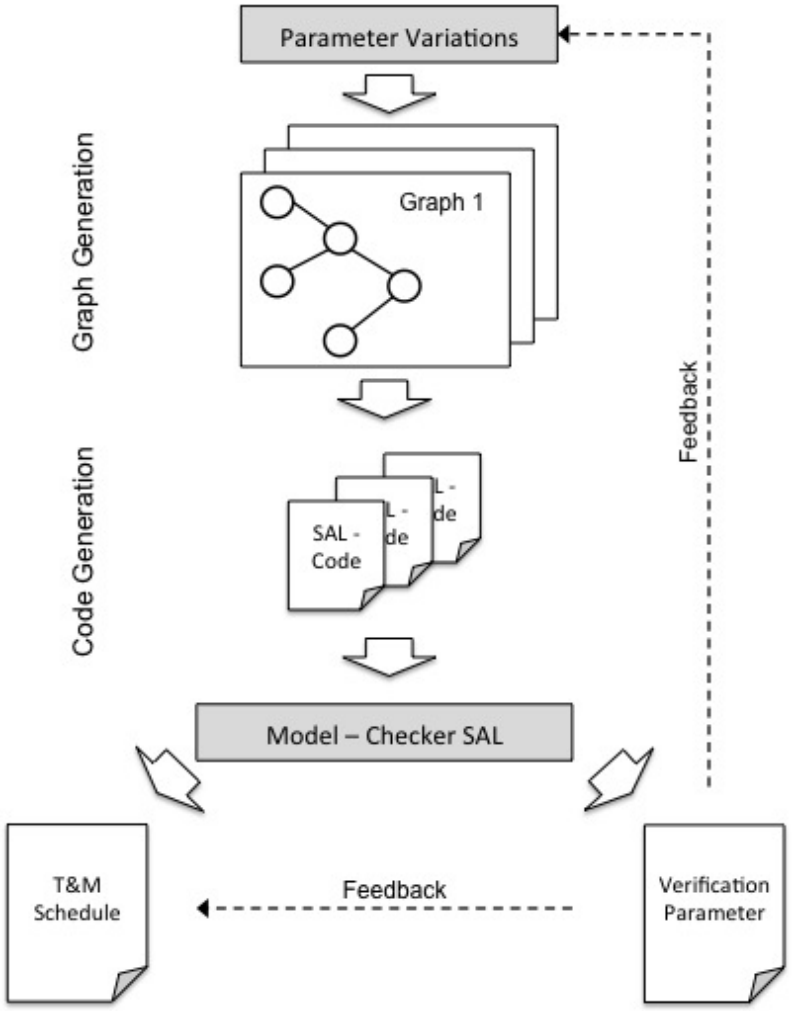


Figure 8.1: Procedure of Complexity Evaluation

We are starting from different parameter variations (cf. figure 8.1). The frameworks' graph generation capability (cf. section 7.1) is then used to generate a set of precedence graphs according to the defined parameter variations. In a next step, each of these generated precedence graphs are automatically translated into SAL specifications according to the symbolic encoding scheme using the Code Generator presented in section 7.2. Each precedence graph generates its own SAL-Code file according to one of the specified scheduling approach. These code files are incrementally transfered into the SAL framework. The model checker analyze the specified task and message scheduling model $\mathcal{M}$ with respect to a given property φ, stating that there is

no possible schedule fulfilling the given system requirement. The model checker returns either *verified* or *falsified*, depending whether the given property is fulfilled by the model or not. In the latter case, the model checker usually outputs a counterexample from which a task and message schedule can directly be obtained. To find such a counterexample we can either use SAL's bounded model checker [dMRS03], `sal-bmc`, or one of SAL's symbolic model checker, `sal-smc` or, `sal-wmc` [SS03]. Experimental results in this chapter uses the symbolic model checker exclusively. To obtain an optimal task and message schedule, we further use a binary search for finding that solution (compare section 5.8).

From the obtained optimal task and message schedule we are able to draw conclusions as presented in the following: We use the calculated schedule length as well as output information from the SAL framework, namely *computation time* and the *number of visited states* to illustrate the following three aspects:

First, for a single precedence graph evaluation (e.g., aeronautic system use case), the feasibility of a given parameter configuration can be checked with respect to the given system requirements. For instance, the required cycle time of an aeronautic system can be compared to the optimal calculated schedule length. In case of any inconsistencies (e.g. an calculated task and message schedule is not able to meet the system requirement), this would enable to refine the given parameterization (e.g. the number of tasks allocated to a certain node).

Second, we are able to state the general behavior of precedence graphs using the same configuration scenarios. This can be done by using several generated precedence graphs containing the same parameterization (e.g. a certain number of nodes, number of tasks per node and complying to a given precedence graph layout or shape). This leads to a better understanding of complexity in configuration scenarios using different parameter sets or variations.

Third, a comparison of the developed task and message scheduling approaches with respect to their usability can be done, meaning experimental evaluation of the developed approaches itself. The following experimental results demonstrate that the latest generation of model-checking tools meets the challenges of providing a convenient modeling language as well as the performance to solve given scheduling problems in terms of scalability using Symbolic Task and Message Scheduling and Weighted Symbolic Scheduling. In this context, the presented complexity approach enables to quantify benefits of using the heuristic approach compared to the Symbolic Task and Message Scheduling approach.

8.2 Design of experiments

For experimental evaluation of the developed Symbolic Task and Message Scheduling and Weighted Symbolic Task and Message Scheduling, we define a set of experiments. This set is used to analyze the usability of the developed approaches in terms of performance and scalability.

	Multiple Starts	**Multiple Ends**	**Randomized Graph**
1 Task per Node (1:1)	20 precedence graphs with 3 - 12 tasks	20 precedence graphs with 3 - 12 tasks	20 precedence graphs with 3 - 12 tasks
2 Task per Node (2:1)	20 precedence graphs with 3 - 12 tasks	20 precedence graphs with 3 - 12 tasks	20 precedence graphs with 3 - 12 tasks
4 Task per Node (4:1)	20 precedence graphs with 3 - 12 tasks	20 precedence graphs with 3 - 12 tasks	20 precedence graphs with 3 - 12 tasks

Table 8.1: Design of Experiments

The given set of experiments focus on two parameters: the precedence graph layouts and the ratio of system functionalities (tasks) to computing resources (nodes). The precedence graph layouts are described by the three different precedence graph scenarios (cf. section 7.1): Multiple Starts, Multiple Ends or Randomized Precedence Graph. The first two scenarios represent possible aeronautic architecture use cases: For instance, the Multiple Starts Scenario complies to an architecture using multiple sensors transmitting a sensor value to a set of receiving components (compare section 7.1).

The ratio of tasks to nodes is investigated using different scenarios: The allocation of a single tasks to a node, or scenarios in which a set of concurrent task are allocated to a certain node. This is represented by two and four tasks per node.

For each scenario listed in table 8.1, we randomly generate 20 different precedence graphs complying to the specified parameter set, but using various precedence relations.

8.3 Results

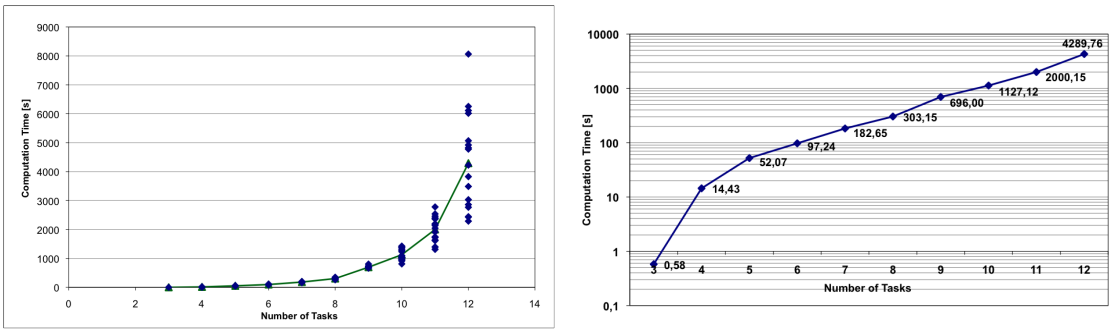
We use the framework of SAL for specification and analysis of concurrent systems (cf. section 2.4). The SAL model checkers are of interest for our analysis. The symbolic model checker `sal-smc` that uses the CUDD BDD package and provides access to many options for variable ordering, and for clustering and partitioning the transition relation. Experimental results in this chapter uses the symbolic model checker `sal-smc` of the SAL framework.

The experiments were carried out on an Intel(R) Pentium(R) 4CPU 2.80GHz and 2.49GB RAM. The experiments illustrate that SAL model checkers can be effectively used to automatically compute schedulers that minimize the transmission latency of the communication medium. Details on the encoding schema can be found in section 5.4.

8.3.1 General Results of Task and Message Scheduling

In order to demonstrate general usability of the Symbolic Task and Message Scheduling approach, we first investigate the scenario using a single task per node with the Multiple Starts precedence graph layout. As depicted in table 8.1, we use 20 random generated precedence graphs using the previous mentioned layout and a ratio of one task per node. Furthermore, the computation duration of a task equals the length of a transmitting slot.

The model checker incrementally processes each precedence graph, in order to automatically compute schedules with minimal end-to-end latency. The model checker outputs a counterexample that contains the integrated task and message schedule and a set of quantitative parameters that can be used for evaluation (cf. section 5.1). Figure 8.2 shows the computation time used by the model checker for finding such a task and message schedule.



(a) Exponential growth of computation time (b) Logarithmic illustration of exponential growth

Figure 8.2: Computation Time for Task and Message Scheduling

All measured computation times are given by the CPU time for computing an integrated task and message schedule using SAL 3.0. These runtimes are given in seconds. The exact computation time for each precedence graph is represented by a single point (cf. figure 8.2a). For every considered number of tasks t, the average computation time value for the corresponding 20 generated precedence graphs can be calculated as well. This average value is depicted by the green line. By increasing the number of tasks in a precedence graph, while leaving the previously defined settings unchanged, demonstrate the exponential growth of computation time necessary to calculate an integrated task and message schedule for the given specification (cf. figure 8.2a and figure 8.2b). Figure 8.2b depicts the computation time in a logarithmical diagram. The computation time complies to the average value of all 20 precedence graph (cf. table 8.2). It can be seen, that starting from precedence graphs containing a minimum of 6 tasks, an exponential correlation can be identified. Thus, for 13 tasks an average verification time of around 10000s can be expected.

The layout of the precedence graph has a major impact on the length of the schedule and thus on the computation time for generating the integrated task and message schedule. A schedule for a sequential precedence graph is computed faster than for highly concurrent precedence graphs.

This is explained by the fact, that by calculating a schedule incrementally, concurrent situations offer a higher number of possibilities. A concurrent situation can either be due to several tasks running on the same computing resource (node) or several message trying to allocate to the same communication slot. All different possibilities have to be explored by the model checker to find the optimal solution. Thus, by increasing the number of tasks in a precedence graph, the amount of concurrent situations might increase as well. This effect can be quantified by the (nearly) exponential increasing value of the mean computation time.

Further interesting values are the standard deviation as well as the coefficient of variation. Especially the coefficient of variation indicates that the dispersion of values to the mean value can be characterized by a statistically expected behavior, as it defines the ratio of standard deviation to the mean value.

Number of Tasks	Computation Time	Standard Deviation	Coefficient of Variation
3	0.58 s	0.03 s	0.52
4	14.43 s	7.45 s	0.52
5	52.07 s	6.82 s	0.13
6	97.24 s	9.05 s	0.09
7	182.65 s	13.47 s	0.07
8	303.15 s	18.06 s	0.06
9	696.00 s	31.93 s	0.05
10	1127.12 s	193.17 s	0.17
11	2000.15 s	414.79 s	0.20
12	4289.76 s	1560.32 s	0.36

Table 8.2: Results of Experiment Series

Another interesting parameter that can be extracted from a counterexample produced by the SAL framework is the *visited number of states* for finding such a counterexample. Figure 8.3 shows the exponential increase of the number of visited states for an increasing number of tasks per precedence graph.

These first experimental results demonstrate that the Symbolic Task and Message Scheduling approach can be used to solve the integrated task and message scheduling problem. Furthermore, it proves how the latest generation of model-checking tools meet the challenges of providing both a modeling language and the performance to solve given scheduling problems.

8.3.2 Relation of Task and Nodes

The previous sections demonstrate that the Symbolic Task and Message Scheduling approach is able to solve the given task and message scheduling problem and finds an optimal solution in terms of minimizing the end-to-end latency. Furthermore, section 8.3.1 has shown that scalability

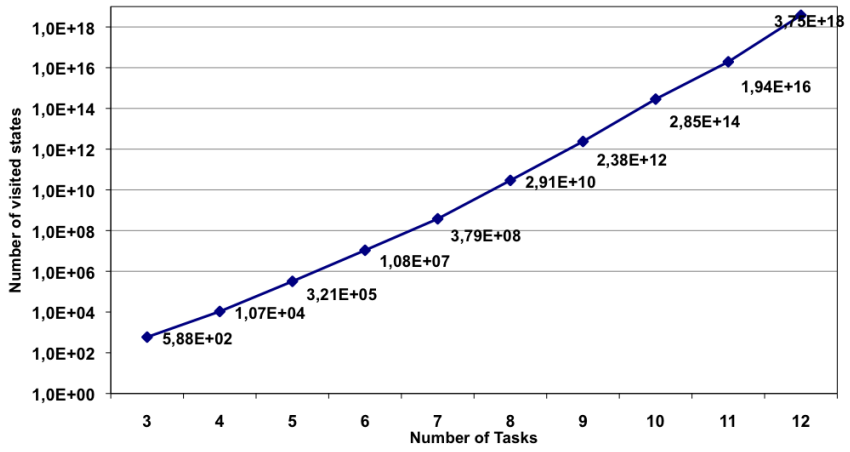


Figure 8.3: Average number of visited states

is a restricting factor. Therefore, we investigate how different relations of tasks to nodes affect the scalability of the presented approaches. We start by investigating various ratios of tasks to nodes.

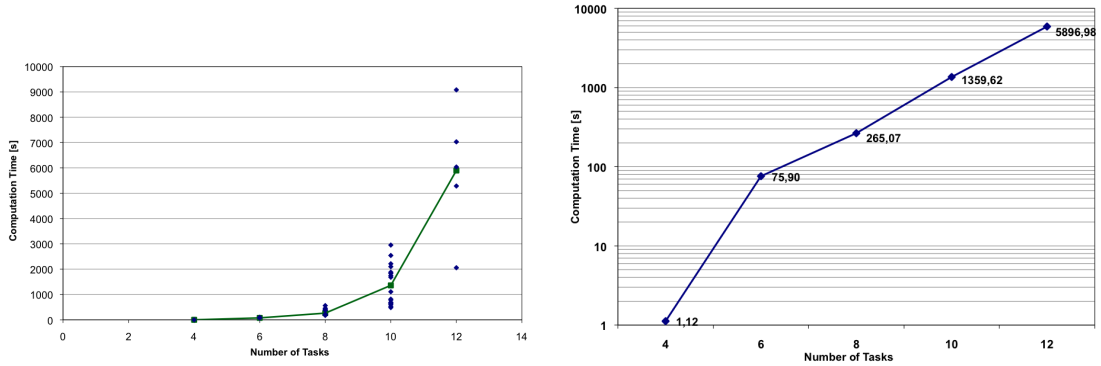
1 Task per Node

We have already shown experimental results for precedence graph layouts with a single task per node and a Multiple Start precedence graph layout. These results are given in the previous section 8.3.1.

2 Tasks per Node

In a next step, we investigate precedence graph layouts with 2 tasks per node. We therefore randomly generate 20 precedence graphs with a number of 4, 6, 8, 10 and 12 tasks on 2, 3, 4, 5 and 6 nodes, respectively.

Figure 8.4a demonstrates the exponential growth of computation time for an increasing number of tasks per precedence graph. For every considered number of task t, the average computation time for the corresponding 20 generated precedence graphs is indicated by the green line. Figure 8.4b shows the computation time in a logarithmical diagram. It can be seen, that starting from precedence graphs containing 6 tasks, an exponential increase in computation time is identified. Thus, for 14 tasks an average computation time of around 30000s can be expected.



(a) Exponential growth of computation time (b) Logarithmic illustration of exponential growth

Figure 8.4: Computation Time for 2 tasks per node

Comparison

Experimental results in this sections demonstrate how various relations of tasks to nodes affect the computation time for finding an optimal schedule using the Symbolic Task and Message Scheduling approach. An exponential increase in computation time can be measured while increasing the number of tasks per node. This effect is based on an increase of the state space. This can be explained by the fact that the crucial point in constructing a schedule is to deal with concurrent precedence graph scenarios, as specified in section 5.1. Concurrency can either be caused by a shared computing resource (node) or a shared communication resource (time slot of the underlying communication bus).

In this section we investigate the case in which one or more tasks use the same computing resource. This might lead to concurrent situations, in which both tasks might access the resource at the same point in time. Therefore, we use the model checker's capabilities for constructing a task and message schedule to explore all specified interleaved possibilities, as already specified in section 5.1. Thus, an increase of computation time for constructing an optimal task and message schedule can be explained by this fact.

However, an exponential increase in computation time affects scalability of the presented approaches as well. In order to quantify this effect, we integrate the presented experimental results for single task per node with two tasks per node, as illustrated in figure 8.5:

In order to make both scenarios comparable, the precedence relations remain the same. Just the number of nodes has been reduces to 1/2. The allocation of task to nodes is randomly generated. Thus, comparing the average computation time for generated precedence graphs using a single task per node and two tasks per node results in a differentiated statement.

As depicted in the logarithmical diagram (cf. figure 8.5) both precedence graph scenarios grow (nearly) exponentially while increasing the number of task. However, for precedence graphs

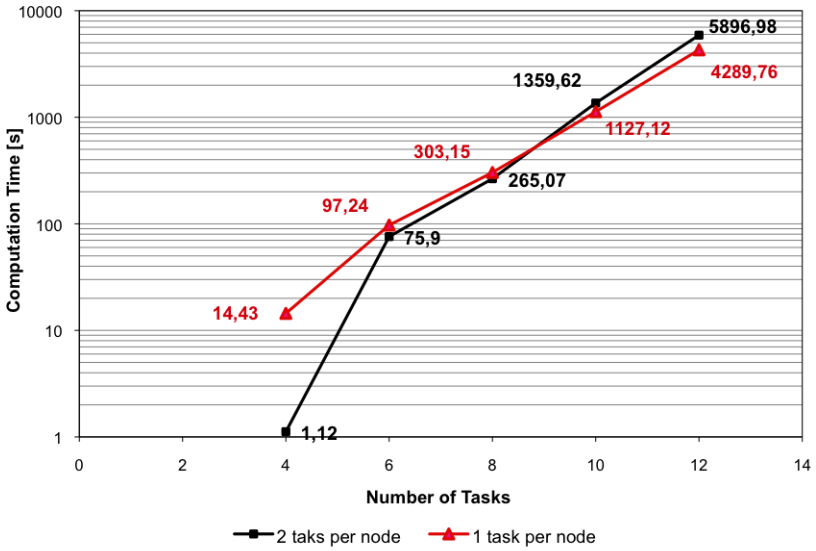
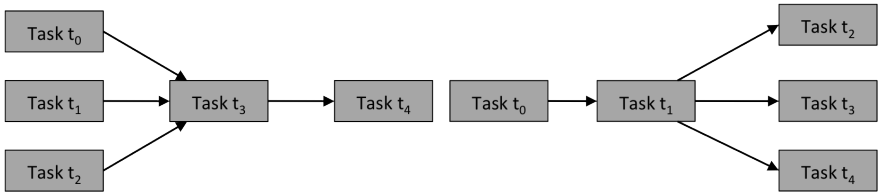


Figure 8.5: Comparing different relation of tasks and nodes: 2 tasks per node vs. 1 task per node

with less than 8 tasks, the task and message scheduling problem is easier to solve for precedence graphs with 2 task per node allocation (black line). In average, the computation time is nearly 22 percent lower for graphs containing 6 tasks than precedence graphs with 1 tasks per node (red line). This trend turns while increasing the number of tasks. This correlates with the expectations that a growing number of potential concurrent situations might occur when having multiple number of tasks trying to allocate the same resource. This leads to an increase in concurrency. For instance, the computation time for an integrated task and message schedule for precedence graphs containing 12 tasks per node demonstrates the expected trend. Measured values show that for graphs using 2 task per node the computation time (the average of 20 generated precedence graphs) is 37 percent bigger (black line) than for precedence graph scenarios with only a single task per node (red line).

8.3.3 Variation in Precedence Graph Layouts

Beside various relations of tasks to nodes, different precedence graph layouts affect the complexity for solving the task and message scheduling problem as well. By using the model checker `sal-smc`, we gradually construct an integrated task and message schedule. Whenever the precedence graph allows for different possible solutions, characterized by concurrent access to a shared resource, we use the model checker's capabilities to explore all specified interleaved possibilities (cf. section 5.1). This, however, impacts the number of transitions called at a certain point in time.



(a) Multiple Starting Nodes Scenario (b) Multiple Ending Nodes Scenario

Figure 8.6: Precedence Graph with multiple starting nodes and multiple ending nodes

Figure 8.6 illustrates two different scenarios. In figure 8.6a the precedence graph consists of multiple starting nodes (task t_0, task t_1 and task t_2). The initial state is specified as a situation in which all tasks are not started and not finished. The goal states are those states where all tasks have finished and all messages have been sent. In case all three starting tasks in figure 8.6a are allocated to the same shared resource (node), the model checker, starting from the initial state executes the *Start Task Transition* three times, following all interleaved scenarios (task t_0 $\succ$ task $t_1\succ$ task t_2 or task $t_2 \succ$ task $t_1 \succ$ task t_0, etc.). Thus, the number of global system states grows exponentially with the number of concurrent situations imposed by the precedence graph. This, however, emphasizes state space explosion. Furthermore, the same effect appears if messages sent by several tasks are trying to allocate the same communication slot. Thus, concurrency, either caused by a shared computing resource or a shared communication resource extends the state space with new branches (complying to different task and message schedules) corresponding to the number of interleaved possibilities.

In the second scenario, depicted in figure 8.6b, interleaved possible schedules may occur by the end of the schedule, assuming that either different messages, sent by task t_1 to task t_2, t_3 and t_4 are trying to allocate the same communication slot, or the tasks (t_2, t_3 and t_4) may try to allocate the same computing resource.

Different precedence graph layouts may therefore cause different level of concurrency while calculating a task and message schedule. Thus, the level of concurrency is mainly based on the precedence graph layout. In the following, we compare different precedence graph layouts in order to illustrate their effect on the computation time needed to calculate an optimal task and message schedule. This, as well, affects scalability of the presented approaches.

Multiple Starting Nodes

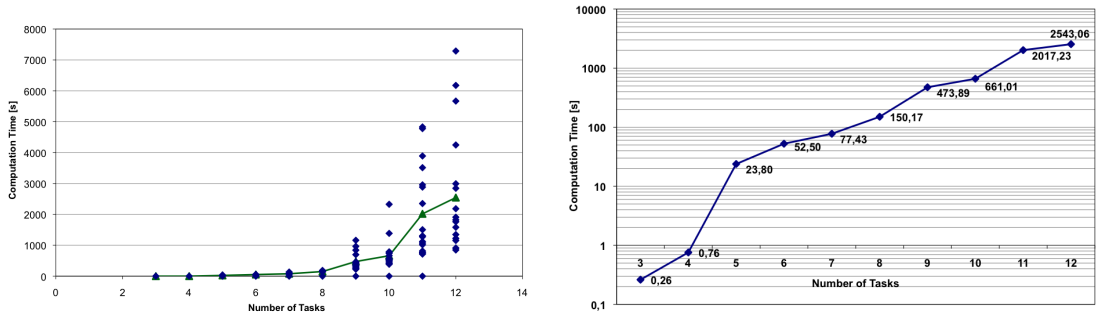
We first investigate multiple starting nodes scenarios, as depicted in figure 8.6a. Experimental results for these precedence graph scenarios are given in the previous section 8.3.1.

Multiple Ending Nodes

In contrast to the multiple starting nodes scenario, we focus on multiple ending nodes scenarios, compare figure 8.6b. Therefore, we generate random precedence graphs with a number of 3 to 12 tasks in a one task per node scenario.

Figure 8.7a illustrates the exponential growth of computation time for an increasing number of tasks per precedence graph. For every considered number of tasks, the average computation time for the corresponding 20 generated precedence graphs is indicated by the green line. Figure 8.7b shows the computation time in a logarithmically diagram. It can be seen, that starting from precedence graphs containing 5 tasks an exponential correlation of computation time can be identified.

The average values (cf. green line) in figure 8.7a depict a bend in the average computation time starting from precedence graphs containing 11 tasks. This effect can be explained by the fact that only 17 of 20 randomly generated precedence graphs following the specified scenario could be calculated. Calculation of the missing precedence graphs can be expected as to complex in terms of computation resources.



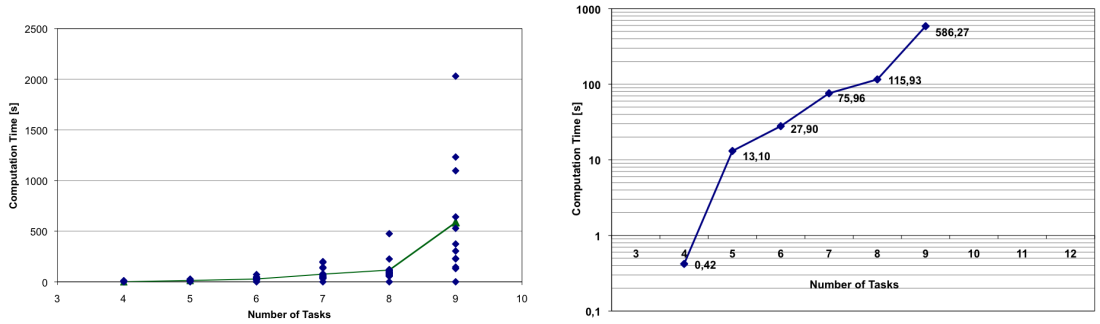
(a) Exponential growth of computation time (b) Logarithmic illustration of exponential growth

Figure 8.7: Precedence graph with multiple ends

Completely randomized precedence graphs

As a third scenario, we specify completely randomized precedence graphs. However, these precedence graphs follow a few simple rules. The number of starting and finishing tasks is not limited. The obtained precedence graph is connected and cycle-free.

Following these rules, we generate 20 complete randomized precedence graphs with a number of 3 to 12 tasks in a one task per node scenario. The following experimental results illustrate the effects of using these complete randomized precedence graphs on the overall computation time (cf. figure 8.8).



(a) Exponential growth of computation time (b) Logarithmic illustration of exponential growth

Figure 8.8: Precedence graph with a random composition

Figure 8.7a depicts the exponential growth of computation time for an increasing number of tasks per precedence graph. Under the given computation resources, schedules for precedence graphs with up to 9 task can be computed. As explained in the previous sections, the level of complexity raises with the level of concurrency in a given precedence graph. Completely randomized precedence graphs may combine these unfavorable properties (e.g. numerous task per node, multiple starting nodes combined with multiple ending nodes) that lead to higher level of complexity.

Figure 8.7b shows the computation time in a logarithmically diagram. It can be seen, that starting from precedence graphs containing 5 tasks an exponential increase of computation is identified. However, even results for precedence graphs containing 9 tasks are vague, because only 10 out of 20 graphs could be calculated at all.

Comparison

In order to compare the different precedence graph layouts, we include all different scenarios in figure 8.9. As expected, precedence graphs with a multiple starting node scenario (blue line) have the highest computation time. Multiple ending nodes scenarios (red line), with one exception at 11 task per precedence graph, have a constantly lower average computation time. This, however, leeds to the assumption that these kinds of precedence graph layouts can be calculated faster. Precedence graphs using completely randomized layouts (green line) are not comparable in this context, because the calculation of schedules was not possible for all graphs due to an increase in complexity.

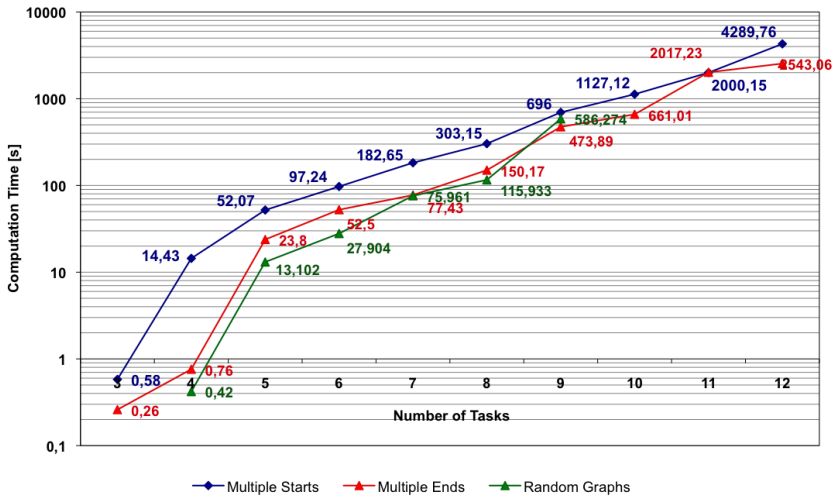


Figure 8.9: Comparison of different precedence graphs layout

8.3.4 Effects of dynamic reordering

This thesis shows that model checking can be used to solve the task and message scheduling problem. However, efficient model checking of problems with huge state spaces is only possible with efficient representation of the model itself. Ordered Binary Decision Diagrams (OBDDs) allow an efficient symbolic representation of the model. As the size of the OBDDs and also the computation time depends on the order of the input variables, dynamic reordering strategies may accelerate the process of computation and increases its efficiency. Dynamic reordering is supported by the SAL framework.

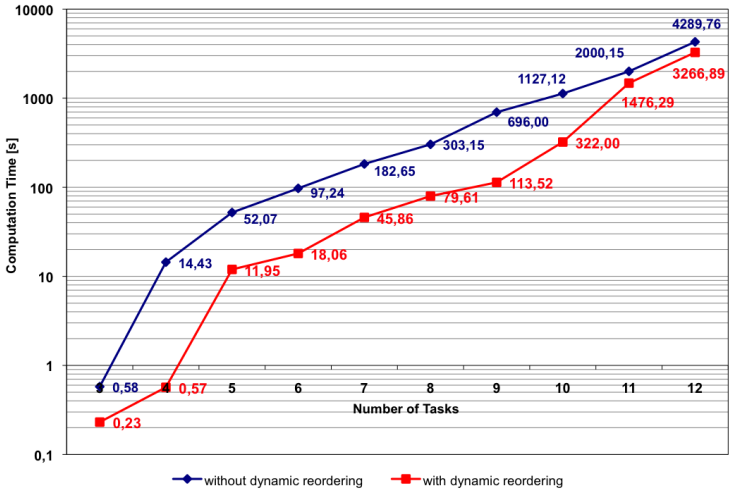


Figure 8.10: Effects of dynamic reordering

Figure 8.10 illustrates the positive effects of dynamic reordering on the computation time. Given exactly the same precedence graphs as an input, we calculated integrated task and message schedules with and without dynamic reordering, in order to highlight the positive effects in terms of average computation time. For instance, an improvement of average computation time of 73,7 percent for precedence graphs containing 8 tasks has been obtained. By increasing the number of tasks this effect reduces to at least 23,9 percent in average for precedence graphs containing up to 12 tasks.

8.3.5 Effects of an additional transition

As demonstrated in previous sections, there are different reasons that increase the complexity of the task and message scheduling problem and thus accelerate the explosion of the state space. Beside various relations of tasks to nodes and different precedence graph layouts, we investigate the effects of an additional transition in this section.

In section 8.3.3 we have shown, that the precedence graph layout has a major impact on the complexity of the integrated task and message scheduling problem when using the presented approaches. This affects the scalability of the presented approaches as well. Increasing complexity is caused by the fact, that concurrent situations (e.g., concurrent use of a computing resource or the communication system) increase the number of global system states because the model checker is used to check all interleaving schedules, namely by the usage of many transitions in parallel.

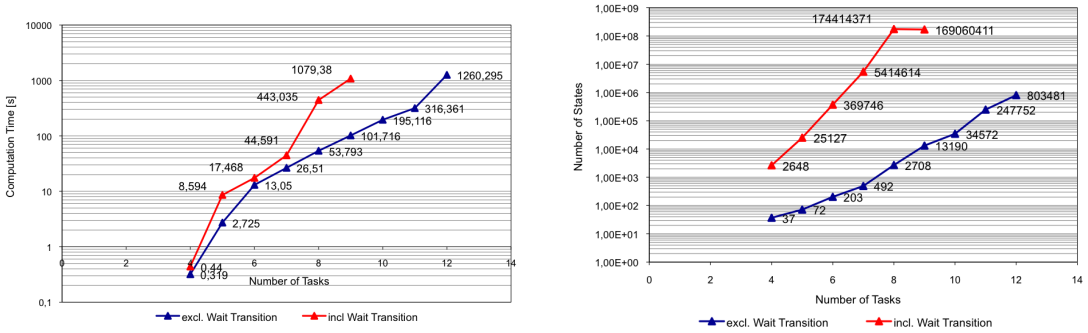
Thus, finding an abstraction for the integrated task and message scheduling problem with minimal number of transitions as possible has a positive effect on our approach in terms of scalability.

In the following, we present how an additional transition affects the computation time for finding an optimal task and message schedule. The `Wait_Message_Transition` allows to skip the next available time slot of the underlying communication bus, although this slot is free. this is rather related to the usage of SAL's bounded model checker. In order to calculate counterexamples with the given length k, we define that a message m_i might not be sent in the actual available slot sl_k, but wait for the next slot sl_{k+1}. This enables to obtain schedules with variable length l. This is already specified in section 5.3.4.7.

First, we randomly generate 20 precedence graphs containing 4 to 12 tasks. The precedence graph layout is a multiple starting nodes - scenario using a single task per node. Figure 8.11 demonstrates the usage with (blue line) and without (black line) the additional `Wait_Message_Transition`.

Figure 8.11a illustrates the computation time in a logarithmical diagram. It can be seen, that starting from precedence graphs containing 6 tasks an exponential increase of the average computation time can be identified. Including the additional `Wait_Message_Transition`, solutions can only be found for precedence graphs containing up to 9 tasks.

However, experimental results for precedence graphs containing 9 tasks are less meaningful, because, as depicted in figure 8.11b, the number of visited states is nearly stagnating. This effect can be explained by the fact, that only 4 of 20 precedence graphs lead to useful results.



(a) Computation Time

(b) Number of states

Figure 8.11: Effects of an additional transition

Thus, the usage of additional transitions emphasizes an increase in complexity and in turn leads to a decrease of scalability using the presented approaches. In the next section we demonstrate how the heuristic approach decreases the state space by selecting the number of transitions.

8.3.6 Weighted Symbolic Task and Message Scheduling

We adopt the principle of symbolic state space exploration to scheduling synthesis, as described in chapter 5. Furthermore, we propose an approach, named *Weighted Symbolic Scheduling*, providing a heuristic that effectively decrease the state space by guiding the exploration and therefore increasing scalability (compare chapter 6). The perfect algorithm would choose the transition that continues a path leading to that desired goal state s_n, namely the optimal task and message schedule. Unfortunately, this cannot be known a priori. Therefore, a mechanism is proposed that increase the efficiency for finding the transition that lead to the desired goal state s_n, while exploring the state space.

Keeping in mind that an additional transition leads to an increase in complexity, as demonstrated in the previous section, we have specified a heuristic approach using a weight - calculation that can be used as a criteria for selecting the next transition. The weight of a transition can be computed off-line, as specified in section 6.2, and is valid for all explorations of the model. Therefore, in each step, the weights of all possible transitions are compared. Transitions leading to states with lower weights are not executed at all, thus saving time and memory.

In order to quantify the effect of the *Weighted Symbolic Scheduling* approach in contrast to the *Symbolic Task and Message Scheduling* approach, we randomly generate 20 precedence graphs containing 4 to 12 tasks each, using a Multiple Starting nodes layout with a single task per

node. In this case the computation duration of a each task is estimated as the same size as a communication slot. For each precedence graph the optimal solution is calculated using both approaches.

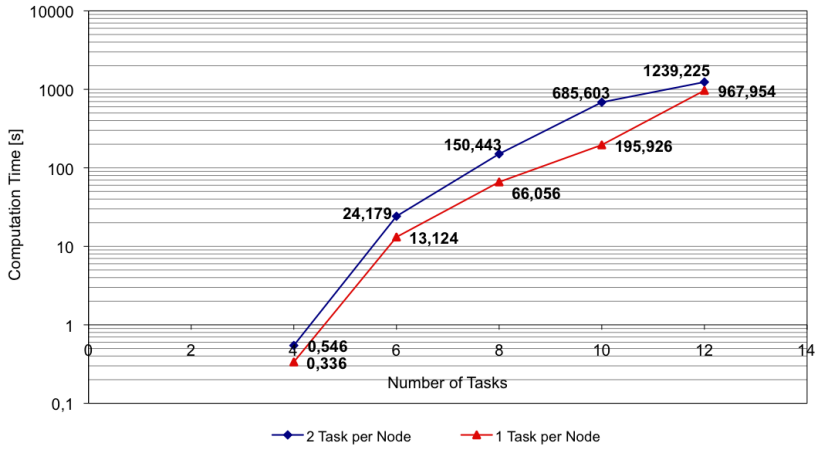


Figure 8.12: Effects on computation time using the heuristic approach

Figure 8.12 compares experimental results using the Symbolic Task and Message Scheduling approach (blue line), as already depicted in section 8.3.1. The average computation time used to calculate an optimal task and message schedule using the Weighted Symbolic Scheduling approach is depicted by the red line. For all precedence graphs an average reduction of computation time can be estimated. For instance, an optimal task and message schedule for precedence graphs containing 10 tasks can be computed (in average) 71,42 percent faster using the presented heuristic approach.

This effect consequently can be seen by the visited number of system states, as depicted in figure 8.13.

After demonstrating positive effects on computation time and number of system states using the *Weighted Symbolic Scheduling* approach, we furthermore demonstrate how this effect even increases, if the complexity given by the precedence graph layout or the relation of tasks to nodes increase. Therefore, we generated a set of precedence graphs using the following settings: We investigate precedence graphs containing 8 tasks in a Multiple Starting nodes scenario with a rising number of tasks per node. Each task is assumed to have a computation duration that equals a communication slot.

As proved by experimental results in section 8.3.2, the complexity of the task and message scheduling problem increases by raising the relation of nodes per task. In order to demonstrate the usability of the developed *Weighted Symoblic Scheduling* approach, especially under conditions of increased complexity, we therefore investigate 20 generated precedence graphs with the following relations of tasks to nodes: one task per node, two tasks per node and four task per node.

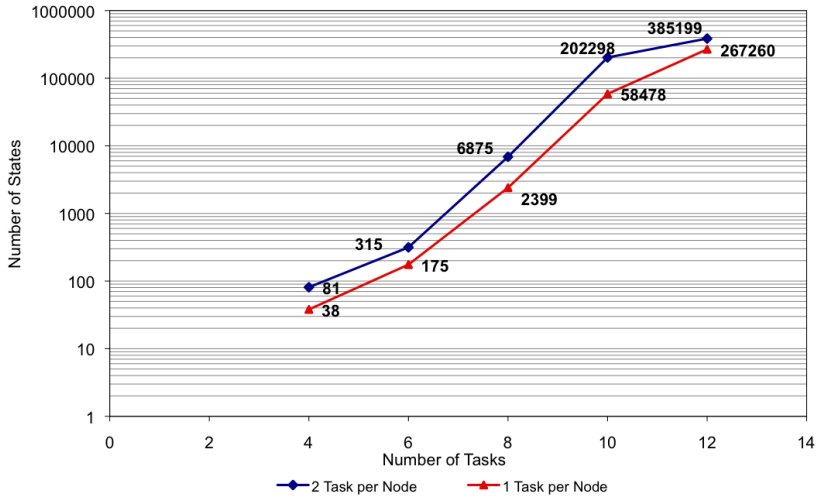


Figure 8.13: Effects on number of states using the heuristic approach

The generated precedence relations between the tasks stays unchanged, just the number of nodes the given set of tasks are allocated to, are reduced.

Figure 8.14 illustrates the increasing benefit of the developed approach.

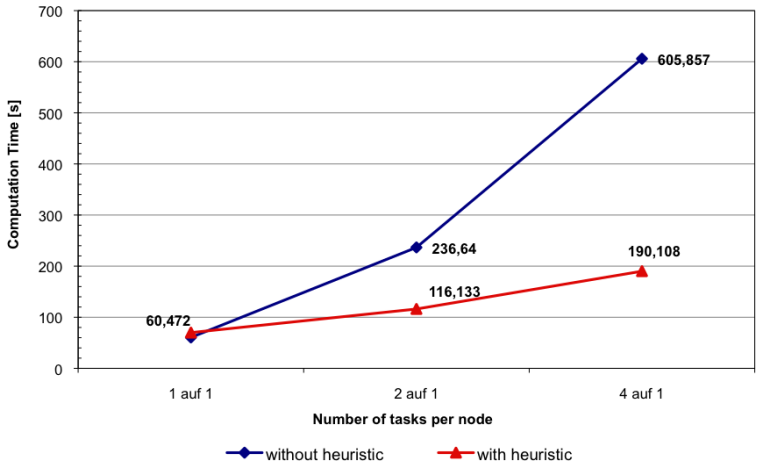


Figure 8.14: Effects of using the heuristic approach (I)

The benefit of using the *Weighted Symbolic Scheduling* approach can be established by the calculation of visited states for the given scenarios. Figure 8.15 highlights the reduction of states needed for finding an optimal task and message schedule, even in concurrent scenarios, where 4 tasks are allocated to one computing resource. Especially in this scenario, a reduction of 92,3 percent in average can be obtained.

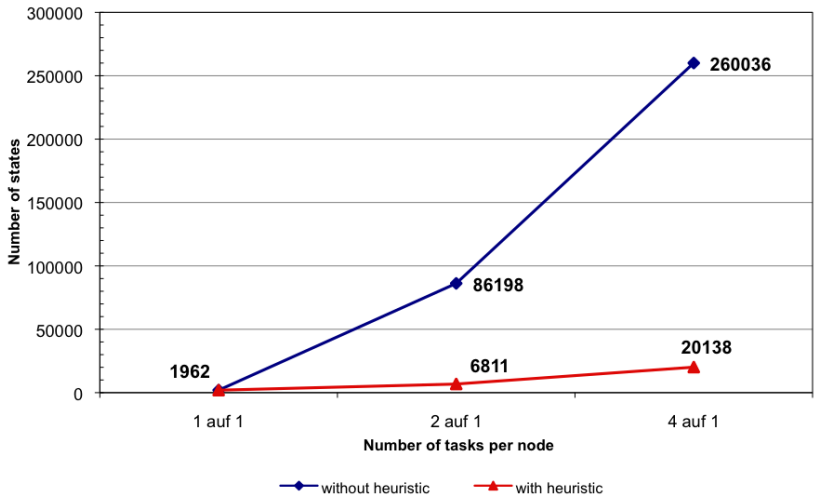


Figure 8.15: Effects of using the heuristic approach (II)

These results prove the benefit of using the *Weighted Symbolic Scheduling* approach for finding an optimal task and message schedule. The *Weighted Symbolic Scheduling* approach is therefore highly eligible for increasing scalability in contrast to the *Symbolic Task and Message Scheduling* approach.

8.4 Discussion of Results

Both presented approaches, namely *Symbolic Task and Message Scheduling* and *Weighted Symbolic Scheduling*, enable to find an optimal solution of the given task and message scheduling problem. However, scalability turns out to be a restricting factor. The previous sections demonstrate that with an increasing number of tasks per precedence graph, the state space and thus the computation time increases (nearly) exponentially. This is caused by an exponentially growing number of possible solutions for finding an integrated task and message schedule, when adding tasks to the precedence graph.

Experimental results demonstrate that the Symbolic Task and Message Scheduling approach using the SAL framework as a model checking tool is able to calculate an optimal task and message schedule up to 13-14 tasks per precedence graph. However, this is not possible for all randomly generated precedence graphs, especially with a concurrent precedence graph layout. Using the heuristic approach proves to be suitable to reduce the complexity of the given task and message scheduling problem and enables to find solution for precedence graphs with up to 16 tasks. Thus, the *Weighted Symbolic Scheduling* approach is eligible for increasing scalability in contrast to the *Symbolic Task and Message Scheduling* approach.

Furthermore, the presented approaches enables to calculated schedules that fulfill various certain system requirements. For instance, the generated schedule should execute a certain task until a certain point in time, because there exist such a system requirement. The developed approach is able to generate a counterexample, namely a task and message schedule, fulfilling this requirement, if there exists such a schedule. Thus, all kinds of requests to an integrated task and message schedule, can be defined and investigated by adapting the optimization criterion.

Chapter 9

Conclusion and Future Work

In this thesis we have developed generation techniques for task and message configurations for time-triggered architectures, that comply with the specific requirements given by IMA architectures (e.g. static off-line schedule, minimal end-to-end latency). The commonality among the given techniques is the problem of integrated task and message scheduling for time-triggered networks. We propose several approaches that allows the automatic generation of task and message configurations. Moreover, the symbolic approaches presented provide optimal system configurations with respect to given system requirements, such as end-to-end latency.

9.1 Accomplishments

Our first approach solves the task and message scheduling problem by regarding it as a graph problem. An off-line scheduling algorithm has been developed for traversing this graph that augments conventional scheduling rules with algorithms addressing the specific problems of scheduling messages on a time-triggered communication medium. The approach integrated task scheduling at system level with message scheduling at communication level. The graph traversal generates a schedule by incremental successor and predecessor calculation in each step. The algorithm incorporates a backtracking and path extension functionality guaranteeing the consistency of the developed schedule. The main advantage is that the algorithm automatically computes schedules and scales up well even for large applications. The price to pay is that the algorithm does not always find an optimal solution. However, the algorithm delivers its result very fast. Even precedence graphs with 100 or 1000 different (concurrent) tasks can be handled in minutes. The algorithm presented is therefore highly scalable and thus applicable for large avionics systems.

Secondly, this thesis extends ongoing research into task and message scheduling based on time-triggered shared resources by first using model checking techniques for solving this kind of problem. We demonstrate that state-of-the-art model checking and bounded model checking

techniques can be used to compute a schedule that fulfills certain system requirements. We introduce *Symbolic Task and Message Scheduling* as a novel approach to this class of problems. This approach allows to automatically compute optimal schedules with respect to minimal end-to-end latency. We adopt the principle of symbolic state space exploration to schedule synthesis and provide a symbolic encoding which guarantees an optimal solution. The symbolic encoding is performed by transferring and defining the task and message scheduling problem as a finite-state model-checking problem $\mathcal{M} \models \varphi$. The major challenge is the scalability of the given approach.

Therefore, this thesis presents - thirdly - a technique focusing on the specific problem of scheduling synthesis to reduce the state space. The developed approach extends the *Symbolic Task and Message Scheduling* approach by introducing a heuristic that decreases the state space by guiding the exploration. This approach is called *Weighted Symbolic Task and Message Scheduling*. This approach increases the scalability of the symbolic task and message scheduling approach by preventing from an exhaustive search through a guided, weight-based exploration.

The developed approaches are implemented in a framework for scheduling synthesis. The framework supports the developed *complexity evaluation approach* by integrating a graph generator as well as a code generator for the symbolic and weighted symbolic approaches. This allows several system specifications to be automatically translated into SAL specification according to the given encoding schemes presented. Furthermore, the framework has several interfaces in order to import configurations (FIBEX- files for FlexRay configurations) and output system configurations (SAL framework for executing and solving the given task and message scheduling problem). The results are transferred back and can be visualized through textual and gnuplot-based graphical elements. Experimental series composed of different precedence graphs containing certain property settings can be computed and compared by this framework.

Finally, experimental results demonstrate that both presented approaches, namely *Symbolic Task and Message Scheduling* and *Weighted Symbolic Scheduling* make it possible to find an optimal solution for the given task and message scheduling problem. Furthermore, it has been possible to demonstrated that *Symbolic Task and Message Scheduling* is able to calculate the optimal task and message schedule with precedence graphs consisting of up to 13-14 tasks. We have shown that the developed heuristic approach effectively reduces the complexity by decreasing the state space and increases the scalability by enabling the calculation of a solution for precedence graphs with up to 16 tasks.

9.2 Perspectives

This thesis provides a basis for further research topics, especially in the developed symbolic and weighted symbolic approaches. As it has been demonstrated in this thesis, formal methods can be used for scheduling synthesis, generating optimal integrated task and message schedules. In this thesis, we focus - triggered by an aeronautic use case and its requirements - on the optimization

criterion of minimizing the system's end-to-end latency, based on a given aeronautic system architecture - meaning a fixed allocation of tasks to nodes.

Further optimization criteria might be investigated in future work - for instance, the generation of schedules with a minimum number of resource constraints, either on the bus or the computing resources. This corresponds to a schedule that is characterized by a minimal amount of concurrent accesses to a shared computing resource or the shared communication medium. A schedule that minimizes the resource constraints would have found an allocation of tasks to nodes which is optimal in correspondence to the given precedence graph. This would require an approach to effectively reallocate certain tasks to other nodes with respect to the collisions found in the schedule calculated by our approach. However, it would be highly interesting to find a system architecture with its corresponding configuration that fulfills such a requirement. The schedule of such a configuration would probably be the schedule with the best end-to-end latency for a given precedence graph.

Another interesting perspective would be to extend the presented approach by the use of different criticality levels. One could image a system configuration comprising of several functions that has different levels of criticality (e.g. safety-critical functions and non safety-critical functions). The heuristic approach could be adapted to prefer tasks or messages that are safety-critical in order to generate a schedule that is optimal with respect to criticality.

Calculating schedules that guarantee quantifiable safety-properties are conceivable as well. The given optimization criterion might be changed according to a given safety- property (e.g. several tasks might meet system-wide timing properties).

A promising line of research deals with different kinds of system requirements, in the sense of multi-period task scenarios - meaning that several tasks may have various periods. In this case a precedence graph needs to be semantically adapted. Our approach assumes, that the given task set, can be transferred to a precedence graph which is used as a basis for the developed approaches. One possibility might be to investigate how task sets composed of tasks with different periods may be mapped in a single precedence graph. This could possibly be performed by exact precedence relations of these multi-period tasks and their periods. It might enable the detection of potential redundant information generated by higher precedence tasks. This would enable the downsizing parts of the given precedence graph to a uni-period task scenario.

List of Figures

2.1 Typical Task Parameters 11
2.2 Simple Example of a Precedence Graph $\mathcal{G}$ 13

3.1 Logical Execution Time (LET) 24

4.1 Simple Precedence Graph $\mathcal{G}$ 28
4.2 Integration of a dummy task t_{dummy} in a precedence graph 29
4.3 Precedence Graph and Graphical Representation of the Algorithm 34

5.1 Different task trying to allocate to the same computing resource 39
5.2 Task trying to allocated a already blocked computing resource 40
5.3 Different messages trying to allocate the same communication resource 40
5.4 Message m_i can be sent in both of the next available slots 41
5.5 Declaration of task, message and node records 49
5.6 Example of possible transition execution order 58
5.7 Calculated Task and Message Schedule for the given Example 1 63

6.1 State Space Reduction Technique 68
6.2 Simplified Architecture of the given Precedence Graph 70
6.3 Declaration of a heuristic records 71

7.1 Single Precedence Graph Generation 76
7.2 Defining the Precedence Relations for each Task 77
7.3 Multiple Precedence Graph Generation 78
7.4 Precedence Graph with Multiple Starting Nodes 79
7.5 Precedence Graph with Multiple Ending Nodes 79
7.6 Precedence Graph with connected randomized nodes 79
7.7 General Parameter for Graph Generation 80
7.8 Precedence Graph Editor 81
7.9 Graphical Representation and Code Generation 82
7.10 Result Generation for Symbolic Task and Message Scheduling 83
7.11 Visual representation of calculated computation time 83
7.12 Visual representation of calculated schedule length 84

8.1 Procedure of Complexity Evaluation 86
8.2 Computation Time for Task and Message Scheduling 89
8.3 Average number of visited states 91
8.4 Computation Time for 2 tasks per node 92
8.5 Comparing different relation of tasks and nodes: 2 tasks per node vs. 1 task per node 93
8.6 Precedence Graph with multiple starting nodes and multiple ending nodes 94
8.7 Precedence graph with multiple ends 95
8.8 Precedence graph with a random composition 96
8.9 Comparison of different precedence graphs layout 97
8.10 Effects of dynamic reordering 97
8.11 Effects of an additional transition 99
8.12 Effects on computation time using the heuristic approach 100
8.13 Effects on number of states using the heuristic approach 101
8.14 Effects of using the heuristic approach (I) 101
8.15 Effects of using the heuristic approach (II) 102

List of Tables

8.1 Design of Experiments 88
8.2 Results of Experiment Series 90

Appendix A

SAL Code Example

We give a short example how the presented framework is used to automatically generate SAL Code, based on the *Symbolic Task and Message Scheduling* approach. The following SAL Code Example comprises to the simple example 1 specified in section 2.3.

```
%****************************************************************
% SYMBOLIC TASK AND MESSAGE SCHEDULING
% Author: Sebastian Voss, EADS Innovation Works
% generated: Dec/11/2009 20:00
%****************************************************************
graph:CONTEXT=

BEGIN
TASKS:TYPE=Task0,Task1,Task2,Task3;
MESSAGES:TYPE=Edge0,Edge1,Edge2,Edge3;
NODES:TYPE=n0,n1,n2,n3;
TIME:TYPE = [0..14];

NATURAL1: TYPE=[0..14];
INDEX1:TYPE = [0..0];
INDEX2:TYPE = [0..1];
INDEX3:TYPE = [0..2];

PREC_MSG:TYPE = ARRAY INDEX1 OF TASKS;
prec_Edge0:PREC_MSG = [[i:INDEX1]Task0];
prec_Edge1:PREC_MSG = [[i:INDEX1]Task0];
prec_Edge2:PREC_MSG = [[i:INDEX1]Task1];
prec_Edge3:PREC_MSG = [[i:INDEX1]Task2];

PREC_TASK1 :TYPE = ARRAY INDEX1 OF MESSAGES;
PREC_TASK2 :TYPE = ARRAY INDEX2 OF MESSAGES;
PREC_TASK3 :TYPE = ARRAY INDEX3 OF MESSAGES;
prec_Task1: PREC_TASK1=[[i:INDEX1]Edge0];
prec_Task2: PREC_TASK1=[[i:INDEX1]Edge1];
prec_Task3: PREC_TASK2 = [[i:INDEX2] IF i=0 THEN Edge3 ELSE Edge2 ENDIF];
```

```
NODETASKS: TYPE = ARRAY INDEX1 OF NODES;
Task0_node:NODETASKS = [[i:INDEX1]n0];
Task1_node:NODETASKS = [[i:INDEX1]n1];
Task2_node:NODETASKS = [[i:INDEX1]n1];
Task3_node:NODETASKS = [[i:INDEX1]n2];

WAITARRAY:TYPE = ARRAY MESSAGES OF BOOLEAN;

taskcalendar:TYPE=
[#
     t_started:BOOLEAN,
     t_finished:BOOLEAN,
     t_start:NATURAL1,
     t_clock:NATURAL1,
     t_comp:NATURAL1
#];
TASKARRAY:TYPE=ARRAY TASKS OF taskcalendar;

msgcalendar:TYPE=
[#
     m_started:BOOLEAN,
     m_set:BOOLEAN,
     m_slot:NATURAL1
#];
MSGARRAY:TYPE=ARRAY MESSAGES OF msgcalendar;

nodecalendar:TYPE=
[#
     node_free:BOOLEAN,
     node_task:TASKS
#];
NODEARRAY:TYPE=ARRAY NODES OF nodecalendar;

%------------------------
%FUNCTIONS
%------------------------
getPrecTask(m:MESSAGES, taskarray:TASKARRAY):BOOLEAN=
     IF (m=Edge0 AND(EXISTS (j:INDEX1):taskarray[prec_Edge0[j]].➥
     t_finished=FALSE)) THEN FALSE
     ELSIF (m=Edge1 AND(EXISTS(j:INDEX1):taskarray[prec_Edge1[j]].➥
     t_finished=FALSE)) THEN FALSE
     ELSIF (m=Edge2 AND(EXISTS(j:INDEX1):taskarray[prec_Edge2[j]].➥
     t_finished=FALSE)) THEN FALSE
     ELSIF (m=Edge3 AND(EXISTS(j:INDEX1):taskarray[prec_Edge3[j]].➥
     t_finished=FALSE))  THEN FALSE
     ELSE TRUE
     ENDIF;
getPrecMsg(t:TASKS, msgarray: MSGARRAY):BOOLEAN=
     IF (t=Task1 AND(EXISTS (j:INDEX1):msgarray[prec_Task1[j]].m_set=➥
     FALSE)) THEN FALSE
     ELSIF (t=Task2 AND(EXISTS (j:INDEX1):msgarray[prec_Task2[j]].m_set=➥
     FALSE)) THEN FALSE
```

```
     ELSIF (t=Task3 AND(EXISTS (j:INDEX2):msgarray[prec_Task3[j]].m_set=➥
     FALSE)) THEN FALSE
     ELSE TRUE
     ENDIF;

getNodeTask(t:TASKS):NODES=
     IF(t=Task0) THEN Task0_node[0]
     ELSIF(t=Task1) THEN Task1_node[0]
     ELSIF(t=Task2) THEN Task2_node[0]
     ELSE Task3_node[0]
     ENDIF;

%-----------------------
%MODULE
%-----------------------
schedule:MODULE =
BEGIN

GLOBAL currenttaskarray:TASKARRAY
GLOBAL currentmessagearray:MSGARRAY
GLOBAL currentnodearray:NODEARRAY
GLOBAL Time:TIME
GLOBAL Bus_free:BOOLEAN

INITIALIZATION
currenttaskarray[Task0].t_comp=2.0;
currenttaskarray[Task1].t_comp=2.0;
currenttaskarray[Task2].t_comp=2.0;
currenttaskarray[Task3].t_comp=2.0;

Bus_free=TRUE;
Time=0;
%Task Calendar Array
(FORALL (i:TASKS): currenttaskarray[i].t_started=FALSE);
(FORALL (i:TASKS): currenttaskarray[i].t_finished=FALSE);
(FORALL (i:TASKS): currenttaskarray[i].t_start=0);
(FORALL (i:TASKS): currenttaskarray[i].t_clock=0);

%Message Calendar Array
(FORALL (j:MESSAGES): currentmessagearray[j].m_started=FALSE);
(FORALL (j:MESSAGES): currentmessagearray[j].m_set=FALSE);
(FORALL (j:MESSAGES): currentmessagearray[j].m_slot=0);

%Node Calendar Array
(FORALL (j:NODES): currentnodearray[j].node_free=TRUE);
(FORALL (j:NODES): currentnodearray[j].node_task=Task1);

%-------------------------------------
%TRANSISTIONS
%-------------------------------------
TRANSITION
```

```
[
  ([](i:TASKS): starttransition:
     currentnodearray[getNodeTask(i)].node_free = TRUE AND
     currenttaskarray[i].t_started = FALSE AND
     getPrecMsg(i,currentmessagearray) = TRUE
     -->
     currenttaskarray'= currenttaskarray WITH [i].t_started:=TRUE ➥
     WITH [i]. t_clock:=0 WITH [i].t_start:=Time;
     currentnodearray'= currentnodearray WITH [getNodeTask(i)]. ➥
     node_free:=FALSE WITH [getNodeTask(i)].node_task:=i)

[]
   ([](i:TASKS): interrupt_change_transition:
     currentnodearray[getNodeTask(i)].node_free = FALSE AND
     currenttaskarray[i].t_started = FALSE AND
     getPrecMsg(i,currentmessagearray) = TRUE
     -->
     currenttaskarray'= currenttaskarray WITH [i].t_started:= TRUE ➥
     WITH [i].t_clock:=0 WITH [i].t_start:=Time
     WITH [currentnodearray[getNodeTask(i)].node_task].t_started:=FALSE
     WITH [currentnodearray[getNodeTask(i)].node_task].t_clock:=0
     WITH [currentnodearray[getNodeTask(i)].node_task].t_start:=0;
     currentnodearray'= currentnodearray WITH [getNodeTask(i)]. ➥
     node_free:=FALSE WITH [getNodeTask(i)].node_task:=i)

[]
   ([](i:TASKS): endtransition:
     currenttaskarray[i].t_started = TRUE AND
     currenttaskarray[i].t_clock = currenttaskarray[i].t_comp AND
     currenttaskarray[i].t_finished = FALSE
     -->
     currenttaskarray'[i].t_finished = TRUE;
     currentnodearray'= (currentnodearray WITH [getNodeTask(i)].➥
     node_free:=TRUE) WITH [getNodeTask(i)].node_task:=Task3)

[]
   runtransition:
   EXISTS (i:TASKS): currenttaskarray[i].t_started = TRUE AND ➥
   currenttaskarray[i].t_clock < currenttaskarray[i].t_comp AND
   (NOT (EXISTS (j:TASKS): j /= i AND
     (currentnodearray[getNodeTask(j)].node_free=TRUE AND
      currenttaskarray[j].t_started = FALSE AND
      getPrecMsg(j,currentmessagearray)=TRUE) )) AND
   (NOT (EXISTS (k:TASKS): k /= i AND
     (currenttaskarray[k].t_comp=currenttaskarray[k].t_clock AND ➥
      currenttaskarray[k].t_finished=FALSE))) AND
   (NOT (EXISTS (l:MESSAGES): (currentmessagearray[l].m_set = FALSE ➥
      AND currentmessagearray[l].m_started = FALSE AND
      Bus_free = TRUE AND
      getPrecTask(l,currenttaskarray)=TRUE ))) OR
```

```
   (EXISTS (m:TASKS): m /= i AND
      currentnodearray[getNodeTask(m)].node_free = FALSE AND
      getNodeTask(m) = getNodeTask(i) AND
      currenttaskarray[m].t_started = FALSE AND
      getPrecMsg(m,currentmessagearray)=TRUE) AND
      currenttaskarray[i].t_started = TRUE AND
      currenttaskarray[i].t_clock < currenttaskarray[i].t_comp AND
   (NOT (EXISTS (j:TASKS): j /= i AND
     (currentnodearray[getNodeTask(j)].node_free=TRUE AND
      currenttaskarray[j].t_started = FALSE AND
      getPrecMsg(j,currentmessagearray)=TRUE) )) AND
   (NOT (EXISTS (k:TASKS): k /= i AND
     (currenttaskarray[k].t_comp=currenttaskarray[k].t_clock AND ➥
      currenttaskarray[k].t_finished=FALSE))) AND
   (NOT (EXISTS (l:MESSAGES): (currentmessagearray[l].m_set = FALSE AND
      currentmessagearray[l].m_started = FALSE AND
      Bus_free = TRUE AND
      getPrecTask(l,currenttaskarray)=TRUE )))
   -->
   currenttaskarray'[Task0].t_clock = IF currenttaskarray[Task0].t_started=➥
   TRUE AND currenttaskarray[Task0].t_clock < currenttaskarray[Task0].t_comp
     THEN currenttaskarray[Task0].t_clock+1
     ELSE currenttaskarray[Task0].t_clock
     ENDIF;
   currenttaskarray'[Task1].t_clock = IF currenttaskarray[Task1].t_started=➥
   TRUE AND currenttaskarray[Task1].t_clock < currenttaskarray[Task1].t_comp
     THEN currenttaskarray[Task1].t_clock+1
     ELSE currenttaskarray[Task1].t_clock
     ENDIF;
   currenttaskarray'[Task2].t_clock = IF currenttaskarray[Task2].t_started=➥
   TRUE AND currenttaskarray[Task2].t_clock < currenttaskarray[Task2].t_comp
     THEN currenttaskarray[Task2].t_clock+1
     ELSE currenttaskarray[Task2].t_clock
     ENDIF;
   currenttaskarray'[Task3].t_clock = IF currenttaskarray[Task3].t_started=➥
   TRUE AND currenttaskarray[Task3].t_clock < currenttaskarray[Task3].t_comp
     THEN currenttaskarray[Task3].t_clock+1
     ELSE currenttaskarray[Task3].t_clock
     ENDIF;
     Time'=Time+1;

[]
   ([](i:MESSAGES): startmessagetransition:
   currentmessagearray[i].m_set = FALSE AND
   currentmessagearray[i].m_started = FALSE AND
   Bus_free = TRUE AND
   getPrecTask(i,currenttaskarray)=TRUE AND
     (NOT (EXISTS (j:TASKS): currentnodearray[getNodeTask(j)].node_free=➥
     TRUE AND
       currenttaskarray[j].t_started = FALSE AND
       getPrecMsg(j,currentmessagearray)=TRUE)) AND
```

```
   (NOT (EXISTS (k:TASKS): currenttaskarray[k].t_started = TRUE AND
     currenttaskarray[k].t_clock = currenttaskarray[k].t_comp AND
     currenttaskarray[k].t_finished = FALSE))
     -->
     currentmessagearray'=currentmessagearray WITH [i].m_started:=TRUE ➥
     WITH [i].m_slot:=Time;
     Bus_free' = FALSE;
     Time'=  IF (NOT (EXISTS (j:TASKS):currenttaskarray[j].t_started= ➥
     TRUE AND currenttaskarray[j].t_finished=FALSE))
       THEN  Time+1
       ELSE Time
       ENDIF;
   )

[]
   ([](i:MESSAGES):
   endmessagetransition:
   currentmessagearray[i].m_set = FALSE AND
   currentmessagearray[i].m_started = TRUE AND
   Time = currentmessagearray[i].m_slot+1
   -->
   currentmessagearray'[i].m_set = TRUE;
   Bus_free' = TRUE
   )

[]
   ELSE -->
]

END;

th1: theorem schedule |- AG(EXISTS (i:TASKS):currenttaskarray[i]. ➥
     t_finished=FALSE);
END
```

Appendix B

Abbreviations

BDD Binary Decision Diagrams

BF Best - Fit

BMC Bounded Model Checker

CAN Controller Area Network

CPU Central Processing Unit

CTL Computational Temporal Logic

DFS Depth - First - Search

DM Deadline - Monotonic

DO - 178B Software Considerations in Airborne Systems and Equipment Certification

DPS Dynamic - Priority Scheduling

EDF Earliest Deadline First

FIBEX Field Bus Exchange Format

FF First - Fit

FPS Fixed-Priority Scheduling

IMA Integrated Modular Avionics

I/O Input / Output

LCM Least Common Multiple

LET Logical Execution Time

LIN Local Interconnect Network

LRM Line Replaceable Modules

LRT Latest Release Time

LRU Line Replaceable Units

OBDD Ordered Binary Decision Diagrams

PTL Propositional Tree Logic

RM Rate - Montonic

SAL Symbolic Analysis Laboratory

SAL-SMC SAL's Symbolic Model Checker

SAL-INF-BMC SAL's Infinite - Bounded Model Checker

SAT Solver Satisfiability Solver

SMC Symbolic Model Checking

TDL Timing Definition Language

TDMA Time Devision Multiple Access

TT Time - Triggered

TTA Time - Triggered Architecture

TTP Time - Triggered Protocol

WMC Witness and Counterexample Model Checker

Bibliography

[AG03] TTTech Computertechnik AG. Time-triggered protocol TTP/C high level specification document. Technical report, TTA Group, 2003.

[And97] H.R. Andersen. An introduction to binary decision diagrams. In *Lecture nodes for Advanced Algorithms*. Department of Information Technology, Technical University of Denmark, 1997.

[ARI93] ARINC. Arinc specification 659: Backplane data bus, December 1993.

[ARI05] ARINC. Arinc664: Aircraft data network, part 7, avionics full duplex switched ethernet (afdx) network, 2005.

[AS99] T. F. Abdelzaher and K. G. Shin. Combined task and message scheduling in distributed real-time systems. *IEEE Trans. Parallel Distrib. Syst.*, 10(11):1179–1191, 1999.

[BCM+92] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 1020 states and beyond. *Inf. Comput.*, 98(2):142–170, 1992.

[BGL+00] S. Bensalem, V. Ganesh, Y. Lakhnech, C. Mu noz, S. Owre, H. Rueß, John Rushby, Vlad Rusu, Hassen Saïdi, N. Shankar, Eli Singerman, and Ashish Tiwari. An overview of sal. In C. M. Holloway, editor, *LFM 2000: Fifth NASA Langley Formal Methods Workshop*. NASA Langley Research Center, 2000.

[BH98] J. Baumgartner and T. Heyman. An overview and application of model reduction techniques in formal verification. In *Performance, Computing and Communications, 1998. IPCCC '98., IEEE International*, pages 165–171, Feb 1998.

[Boe07] S. Boehm. *Reachability Analysis of Fault-Tolerant Protocols*. PhD thesis, University of Duisburg-Essen (Campus Essen), 2007.

[BS05] B. Bouyssounouse and J. Sifakis. *Embedded Systems Design: The ARTIST Roadmap for Research and Development (LNCS)*. Springer-Verlag New York, Inc., 2005.

[But04] G. C. Buttazzo. *Hard Real-time Computing Systems: Predictable Scheduling Algorithms And Applications (Real-Time Systems Series)*. Springer-Verlag Telos, 2004.

[CE82] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs, Workshop*, 1982.

[CLRS01] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Second Edition*. The MIT Press, September 2001.

[Com99] International Electrotechnical Commission. Iec 61158-cpf3 profibus standard. www.profibus.com, 1999.

[Con05] FlexRay Consortium. Flexray communications system protocol specification, version 2.1, revision A. URL http://www.flexray.com, 2005.

[DdM06] B. Dutertre and L. de Moura. Fast linear-arithmetic solver for dpll(t). In *Proc. 18th Computer-Aided Verification conference*, volume 4144 of *LNCS*, pages 81–94. Springer-Verlag, 2006.

[DE98] A. Doboli and P. Eles. Scheduling under control dependencies for heterogeneous architectures. In *International Converence on Computer Design*, 1998.

[Der74] M. L. Dertouzos. Control robotics: The procedural control of physical processes. In *IFIP Congress*, pages 807–813, 1974.

[dMOR+04] L. de Moura, S. Owre, H. Rueß, J. Rushby, N. Shankar, M. Sorea, and A. Tiwari. Tool presentation: Sal 2. In *Computer-Aided Verification (CAV 2004)*. Springer-Verlag, 2004.

[dMOS03] L. de Moura, S. Owre, and N. Shankar. The sal language manual. Technical Report SRI CSL 01-02 (Revision2), SRI International, August 2003.

[dMRS03] L. de Moura, H. Rueß, and M. Sorea. Bounded model checking and induction: From refutation to verification. In *CAV 2003*, volume 2725 of *LNCS*, pages 14–26. Springer-Verlag, 2003.

[EKP+98] P. Eles, K. Kuchcinski, Z. Peng, A. Doboli, and P. Pop. Prochess scheduling for performance estimation and sythesis of hardware/software systems. In *Proceedings of the 24th Euromicro*, 1998.

[EMCGP99] Jr. E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, Cambridge, MA, USA, 1999.

[Far06] E. Farcas. *Scheduling Multi-Mode Real-Time Distributed Components*. PhD thesis, University of Salzburg, 2006.

[FDD89] Fibre distributed data interface (fddi) - part 2: Token ring media access conctrol (mac). ISO INterational Standard 9314-2, 1989.

[FFPT05] E. Farcas, C. Farcas, W. Pree, and J. Templ. Transparent distribution of real-time components based on logical execution time. *SIGPLAN Not.*, 40(7):31–39, 2005.

[FH03] M. Fränzle and C. Herde. Efficient sat engines for concise logics: Accelerating proof search for zero-one linear constraint systems. In *Logic for Programming, Artificial Intelligence, and Reasoning*, 2003.

[fS98] International Organization for Standardization. Iso 11898: Road vehicles: Controller area network (can), 1998.

[fSoAS04] Association for Standardisation of Automation and Measuring Systems. Mcd-2[fbx] - fibex - field bus exchange. www.asam.net, 2004.

[GCG+99] C. Grumberg, E. M. Clarke, O. Grumberg, M. Minea, and D. Peled. State space reduction using partial order techniques. In *Software Tools for Technology Transfer*, pages 279–287. Springer-Verlag, 1999.

[God91] P. Godefroid. Using partial orders to improve automatic verification methods. In *CAV '90: Proceedings of the 2nd International Workshop on Computer Aided Verification*, pages 176–185, London, UK, 1991. Springer-Verlag.

[HD93] K. Hoyme and K. Driscoll. Safebus (tm). IEEE Aerospace and Electronics Systems Magazine, March 1993.

[HHK01a] T. Henzinger, B. Horowitz, and C.M. Kirsch. Embedded control systems development with giotto. In *Proceedings of the International Conference on Languages*. ACM Press, 2001.

[HHK01b] T. Henzinger, B. Horowitz, and C.M. Kirsch. *Giotto: A time-triggered language for embedded programming*. Springer-Verlag, 2001.

[HHM+] T. Henzinger, B. Horowitz, S. Matic, C.M. Kirsch, M.A. Sanvido, and A. Ghosal. The giotto project. http://www-cad.eecs.berkeley.edu/ fresco/giotto/.

[JLL04] A. R. Jensen, L. B. Lauritzen, and O. Laursen. Optimal task graph scheduling with binary decision diagrams, 2004.

[KB01] H. Kopetz and G. Bauer. The time-triggered architecture. *Proceedings of the IEEE, Special Issue on Modeling and Design of Embedded Software*, October 2001.

[L'E10] P. L'Ecuyer. Uniform random number generation. *International Encyclopedia of Statistical Science*, 2010.

[Liu00] J. W. S. Liu. *Real-Time Systems*. Prentice Hall PTR, 2000.

[LK03] Motorola LIN Konsortium. Lin specification package. Revision 2.0, 2003.

[LL73] C.L. Liu and J.W. Layland. Scheduling alogrithms for multiprogramming in hard real-time enviroments, Januar 1973.

[LSCK01] P. L'Ecuyer, R. Simard, E. J. Chen, and W. D. Kelton. An object-oriented random-number package with many long streams and substreams. *Operations Research*, 50:1073–1075, 2001.

[LW82] J. Y.-T. Leung and J. Whitehead. On the complexity of fixed-priority scheduling of periodic real-time tasks. In *Performance Evaluation*, 1982.

[MD78] A. K. Mok and M. L. Dertouzos. Multiprocessor scheduling in a hard real-time environment. In *7th Texas Conference of Computing Systems*, 1978.

[Mer01] S. Merz. Model checking: A tutorial overview. In *MOVEP '00: Proceedings of the 4th Summer School on Modeling and Verification of Parallel Processes*, pages 3–38. Springer-Verlag, 2001.

[MFHS05] A. Metzner, M. Fränzle, C. Herde, and I. Stierand. Scheduling distributed real-time systems by satisfiability checking. In *RTCSA '05*, pages 409–415, Washington, DC, USA, 2005. IEEE Computer Society.

[Mok83] A. K. Mok. *Fundamental Design Problems of Distributed Systems for Hard Real-Time Enviroments*. PhD thesis, Laboratory for Computer Science (MIT), 1983.

[Pau02] M. Paulitsch. *Fault-Tolerant Clock Synchronization for Embedded Distributed Multi-Cluster Systems*. Doctoral thesis, Institut für Technische Informatik, Technische Universität Wien, Treitlstr. 1-3/3/182-1, Vienna, Austria, 2002. Available at http://www.vmars.tuwien.ac.at.

[PEP99] P. Pop, P. Eles, and Z. Peng. An improved scheduling technique for time-triggered embedded systems. In *EUROMICRO Conference Proceedings*, number 25, pages 303–310. ACM, 1999.

[PH08] M. Paulitsch and B. Hall. Flexray in aerospace and safety-sensitive systems. *IEEE Aerospace and Electronic Systems Magazine*, 23:4–13, 2008.

[PRS08] M. Paulitsch, H. Rueß, and M. Sorea. Non-functional avionics requirements. In *Leveraging Applications of Formal Methods, Verification and Validation, ISoLA 2008*, October 2008.

[PT08] W. Pree and J. Templ. Forget about flexray. Technical report, preeTEC.com - time is on our side, 2008.

[Ran92] E. B. Randal. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Comput. Surv.*, 24(3):293–318, 1992.

[RTCfA] Inc. Radio Technical Commision for Aeronautics. *DO - 178B*. RTCA.

[Ruh05] J. Ruh. *Entwurf von fehlertoleranten Steuergeräteapplikationen in Kraftfahrzeugen unter Berücksichtigung moderner Entwicklungsmethodiken*. PhD thesis, University of Stuttgart, 2005.

[SJ08] R. Shaw and B. Jackman. An introduction to flexray as an industrial network. *IEEE Iternational Symposium on Industrial Electronics*, pages 1849–1854, 2008.

[SL78] S.K.Dhall and C.L. Liu. On a real-time scheduling problem. *Operations Research*, 1978.

[SS03] N. Shankar and M. Sorea. Counterexample-driven model checking. Technical Report SRI-CSL-03-04, SRI International, 2003.

[TBW94] K.W. Tindell, A. Burns, and A. Wellings. Analysis of hard real-time communications. In *Real-Time Systems*, 1994.

[TC94] K. Tindell and J. Clark. Holistic schedulability analysis for distributed hard real-time systems. *Microprocessing and Microprogramming - Euromicro Journal (Special Issue on Parallel Embedded Real-Time Systems)*, pages 117–134, 1994.

[TV99] E. Tovar and F. Vasques. From task scheduling in single processor environments to message scheduling in a profibus. *IPPS/SPDP Workshops*, pages 339–252, 1999.

[VSE09] S. Voss, M. Sorea, and K. Echtle. Sal-based symbolic scheduling in time-triggered networks. In *IFM '09: Proceedings of the 7th International Conference on Integrated Formal Methods*, pages 200–214. Springer-Verlag, 2009.