A Computational Architecture for Real-Time Systems

Sias Mostert

Dissertation presented for the
Degree of Doctor of Philosophy at the
University of Stellenbosch.

Promoter: Prof JJ du Plessis
Joint study leader: Prof WA Halang

December 2000

"*Declaration*

I, the undersigned, hereby declare that the work contained in this dissertation is my own original work and that I have not previously in its entirety or in part submitted it at any university for a degree.

Signature:..

Date:.

Opsomming

Die skep van betroubare intydsestelsels vir missie kritiese toepassings is 'n proses wat baie hulpbronne verg en waarin maklik 'n fout gemaak kan word. Om 'n betroubare stelsel te skep vereis 'n konsensus oor die korrektheid van 'n stelsel, wat bereik word wanneer die eienskappe van die stelsel goed verstaan word. Dit vereis op sy beurt weer 'n konsensus oor 'n goed gedefinieerde berekenings argitektuur. Daar is tans geen enkel ooreengekome berekenings argitektuur op die toepassingsvlak wat kan dien as 'n gemeenskaplike voertuig vir die ontwerp en implementering van intydsestelsels nie. Dit is die hipotese van die proefskrif dat 'n berekenings argitektuur met 'n streng basis, wat toegepas kan word vanaf ontwerp tot implementering, ingenieurs in staat sal stel om intydsestelsels beter te kan verstaan. Die hipotese word ondersoek deur die intydse datavloei argitektuur, RDF, te gebruik om 'n stelsel vanaf ontwerp tot implementering te beskryf. Daar is vier spesifieke navorsings areas ter verbetering van die ingenieurswese proses vir intydsestelsels, wat in die proefskrif aangespreek word: 1) die ontwikkelling van 'n argitektuur vir intydsestelsels wat geskik is vir die ontwerp en implementering in programmatuur en apparatuur, 2) die konsolidering van 'n aantal grafiese tale in 'n grafiese notasie vir die funksionele spesifikasie, ontwerp en implementering van intydsestelsels, 3) die ontwikkelling van 'n eenvoudige verwerker argitektuur vir die uitvoering van intydse toepassings en 4) die evaluering van die argitektuur in die konteks van 'n mikrosatelliet gevallestudie. Die volgende oorspronklike bydraes word gemaak: 1) die sneller voorwaardes vir datavloei stelsels word uitgebrei met 'n disjunktiewe patroon saam met die tradisionele konjunktiewe patroon, 2) die inherente intydse datavloei eienskap, n.l. dat 'n taak wat boodskappe ontvang, alle verwerking wat met 'n vorige boodskap gepaard gegaan het moet afhandel, voordat 'n volgende boodskap ontvang word, word uitgebrei na die sinkrone datavloei model, 3) 'n notasie om al die eienskappe van 'n intydsestelsel te beskryf word gedefinieer met RDF as die basis taal, 4) twee apparatuur verwerker argitekture word beskryf wat 'n een-tot-een kartering aanbied tussen die ontwerp en die implementering, en wat gevolglik die semantiese gaping verklein tussen ontwerpstaal en die uitvoeringsargitektuur en 5) die klasse van stelsels wat gemodelleer kan word met RDF sluit beheerstelsels en datavloeistelsels in. Die grafiese notasie en verwerker argitektuur was toegepas op sekere aspekte van die SUNSAT mikrosatelliet projek.

Abstract

The engineering of dependable real-time systems for mission critical applications is a resource intensive and error prone process. Achieving dependability requires a general consensus on the correctness of a system with regard to its intended function. For a consensus to be achieved, the properties of the system must be well understood which, in turn, requires consensus on a rigorously defined computational architecture. There is currently no single agreed upon computational architecture at the application level which can serve as a common denominator for the design and implementation of real-time systems. It is the thesis of this dissertation that a rigorous computational architecture, applicable from design to implementation, enables engineers to better understand software for real-time systems. To substantiate this claim, the real-time data flow architecture RDF with its notation allowing the description of complete systems from design to implementation will be explored. Four distinct research areas for improving the engineering process of real-time systems are dealt with in the dissertation: 1) the development of an architecture for real-time systems being suitable for design and implementation in software and hardware, 2) the consolidation of a number of graphical languages into a graphical notation for functional specification, design and construction of real-time systems, 3) the development of a simple processor architecture for the execution of real-time applications, and 4) and the evaluation of the architecture in the framework of a microsatellite case study. In particular, the following original contributions are made: 1) the firing semantics of data flow systems are expanded to include disjunctive firing semantics in a novel way in addition to the classical conjunctive firing semantics, 2) the inherent real-time data flow property, i.e. that a receiving task must be ready to receive the next incoming message when it is sent, is extended to the synchronous data flow model, 3) a notation for describing all properties of real-time systems is defined with the real-time data flow language RDF as base language, 4) two hardware processor architectures are introduced that offer one-to-one correspondence between design and implementation and, thus, reduce the semantic gap between design language and program execution, and 5) the class of systems that can be modelled with data flow architectures is shown to include control systems and data flow systems. The language set and processor architecture were applied to certain aspects of the SUNSAT microsatellite project.

Acknowledgements

This dissertation is a result of a process which has been influenced by many people and circumstances. I am indebted to a number of people and, on the completion of this dissertation, I would like to thank them.

The past years of collaboration on the SUNSAT project have provided opportunities untold to experience development of a computer system for a mission critical application with a large team of people. The following people, in particular, have contributed in some way to the developmnet software since 1992: Rein Brune, Herman Gouws, Johan Boot, Johan du Buisson, Chris Tribelhorn, Rikus Grobler, Hans van der Merwe, Niki Steenkamp, Hans Grobler, Ben van der Merwe, Tielman Botha, Johan Uys, Gerhard le Roux, Herman Steyn, Pierre Oosthuisen, Laurentius Joubert and Marius Prinsloo.

My gratitude to my supervisor, Prof Jan J. du Plessis, who was always ready to share an insight into the bigger picture. Without the opportunities that he provided, this dissertation would not have been possible. His encouragement has kept me pursuing my research.

The insight and perseverance of Prof Wolfgang A. Halang is sincerely appreciated, as he was the one who introduced me to the world of real-time system engineering. Thank you to Prof Pieter Bakkes for introducing me to the research area of data flow architectures. The opportunity to spend time with Prof Rudy Lauwereins at the Catholic University of Leuven is appreciated, and a thank you goes to Nathalie Cossement for the constructive talks on modelling.

The encouragement and patience of my parents, my wife Belinda, my son Werner and daughter Ané need special mention. This dissertation is dedicated to them.

Praise God, my Heavenly Father, for the courage to undertake and the will to persevere.

Contents

List of Figures

Glossary

ADCS Attitude Determination and Control System

APRS Automatic Position and Reporting System — short message protocol standard

ASIC Application Specific Integrated Circuit

AtoD Analog to Digital converter

AX.25 Data link layer packet communication standard

BDF Binary Data Flow

C&C Command and Control — a software component that receives commands and controls a system

CSDF Cyclo-Static Data Flow

DDF Dynamic Data Flow

DMA Direct Memory Access — a mechanism whereby data are transferred between a source and a destination without the intervention of a general purpose processor

DtoA Digital to Analog converter

EDF Earliest Deadline First — a scheduling algorithm for single processors

EDAC Error Detection and Correction — an algorithm that can detect single or multiple bit errors and correct the errors

EEPROM Electrical Erasable Programmable Read Only Memory

FLASH memory Fast Electrical Erasable Programmable Read Only Memory

FPGA Field Programmable Gate Array — a programmable logic device

FSM Finite State Machine

GPP General Purpose Processor — a traditional microprocessor with input/output functionality and a central processing unit

HIL Hardware In the Loop

ICP Interface Control Processor — an 8031 microcontroller on the ADCS system responsible for many of the input/output tasks

LEO Low Earth Orbit — and orbit between 300km and 1000km altitude from the earth geoid.

MTG Multi-Thread Graph

OBC1 On board computer 1 — an 80C188EC processor controlling normal operation of SUNSAT

OBC2 On board computer 2 — an 80C386EX processor capable of controlling normal operation of SUNSAT; currently used as back up

Pacsat Packet satellite communication — a communication service planned for SUNSAT

PBBS Public Bulletin Board Service — a communication service planned for SUNSAT

PCI Professional Computer Interface — a computer interface standard available on most PC compatible computers

PIC Programmable Interrupt Controller — a device integrated with the 80C386EX processor and available as a separate package; it prioritises and manages interrupts

PROM Programmable Read Only Memory

QoS Quality of Service

RAM Random Access read/write Memory

RD ReaD — an operation normally performed by a processor to load a word into the processor's internal registers

RF Radio Frequency — refers to wireless communication

RFP Request for Proposal

RDF Real-time Data-Flow

RTDF paradigm Real-time Data-Flow paradigm — the paradigm where each task must consume its input tokens before the next input tokens are available

RTX Real-Time Executive — the name for the kernel that implements the RDF computational architecture

SatFTP Satellite File Transfer Protocol — a communication protocol for file transfer

SCC Serial Communication Controller — a device converting serial bit streams into parallel bit streams and vice versa

SCF State machine Controlled Flow diagrams [105]

SDF Static Data Flow — same as synchronous data flow

SIOP Simple Input Output Processor — a simple processor dedicated to input and/or output operations

SOC System-On-a-Chip — technology where subsystems made in silicon are integrated onto one substrate or into one package

Sproc Simple task processor — a simple processor dedicated to execute one or more tasks based on a trigger within the RTDF paradigm and EDF scheduling algorithm

SSTP SUNSAT Satellite Time Protocol — a protocol used for time synchronisation on SUNSAT

SUNSAT Stellenbosch UNiversity SATellite — a 64 kg microsatellite launched on 23 February 2000

Transputer A microprocessor optimised for parallel computation

UML Unified Modelling Language

UART Universal Asyncrhonous Receiver/Transmitter

WR WRite — an operation normally performed by a processor to write a word into external memory

Chapter 1

Introduction

1.1 Statement of the Problem Considered

The engineering of dependable real-time systems for mission critical applications is a resource intensive and error prone process. The criteria to assess improvements are reduced development time, lower system cost and enhanced dependability.

Real-time systems with high dependability requirements are found in safety and mission critical applications. A failure in these types of systems can lead to loss of life or very high cost of replacement. Consider the Ariane 5 launch failure that was pinpointed to the re-use of a software component for the Ariane 4 launch vehicle. The component was used in the Ariane 5 software environment without a rigorous testing cycle of the software to save costs by renouncing the testing phase. The resulting loss was estimated at $\$500,000,000$.

Achieving dependability necessitates certification of the software. This requires a general consensus on the correctness of a system with regard to its intended function. This general consensus can only be achieved by a group of people agreeing on the function of the software. In extreme cases, this must be done in a court of law where the actors are not experts in the field of computer systems. For this consensus to take place, the properties of the system must be understood and argued about. This requires consensus on the computational architecture, a notation to describe the computations in this architecture, and rigorous semantics of actions in the computational architecture.

There is currently no single agreed upon computational architecture at the application level which can serve as a common denominator for the design and implementation of a real-time system. The available computational architectures are embedded in hardware processors and at a higher level in compilers. The highest level of computational architecture is embedded in operating systems, which are designed for the support of a multitude of functionality, rather than the minimum necessary dependable functionality.

The lack of a common computational architecture means that there is no commonly agreed notation for describing a real-time system, except for the machine code that executes on a processor or the C, C++ or assembly language source code for a system. The implied architecture of the C, C++ and assembly languages are clearly that of the hardware processor and not of the application.

A notation or language is not an end in itself, it rather provides a way to communicate about the properties of a system of interest. The communication is among people and between people and a computer system that implements the intent described in the notation. It is recognised that compilers and processors best understand the current notations used for communication. A notation requires a rigorous structure and semantics so that one representation of a language means the same to two different people, without any further communication. Indeed the computer accepts the source code input as is, as there are no other mechanisms available for communication.

The systems considered in the sequel exhibit the following properties:

1. The systems have a real-time component that requires predictable timing behaviour.
2. The systems must be dependable which requires rigorous validation of the final implementation against the specifications.
3. The systems must be cost effectively engineered which leads to shorter development periods.
4. The systems interact with environments that exhibit parallel processing behaviour.

In the related field of embedded software for consumer electronics, recent work on component models for software [110] identifies the following problems:

1. Size and complexity of the software are increasing at a rate approximately given by Moore's law, i.e. doubling in size every two years.
2. Development time must decrease rapidly.

Thus, methods found in either the dependable real-time domain or the embedded consumer electronics domain should also be applicable to the other domain.

1.2 Viability of the Problem

Why does creating a real-time system on which we want to depend consume so many resources? A part of the answer can be found by considering the activities in a software engineering development process.

The partial refinement development process consists of the following steps. It is derived from the object oriented analysis and design process described by Booch in [7].

1. Establish core requirements (conceptualisation)
2. Develop a model of the desired behaviour (functional specification and analysis)
3. Create an architecture for the evolving implementation (design)
4. Evolve the implementation through partial refinement
   (a) Identify classes and objects

(b) Identify semantics of classes and objects

(c) Identify relationships between classes and objects

(d) Specify interfaces and implementation of classes and objects

(e) Refine a number of classes and objects suitable for implementation

(f) Go back to identifying classes and objects

5. Manage postdelivery evolution (maintenance)

One could consider the first two steps as a *specification* activity. The third step as a *design* activity, the fourth one *programming* and the last major step is *maintenance.*

Considering the software development process in the previous paragraph, one can identify a number of transitions in computational architecture during the process:

- Step (1) to (2): from requirements to model of desired behaviour
- Step (2) to (3): from model of desired behaviour to creating an architecture for the evolving implementation
- Step (4a) to (4f): in the partial refinement process where the components are identified, a multitude of computational architectures is possible; in fact, there could be a different computational architecture for each object identified
- Step (4d) to (4e): the final architectural translation is typically required when a system is translated into assembly language and then into binary machine code

Lee [64] recognises the same problem of identifying computational architectures that are closer to the application domain in embedded system development:

> "The key challenge in embedded software research is to invent frameworks with properties that better match the application domain. One of the requirements is that time be re-introduced."

The lack of single computational architecture and an associated notation means that

1. design engineers cannot communicate with each other across natural language boundaries,
2. there is no commonly accepted transformation from description language to implementation, and
3. temporal properties cannot be specified and, hence, cannot be checked.

It is further interesting to consider the observation of Lee [64] that what is meant by **software** is primarily sequential execution, with the same hardware resources being multiplexed in time to perform a variety of functions. That is, there is a single instruction stream. What is meant by **hardware** is primarily parallel execution, with hardware resources not being shared among functions (or at least, not as much).

The problem of architectural transformations can, thus, be traced back to a mismatch between the computational model of (sequential) implementation and the (parallel) real world with which the interaction is consummated. This mismatch in computational models consumes enormous amounts of resources to create dependable real-time sequentially operating systems to interact with a parallel computational world.

1.3 State of the Art

How do other systems address the mismatch in computational architectures? There is work to improve architectures in order to better match the application domain. This typically departs from a conventional processor point of view and extends the computations possible with a particular compiler or programming language through libraries [9, 21]. The creation of libraries for conventional processors has a number of implications:

1. Increasing reliance is placed on software architecture translators.
2. Validation based on reverse backward translation is made almost impossible.
3. Simple operations such as system debugging are carried out in a different architecture than the design architecture.

Efforts in signal processing have focussed on signal flow languages and data flow processors. These have evolved from homogeneous data flow systems [62] to cyclo-static ones [6] to control dominated data flow systems [104, 106, 15, 28].

Recognising that a single language notation cannot describe the complexity of a computer based system, the Unified Modelling Language (UML) standard defines 9 views and 7 languages with which intents can be expressed [22]. Statemate [37, 36] allows to describe a system with 3 views to try to achieve a complete description of the system.

The current status of adding a real-time profile to UML [93] is found in a response to the "Real-Time Profile for UML RFP" dated 14 August 2000. The response from the Object Management Group includes:

"Predictability plays a key role in the submission, because problems detected early on during the development life cycle can be removed at a much lower cost, and with substantially less re-work, making the ability to model the key elements of schedulability analysis, performance analysis, and queuing theory a natural part of the submission. Especially the ability to model rate monotonic analysis has been scrutinised as part of this work." The definition of real-time systems (hard, soft, time-driven, event-driven and so on) for UML [20] is so broad that it does not address the problem of working within an architecture with commonly agreed modelling and implementation semantics. It is expected that the architecture proposed in this dissertation could largely be modelled in real-time UML, once the real-time UML extensions will be complete.

State Machine Controlled Flow Diagrams (SCF) [105] consist of functional components (data flow components) controlled by associated Finite State Machines (FSM). It is claimed that they can represent cyclo-static data flow, marked graphs, communicating state machines, as well as Petri Nets. This could make them suitable for a lower-level computational architectural representation of any of the other computational architectures. It is questionable whether the

availability of the wide variety of computational models shortens the development cycle or increases the audience that can understand and reason about a system. SCF diagrams support hierarchy through components, refinement and abstraction. A component can be replaced by another component that is refined (design purposes) or abstracted (specification purposes). Parallelism and non-determinism are also supported.

Data flow machines are good at modelling concurrent behaviour, and finite state machines are good at modelling reactive control. What is missing though, is an explicit notion of time so that systems realised can be checked for temporal correctness. Recent developments in data flow systems combine control flow concepts in the data flow graph. In the case of the DF* [15] (Data Flow with STAte Machine controlled Reconfiguration) and Funstate [105] (State Machine Controlled Flow Diagrams) modelling languages, the execution of the data flow flow part of each node is controlled by a finite state machine. The state machine representation provides additional information such as which code segment is executed on which incoming message.

Improving embedded software for consumer electronics by lower cost development and higher dependability is based on the use and re-use of software components that work within an explicit software architecture. The Koala Component Model described in [110] is such an environment. Applicable at the system level only, it does not explicitly take into account the notion of time, and ignores the progress offered in software re-use by object orientation.

1.4 Addressing the Problem

The thesis of this dissertation is that a rigorous computational architecture will enable engineers to better understand and manage software for real-time systems. This should lead to shorter development cycles, and enable larger software systems to be validated due to a better understanding of them. An adequate computational architecture should be characterised as follows.

1. A computational architecture includes a notation for describing systems that compute in this architecture.
2. This notation allows one to capture all properties of interest in a system in any of the development phases, in particular the time property.
3. The architecture must by close enough to the application domain to allow domain specialists to create systems within the architecture.
4. An implementation architecture is available matching the computational architecture in such a way that no architecture translation steps are required to realise systems, viz. a processor architecture which allows a one-to-one mapping of designed components to hardware.
5. The architecture enables generic components to be developed ready for re-use.

The question arises whether such a computational architecture, notation and implementation media are rich enough to be used for real systems. To explore this question, the use of an architecture and notation is shown for the design, analysis and implementation of a microsatellite software system. See Chapter 5 for a number of case studies where the RDF architecture is used to model a microsatellite system.

An analogy for a computational architecture with a direct equivalent implementation, are the schematic circuit diagrams of electronic circuits that processes potential differences leading to current flows. Each of the symbols in a circuit diagram has a model of some transfer function on the incoming signal (voltage or current). Each of the components in an electric circuit has the same transfer function on the incoming signal.

Another analogy can be found in the digital hardware domain where a higher level of abstraction is achieved. The voltage levels between interconnections are agreed upon, and the currents flowing as results of voltage differences are only functions of how many outputs are driven by an input. More importantly, meaning is defined as the variations in the voltage level of a particular wire, also called a signal. A signal is defined as a line from an output to one or more inputs, and the time behaviour of the voltage changes on the wire are what describes the properties of the signal.

This dissertation will explore a notation that can be used to describe a complete system from design to implementation, which is embedded in an architecture being the same for design and implementation. This allows designers not only to exchange information in one common language, but also allows engineers to validate the interaction between the implemented system components and the design components.

The dissertation takes a wider view of the system engineering problem leading to system-wide improvement (one architecture from design to implementation) rather than localised optimisation. The notation introduced is, in the first place, able to describe real-time systems in a certain architectural context and, in the second place, efficiently mapped to hardware or software exhibiting the same architectural computational model as the design.

The architecture described in this dissertation has been used in the SUNSAT microsatellite project for the last 8 years (see Section 4.6 Chapter 4). For actual implementation the language Modula-2 was employed (see Appendix A for details on the language choice).

There are four distinct research efforts in improving the engineering process for real-time systems in this research:

1. the development of an architecture the engineering of real-time systems being suitable for design and implementation in software and hardware,
2. the consolidation of a number of graphic languages into a graphical notation for functional specification, design and construction of real-time systems,
3. the development of a simple processor architecture for the execution of real-time applications, and
4. the evaluation of the application of the architecture to the engineering of a microsatellite case study.

In particular, the following contributions are made:

1. the firing semantics of data flow systems are expanded to include disjunctive firing semantics in a novel way in addition to the classic conjunctive firing semantics,
2. the inherent real-time data flow property introduced by [52], i.e. that a receiving task must be ready to receive the next incoming message when it is sent, is extended to the synchronous data flow model,

3. a notation for describing all properties of a real-time system is defined with a real-time data flow language (RDF) as the base language,

4. two hardware processor architectures are introduced that offer one-to-one correspondence between design and implementation, and

5. the class of systems that can be modelled with data flow architectures is shown to include control centric systems and data flow systems.

The engineering process, language set and processor architecture were applied to certain aspects of the SUNSAT microsatellite project [77]. In particular, the engineering process was influenced by the actual practice (see Appendix A), the RDF architecture was applied to the high level design [91], the RTX operating system kernel that schedules the task set, implements the RDF architecture (see Section 4.2.3 Chapter 4), the processor architecture has influenced the current satellite instrumentation bus architecture and a new generation architecture is proposed for a follow-on microsatellite (see Section 4.5 Chapter 4).

1.5 Overview of the Dissertation

Chapter 2 reviews the existing techniques for real-time system engineering against the background of computing as a discipline, identifies the sources of error in software, the key aspects of real-time systems, and touches on how to manage complexity. Advances in scheduling, architecture descriptions, and data flow architectures are discussed. From the background information a motivation for the approach taken in this dissertation is derived.

Describing all properties of a real-time system requires a notation that can express all characteristics of interest. In particular, the structural and behavioural properties of a system require different description techniques. Furthermore, a system evolving from design to implementation requires an ever increasing level of detail to capture all required parameters. A multi-language notation is introduced in Chapter 3, of which the real-time data flow language RDF forms the core language. Other existing languages complement an RDF system description to ensure a complete description of all system properties.

Hard real-time systems require guarantees from their implementation platforms with regard to performance. Traditional systems were implemented by multi-tasking on uni-processors. This has led to resource constrained implementations, whose time behaviour is, due to the complexities of modern microprocessors, not predictable any longer without disabling some of the performance enhancing features such as caches for example. Chapter 4 introduces two resource adequate hardware architectures which not only guarantee certain performance parameters, but also exactly match the architecture of a system designed. These hardware platforms thus allow easier fault finding, and one-to-one mapping from a designed system to an adequate amount of resources which will guarantee performance under all defined circumstances. However, current applications must be implemented on existing processors. Therefore, Chapter 4 describes a pragmatic approach to implementing the RDF architecture on a single processor.

The computational architecture proposed has been used as the basis of the SUNSAT microsatellite's space segment software and ground segment software. The flight software and ground-station software successfully run on the RDF architecture. The existing implementation was carried out by conventional architecture translation processes. Chapter 5 shows how the RDF architecture is useful on the modelling, analysis and design levels, and explores the temporal

properties of a software suite that could be expected to run on a microsatellite such as SUNSAT. A case study in the signal processing domain is included in Appendix E to compare the modelling capability of RDF with the current state of the art signal processing modelling system DF* [15].

The dissertation concludes with considering its main focus, the motivation for the work done, and a review of the contributions made. As the process of real-time system engineering is still open for further improvement, a brief section is included considering aspects that should be investigated further.

This dissertation is set against the backdrop of the software engineering development process as found in Appendix A. Some results of applying the process in an innovative university environment are included. Most of the examples used in the dissertation are demonstrated in a case study of the SUNSAT microsatellite [77]. An early high level functional software requirement specification can be found in Appendix B.

Chapter 2

Background

Traditionally hand crafted solutions, multi-tasking libraries or real-time operating systems are used as implementation media to solve the embedding of real-time software. A key driver of the implementation efforts is the efficient use of limited resources. In this area a significant amount of work has been done in task concurrency management on limited resources.

This chapter reviews the existing techniques for real-time system engineering against the background of computing as a discipline, identifying the sources of error in software, the key aspects of real-time systems, and how to manage complexity. The progression of languages for real-time systems is reviewed. Advances and results in task concurrency management are followed by a view on architectures, from where the discussion moves on to data flow architectures. From this background review the motivation for the approach in this dissertation is derived. It will become clear that, the major problems are related to the lack of a commonly agreed upon architecture being suitable for engineering real-time systems. Lack of an architecture also means lack of a formal design method and of tool support, as well as a semantic gap between functional specification and implementation.

2.1 Computing

Computing as a discipline consists of the following three fundamental paradigms [17]:

1. Theory is rooted in mathematics and involves the following four steps:
   (a) characterise objects of study (definition)
   (b) hypothesise possible relationships among them (theorem)
   (c) determine whether the relationships are true (proof)
   (d) interpret results
2. Abstraction (modelling) is rooted in experimental scientific methods and consist of the following four stages:
   (a) form a hypothesis
   (b) construct a model and make a prediction
   (c) design an experiment and collect data

(d) analyse results

3. Design is rooted in engineering and consists of:

   (a) state requirements
   (b) state specification
   (c) design and implement a system
   (d) test the system

The nine domains which the Task Force on the Core of Computer Science identified are [17]:

1. Algorithms and data structures
2. Programming languages
3. Architecture
4. Numerical and symbolic computation
5. Operating Systems
6. Software methodology and engineering
7. Databases and information retrieval
8. Artificial intelligence and robotics
9. Human-computer communication

There is a need to integrate the theories and the models into the creative engineering process so that the design process can be underpinned by rigorous models and theories that can be used to assist in describing and evaluating system properties throughout the engineering process. The domains which need to be consolidated for a significant improvement in engineering methodology are: programming languages, architecture, operating systems and software methodology and engineering.

2.2 Software Reliability

The sources of errors in software fall in two categories; those errors induced in the development phase and those errors which can occur in the environment in which the software operates. [68]:

1. The sources of possible errors in the development phase include:

   (a) system design errors due to incomplete specifications and lack of verification of system level properties
   (b) programming errors that are generated in the coding phase of the software

2. The sources of possible errors in the program execution phase:

(a) supporting fault tolerant architectures implies two runtime operations, viz. (1) detecting errors, and (2) system repair so that the errors do not lead to faults

(b) handling of illegal input data

(c) external disturbances due to hardware failure, radiation, crosstalk EMI transients, etc.

The development phase is of particular importance as all sources of unreliability induced during its development are endemic to a system.

2.3 Real-Time Systems

In [19] **real-time operation** is defined as follows:

> Real-time operation is the operating mode of a computing system, in which the programs for the processing of data arriving from the outside are permanently ready in such a way, that the processing results become available within a priori given time frames. According to the applications, the data may become available for processing either at randomly distributed instants or at pre-determined points in time.

Predictability implies that the temporal behaviour is known, and that a system is always on time. It can be assured if all computations have acceptable known bounds of execution time. If the computations do not have known acceptable timing bounds, the actual execution time for each computation must be measured and corrective action taken if required.

The recognition that using imprecise answers may be better than having no results at all led to the work of [72, 98, 74]. If imprecise answers are acceptable, then the system parameters such as the required computational bandwidth could be decreased to allow enough time for sufficiently correct answers.

In modelling real-time systems for purposes of system analysis and design it is of particular importance to realise that every theory or model is an abstraction omitting some aspects of reality. Addressing the misconceptions or half-truths on real-time systems, it is noted in [57] that it does not make sense to abstract real time out of a system model with which one needs to reason about time itself. Reality is deterministic, but incompletely known. This implies that the higher the level of a model is the more non-deterministic it is, and as it is refined towards an implementation, it becomes more deterministic.

When modelling a real-time system, even at the highest level, the expressibility of temporal properties is essential.

2.4 Complexity

Complex systems are characterised by the following features [102]:

1. Newly developed systems need to be integrated with existing systems on widely distributed and dissimilar platforms.

2. Systems must simultaneously satisfy complex combinations of objectives.
3. System must adapt their behaviour dynamically to fluctuating environmental conditions.
4. Systems are expected to last for decades to amortise huge development cost.
5. Systems need to satisfy application and system requirements including security, robustness, coherence, real-time operation and physical distribution.

It is of interest to note that working with complex systems as defined in [102] involves an umbrella of conceptual views:

1. The *functional view* presents a system in terms of active processes and their dependencies, how both interact, and how they use passive resources.
2. The *timing view* presents the time constraints of each process, plus the resource or interaction timing requirements; for example, a process may be strictly periodic, with a hard deadline equal to its period, and requiring a certain resource for a given amount of time in each period.
3. The *fault tolerance view* deals with the robustness and reliability requirements for each process and each use of or interaction with a resource. Thus, a process may need to survive the total disconnection of any one system site.
4. The *security view* presents such requirements as the specific clearance needed for access to a specific resource.

A hierarchical definition of the operational views above must be supported. For large complex systems, only the relevant interactions should be described. The internal behaviour of components is of no concern to a complete system, rather the interaction between them.

Recognising complexity in hardware is easy [116] considering the advances in computer technology. From 1978 to 1993, in its line of 80X86 processors Intel increased computational power by a factor of 335, transistor density by the factor 107, and price threefold. What, on the other hand, drives software towards complexity? The work reported in [116] shows that software vendors uncritically adopt any feature users want. The following specific issues are highlighted:

1. All features are present all the time.
2. Complexity is not sophistication.
3. Software is all design as duplication costs are insignificant. Truly good solutions emerge after iterative improvements or after re-designs that exploit new insights.
4. There is never enough time due to time-to-market pressure.
5. Good engineering products are a result of refinement. As software has no manufacturing costs, there is no incentive to apply meticulous engineering habits in software.

A possible solution is to reduce software complexity by concentrating on the essentials, only. The goal of the Oberon project was to show that software can be developed with a fraction of the memory capacity and processor power normally required, but without sacrificing flexibility, functionality or user convenience. The Obeoron project relied on three underlying tenets:

1. Concentrate on the essentials.
2. Use a truly object oriented programming language, which is type-safe.
3. To achieve simple, efficient and useful systems, systems must be flexibly extensible.

From the above discussion it is clear that complex systems last long due to high development costs. Suggestions to better understand complex systems are motivated from multiple views, and by the need to concentrate on the essential functionality.

2.5 Notations and Languages

A computer language is a means to express design intentions. It provides the ability to communicate about an application with others and with the computer (computing device). The emphasis on programming stems from a long-standing belief that learning a programming language is an excellent vehicle for gaining access to the rest of the field [17].

2.5.1 A Brief History

In the 1970s it was widely believed [116] that program design must be based on well structured methods and layers of abstraction with clearly defined specifications. The abstract data type best illustrated the idea in languages like Modula 2 and Ada. In the 1980s the abstract data type re-appeared under the heading of object orientation. In particular, object oriented languages must embody strict, static typing that cannot be breached, whereby programmers can rely on compilers to identify inconsistencies. Compare this to the most popular object oriented language C++. Graphical programming became possible in the 1990s with the introduction of Delphi and Visual C++ (Basic), and the programming environments around the Java language. In most cases a particular event handling architecture is assumed for the computer console interaction, and a programming environment is provided to build systems in that architecture.

One example of an integrated environment for the representation of real-time systems is the SARA [23]. The SARA environment supports modelling in the control domain with minimised modelling in the data domain. The control domain notation is, with restrictions, equivalent to the Petri net model and supports methods for reachability graph analysis. In particular, a structural model and behavioural model are supported by SARA. The structural model is fully hierarchical. The behavioural model uses the Graph Model of Behaviour (GMB) language allowing to build a functional model in the three domains control, data and interpretion. The control domain primitives include nodes and control arcs. The data domain primitives include processors, data sets and data arcs. The processor domain only specifies which data sets a processor may read or write. The interpretation domain specifies what the processor activity may include.

2.5.2 Parallel and Real-Time Programming

For parallel programming, the parallel instructions can be explicitly specified such as in Occam [47], or the parallelism in an algorithm can be extracted as in vector processors or in data flow languages [112]. As data flow languages have been shown to be optimal in the extraction of parallelism, they are considered here for expressing real-time systems.

Data Flow Programming

The data flow programmer specifies the *data* to be processed and the *transformations* to be applied. Languages such as Lucid [112] allow this in a very succinct way. Data flow programming is possible on a high level, e.g. in the form of Unix pipes, or on a very low level such as in Lucid. The paradigm is, thus, applicable for a large variety of problems. Parallelism is expressed and extracted in the most natural way in a data flow language [96]. Conditional activation of a function based on a Boolean condition which is expressed in terms of input messages was described by [86].

Real-Time Programming

Real-time programming consists of expressing the interactions between the real world and a system interacting with the real world with temporal determinism. This is currently achieved by the elimination of constructs that have indeterminate execution times such as in Real-Time Euclid [34], by extending existing languages with operating system support [70, 51], and by designing new languages in which temporal properties can be expressed [34].

Two phenomena make real-time programming more difficult. Firstly, general purpose processors are becoming more complex and the growing trend to superscalar architectures with large caches renders temporal prediction virtually impossible. Secondly, final programs derived with the main stream scheduling mechanism, Rate Monotonic Scheduling (RMS) [95], bear little resemblance to the expression of original temporal properties.

Real-Time Data Flow Programming

Data flow programming lets the data flow determine the program execution rather than the control flow. This makes the execution time of data flow programs inherently data dependent.

Real-time features were introduced to data flow languages with the expression of temporal properties of data streams. The work reported in [99] uses an earliest and latest time stream specifying the earliest and latest time at which data are produced, and RT Lucid [24] proposes a pair of time values denoting the earliest and latest time each data item is produced. A far simpler approach is found in [52] where the maximum rate at which data are produced is indicated on each outgoing arc. Each processor must then ensure that it consumes the data in a shorter or just as short period as the incoming period.

Data Flow languages appear to be good in describing parallel execution, both on high and low levels. This observation is chosen as a major departure point for the work in this dissertation. A real-time program must contain, in the clearest way possible, the temporal properties of that program. A deadline gives a far more accurate view of the temporal properties of a program than priorities allocated to tasks. Detailed scheduling information should be of no concern to the program author. In addition, all constructs which allow temporal non-determinism must be eliminated, or their execution time bounds must be managed.

2.6 Software Engineering Practice

On completion of the Oberon project the following lessons learnt were reported:

1. The exclusive use of a strongly typed language was the most influential factor in designing this complex system in such a short time.
2. The most difficult design task is to find the most appropriate decomposition of a whole into a module hierarchy, minimising duplications of code and function.
3. Oberon's type extension construct was essential for designing extensible systems where new modules added functionality and new object classes integrated compatibility with existing classes or data types.
4. In an extensible system, the key issue is to identify those primitives that offer most flexibility for extension.
5. A system that is not understood in its entirety, or at least to a significant degree of detail, by a single individual, should probably not be built.
6. Keep teams small to eliminate communication problems.
7. Reducing complexity and size must be the goal in each of the steps system specification, design and in detailed programming.
8. To gain experience, there is no substitute for one's own programming effort.
9. Programs should be written and polished until they acquire publication quality.

The lessons reported have particular relevance to the software engineering process described in Appendix A, which was proposed for the flight software of the SUNSAT microsatellite.

2.7 Scheduling Task Sets

The execution of a program with more than one parallel task on a single processor (or a limited number of processors) has received a lot of attention in the past [11, 66, 88, 92, 14, 10, 13]. Work in this area is expected to continue because determining a perfect schedule is an NP complete problem.

2.7.1 Periodic Tasks

In fixed priority pre-emptive scheduling, two issues were focussed on, viz. (1) policies for priority assignment, and (2) feasibility tests for task sets. Restricted periodic task sets [71] are characterised as follows:

1. deadlines are equal to the periods,
2. tasks are independent,

3. tasks have fixed execution times, and
4. tasks are released for execution as soon as they arrive.

For such a periodic task set, a feasible schedule can be found by scheduling the tasks according to the Rate Monotonic Assignment (RMA) scheme. Its results have been improved with algorithms for handling priority inversion and resource blocking [94].

2.7.2 Aperiodic Tasks

For more general aperiodic task sets a scheduler must be able to:

1. guarantee the observation of hard deadlines of aperiodic tasks,
2. provide good average response times for tasks with soft deadlines (whose requests may be non-deterministic), and
3. accomplish the above goals without compromising the hard deadlines of periodic tasks.

Solutions for managing aperiodic tasks in conjunction with periodic task loads have been proposed employing both bandwidth preserving and non-bandwidth preserving algorithms.

Non-Bandwidth Preserving Background servicing of aperiodic tasks is only carried out when the processor is idle. If no aperiodic task is available to run, the expected processor utilisation for aperiodic task use is given up for use by the periodic task set.

Bandwidth Preserving Various algorithms have been proposed [100] which all create a periodic task for servicing aperiodic requests. The available time can be managed by allocating it to an outstanding aperiodic task request or, if no aperiodic tasks are available, to a lower priority periodic task. The lower priority time slot is thus available in the future when an aperiodic task request arrives.

Taking advantage of slack time, i.e. reclaiming spare time, was addressed by [65] with a static slack stealer which, at each priority level, keeps track of all early task terminations, and adds up the time that can later be applied for soft real-time tasks. The algorithm has been proven optimal. With an execution time of $O(n^2)$, however, it is infeasible to be applied in practice.

2.7.3 Sporadic Task Sets

Sporadic tasks are aperiodic tasks with hard deadlines *and* minimum inter-arrival times. Non-pre-emptive Earliest Deadline First (EDF) scheduling was proven an optimal scheduling algorithm for sporadic tasks, and [53] derived necessary and sufficient conditions for testing the schedulability of a set of periodic and sporadic tasks with arbitary release times and a non-idle time inserting scheduling algorithm.

Hard real-time systems interact with environments which have unpredictable, but bounded temporal properties. Task sets with sporadic activation behaviour are expected to run in such real-time systems, and the results of scheduling sporadic tasks are useful in this context.

2.8 Architectures

The architecture area is defined in [17] as follows:

> "This area deals with methods of organising hardware (and associated software) into efficient, reliable systems."

This definition stems from the paradigm that a hardware architecture is dictated by what is offered on the market, and that the application must be transformed to fit the hardware through a combination of hardware and software. A more correct definition of architecture would be:

> The structure and behaviour of computations to solve a particular problem.

A definition which is underpinned by the practical experience of using architectures is given by [18]:

> "Software architecture is the structure of the components of a program or system, their interrelationships, and the principles and guidelines governing their design and evolution."

Different application areas have different architectures. The areas of interest in this dissertation and the corresponding architectures are: a layered architecture suitable for the construction of communication software, a data flow architecture suitable for the construction of signal processing, instrumentation management and dynamic control systems, and a state based architecture suitable for the construction of command and control systems. Common to all these architectures is a set of tasks which performs its collective function in parallel and by communication.

2.8.1 Frameworks and Components

Frameworks and components are another way of referring to the elements of an architecture. Lee [64] motivates component technology and comments that:

> "However, as a component technology, processes and threads are extremely weak. A composition of two processes is not a process (it no longer exposes at its interface an ordered sequence of external interactions). Worse, a composition of two processes is not a component of any sort that we can easily characterise. It is for this reason that concurrent programs built from processes or threads are so hard to get right. It is very difficult to talk about the properties of the aggregate, because we have no ontology for the aggregate. We do not know what it is."

Lee [64] motivates that the key challenge in embedded software research is to invent frameworks with properties that match the application domain better. One of the requirements is that time must be re-introduced. In this context, a framework is a set of constraints on components and their interaction, and a set of benefits that derive from these constraints. In order to evaluate the usefulness of a framework, it is helpful to orthogonalise its services:

Ontology A framework defines what it means to be a component. For example, is a component a subroutine, a state transformation, a process, or an object? Is a component active or passive (can it autonomously initiate interactions with other components or does it simply react to stimuli)?

Epistemology A framework defines states of knowledge. What does the framework know about the components? What do components know about one another? Can components interrogate one another to obtain information (i.e. is there reflection or introspection)? What do components know about time? More generally, what information do components share? Scoping rules are part of the epistemology of many frameworks. Connectivity of distributed components are another part of the epistemology.

Protocols A framework constrains the mechanisms by which components can interact. Do they use asynchronous message passing, rendezvous, semaphores, monitors, publish and subscribe, timed events, or transfer of control? In the latter case, the components are states and their interactions are state transitions.

Lexicon The lexicon of a framework is the vocabulary of the interaction of components. For components interacting by sending messages, the lexicon is a type system that defines the possible messages. The words of the vocabulary are types in some language (or family of languages, as in CORBA).

2.8.2 Java

Java offers a complete computing architecture, satisfying the goal for an architecture to describe a system (the programming language) and the implementation architecture (Java Virtual Machine and Java processors). Java is used in general purpose applications and embedded applications. However, no explicit provision is found in Java for supporting real-time systems [41]. Its object orientation offers a way to describe a system at a high level. It is, however, not restricted to a clear architecture. The elements supporting the Java architecture are:

1. Java OS — providing a compact operating system for computing devices executing Java programs,
2. Java Virtual Machine — a layer of software on conventional operating systems that enables the computer to run Java code,
3. Java APIs — standard software interfaces between Java applications and run-time services,
4. Java chips — a family of processors optimised for executing Java code.

2.9 Data Flow Architectures

Architectures proposed for real-time systems include a Blackboard [38, 73], Communication State Machines [97], Extended State Machines [87], Intelligent Agents [39], a Modular Architecture [29] and Data Flow Architectures [67, 103]. Data flow architectures offer inherent expressibility of parallelism. A significant amount of progress has been made studying both static and dynamic data flow systems.

A data flow system consists of a number of actors (tasks/processes/nodes) that are interconnected by edges (communication channels). Processing takes place with each actor operating in parallel. Each actor repeatedly goes through the following sequence:

1. consume the tokens at its input ports according to a firing rule,
2. execute code that operate on the tokens, and
3. produce tokens as output which is sent through the outgoing ports to the edges that leaves the actor.

2.9.1 Static Data Flow

Static data flow architectures refer to a particular computation structure where the order of execution can be determined at compile time. The execution schedule is, thus, static and there is no non-determinism in the execution order. Hence, the temporal properties of a system can be determined statically at compile time. Static data flow systems evolved from homogeneous data flow systems to cyclo-static data flow systems.

Homogeneous Data Flow (HDF)

An actor produces one token for another actor which consumes the token. For a node B and for all input ports on B the firing rule is:
HDF Firing rule: $\exists B, \forall i \cup (in_i\# \geq 1)$ where $\forall i$ in_i is the set of input ports on node B and $in_i\#_i$ is the number of tokens on input port in_i.

Synchronous Data Flow (SDF)

An actor can produce n tokens which must be consumed by the consumer. This was extended so that the number of tokens can vary from iteration to iteration. For a node B, and for all input ports in_i the firing rule is:
SDF Firing rule: $\exists B, \forall i \cup (in_i\# >= n_i)$ Node B consumes n_i tokens on input port in_i.

Synchronous data flow is also known as **static data flow (SDF)** and SDF is associated with a signal-flow graph where each edge can carry multiple tokens; each node represents some operation or complete process. A simple firing rule is associated with each graph node. It states how many tokens must be present in each fan-in edge before it can fire, and how many tokens will be produced after firing the node. In SDF these numbers are noted on the graph. In addition, it is assumed that when a node can fire, it is fired. These properties enables the determination of a static schedule of the nodes [60].

Cyclo-Static Data Flow (CSDF)

Departing from the view point that a model must be statically analysable and, hence, statically schedulable, the data flow computational model was advanced to the Cyclo-Static Data Flow (CSDF) model described in [6]. Here, each actor has a number of code segments that can execute

cyclically one after the other. Each code sequence has its own production and consumption rule.

CSDF Firing rule: $\exists B, \forall i \forall p \cup (in_{i,p}\# >= n_{i,p})$ Node B consumes $n_{i,p}$ tokens on input port in_i in phase p.

The CSDF model can be extended to allow any code segment to be selected based on the value of a local variable. The local variable can have a new value assigned on receipt of the value of an input token. This leads to the next form of data flow, viz. dynamic data flow.

2.9.2 Dynamic Data Flow

Dynamic data flow architectures provide for non-deterministic choice in execution paths and data dependent choice in execution. In general, it is not possible to determine a static schedule, and the execution order of the actors is determined at runtime.

Boolean Data Flow

Boolean data flow, which is also called token flow graphs [63], adds two actors which provide for limited data dependency in a data flow graph. Requiring data dependency and making decisions in the design graph establishes the need for these actors. The two special actors that are added to the generic actor types are:

1. Switch — control input determining to which output the one input is going to be switched,
2. Select — control input determining which of the two inputs is selected for output.

Dynamic Data Flow

There are two schools of thought. On the one hand data flow is extended with a limited set of additional actor types such as Merge, Select, and Switch. While on the other hand, data dependent data flow is enabled in the strongest sense of the word, viz. a firing rule can contain any combination of tests based on the values of variables, number of tokens in the queue, internal state of the actor, etc. In the constrained instance, the Merge actor type is added to the actor set. Merge selects non-deterministically which of its inputs should be selected and transferred to the output. A general definition of Dynamic Data Flow (DDF) can be found in [106].

Definition 1 (Dynamic Data Flow) *DDF is associated with a similar type of graph as SDF, but the firing rule is more general and consists of: (1) a condition rule, (2) a consumption rule, and (3) a production rule. The condition rule states (1) how many tokens must be present in each fan-in edge, and (2) what values the data tokens must have. The latter is represented by a Boolean condition. The consumption (production) rule states how many tokens will be removed (produced) from the fan-in (fan-out) edges. In addition, the semantics are that if a node can fire, it need not fire right away. Because of the Boolean conditions, the flexibility of the consumption and production (which do not need to manifest numbers on the graph) rules, and the asynchronous firing of the nodes, a static schedule cannot be derived anymore [63].*

2.9.3 Generalising Data Flow

The firing rules of data flow can be abstracted to the following level:

Definition 2 (Data Flow) *Firing rule = f(#tokens on each input port, [semantics of input and output ports (i.e. BDF (switch/select), DDF (merge etc.), RDF (OR,AND)], [value of input token] (RDF,DF*), [type of input token] (RDF), [value of internal variable] (DF* loops))*

Even with all the advances in data flow computational architectures, no explicit provision has been made for representing temporal properties. Recognising that not all systems can be represented with predictable data flow static scheduling, the current focus is on dynamic data flow mechanisms, again without the explicit expression of temporal properties.

2.10 Integrated Real-Time Computation Modelling and Implementation

There are four specific research efforts aiming at improving modelling and realisation of embedded systems [106, 104, 28, 15]. As these efforts represent the current state of the art in data flow computing, it is useful to review them.

2.10.1 DF*

DF* is described in [15]. It is a system of data flow nodes of which the function execution of each node is controlled by an associated finite state machine expressed in SDL. The data flow is thus controlled by the control flow. DF* separates control and data paths with a different execution semantics for each. The control path of DF* and the state machine control the data flow paths and code segments being executed. Another way to consider DF* is that the state machine enables different input port sets to fire based on the internal state of a task node and the control flow of each node. The DF* extended firing rule is based on the number of tokens in an input queue and on the value of a token, by first reading it, storing it locally and then comparing it. The trigger condition can test whether there are no tokens on another input than itself. This removes some non-determinism from a graph. The actual state transitions can also test the value of a local variable.

2.10.2 Multi-Thread Graph (MTG)

The Multi-Thread Graph system builds on Petri nets for its computational architecture. The fine-grain behaviour of the MTG model is based on the control data flow graph (CDFG) described in [58]. It makes provisions for elementary operations such as add/multiply and interface operations (input, output). The edges indicate data dependencies. Additional control constructs are added to express basic blocks, conditional scopes and loops. An MTG adds the following timing constraints to a system:

1. latency timing constraints,
2. response timing constraints, and

3. instantaneous operation rate constraints and average rate operation constraints.

Structural information is represented by the graph nodes and their interconnections. The model is equipped with token flow, expressing the system dynamics. Petri net type control flow has tokens that do not have a meaning other than activating the execution of a function. Data are transferred between nodes and the environment through IO ports and variables which are explicitly modelled as entities in an MTG. The control flow resembles data flow very closely, except that the firing rules are now dictated by tokens without values, while in the case of data flow the tokens have values.

2.10.3 Funstate — State Machine Controlled Flow Diagrams (SCF)

Funstate diagrams [105] consist of functional components (data flow components) controlled by an associated FSM. The Funstate model was shown to represent cyclo-static data flow, SDF, marked graphs, communicating state machines, as well as Petri Nets. The model is characterised by the following properties:

1. A system consists of components interacting by exchanging information via queues and registers (a register is a queue of finite length; when full it is overwritten with new data).
2. A component can contain other components leading to hierarchy.
3. In each component there is a separate state based and functional part. The functions are activated under the control of the state machine. This is similar to the mechanism proposed in DF*.

The firing rules of the different data flow models are, thus, embedded in the FSM part of each component. The extended firing rule for Funstate is a function of the number of tokens in a queue and the value of a token received. The model supports hierarchy through components, refinement and abstraction as a component can be replaced by another one which is refined (design purposes) or abstracted (specification purposes). Parallelism and non-determinism are supported.

2.10.4 Hierarchical Finite State Machines with Multiple Concurrency Models

The work in [28] assumes that FSMs are the basis of modelling systems, and then explores the hierarchical combination of FSMs with three different concurrency models, viz. data flow, discrete event and reactive systems. Data flow systems have various forms such as homogeneous, synchronous, cyclo-static, Boolean or dynamic. In each case the rules for the number of tokens produced and consumed are more general. Discrete event systems execute the next node for the token on any edge with the smallest time stamp. Each event is thus time stamped. This corresponds to the simulation of digital hardware in which the next event is the one with the next time stamp. There is some analogy with the Earliest Deadline First scheduling paradigm where the next task to execute is the one with the earliest deadline. Synchronous or reactive systems execute at particular time intervals called ticks, and are suitable for modelling parallel hardware systems. Time delays in computations become irrelevant, as long as they are executed

within a time cycle. Perhaps the most important contribution described in [28] is exploring the semantics of a finite state machine inside a node of a data flow graph and vice versa, and of a data flow node inside a finite state machine. However, no explicit modelling of temporal properties or requirements is given.

2.11 Unified Modelling Language (UML)

The Unified Modelling Language has evolved from the field of object orientation to provide one modelling system for all computational systems. It is useful to briefly consider the UML modelling system in order to identify what can be described in the views of UML. UML offers 5 views of a system, each with a specific purpose and used by a particular rôle player in a project development cycle. Each view can be described by one or more languages.

Component view Shows the organisation of code components
to describe/analyse: description of implementation modules and dependencies, design refinement and implementation,
used by: developers,
diagrams: component diagrams with optional project management information such as resource allocation and development status.

Logical view Shows the functionality inside a system in terms of structure and behaviour
to describe/analyse: how system functionality is provided, static structure and dynamic collaborations,
used by: designers and developers,
static structure diagrams: class and object diagrams,
dynamic collaboration diagrams: state, sequence, collaboration and activity diagrams.

Deployment view Shows the physical deployment of a system
to describe/analyse: physical architecture of hardware and software,
used by: developers and integrators,
diagrams: deployment diagrams.

Concurrency view Division of a system into processes and processors
to describe/analyse: communication and synchronisation,
used by: designers,
dynamic diagrams: state, sequence, collaboration and activity diagrams,
implementation diagrams: component and deployment diagrams.

Use-Case view Functionality of a system as perceived by external actors
to describe/analyse: typical scenarios of system deployment,
used by: external actors,
diagrams: use-case and activity diagrams.

UML offers an integrated environment to support engineering from user **requirements** *through to final* **implementation**. *There is currently no support for real-time systems, but efforts are underway in this direction [93].*

2.12 Motivation for this Thesis

The background study above clearly illustrates that, although a system can be conveniently modelled by a set of interacting processes [106, 105, 28, 15, 43, 62], the problem to design and implement real-time embedded software is not completely solved, yet, and that contemporary approaches still suffer from shortcomings. Major sources of the latter are embedded in the development process, expressing temporal properties is not supported by development systems, complexity is growing as system size increases, languages used for real-time development are not tailored for expressing temporal properties, and there is no high level architecture for real-time systems for which a corresponding implementation architecture exists. A notable exception of modeling languages that does not represent time, is the MTG [106] system which explored the problem of building real-time systems within a Petri net model for automatic system synthesis. However, the basic requirements for a simple model underpinned by a sound theory that can be used to support real-time system engineering by humans is still open to be addressed. Specific requirements for a new approach to the engineering of real-time systems are:

Real-time support From a specification point of view, a model must have operational timing semantics, and capture both functional timing (e.g. execution time behaviour and occurrence rates) as well as timing constraints. For multi-tasking, software scheduling has to be time-driven and not rest on priorities as offered by current real-time operating systems.

Real-time architectural model A detailed and rigorously defined architectural model is required for unambiguous communication.

Automated design support Design support should assist to close the gap in the abstraction level between high level description (i.e. communicating tasks) and final implementation on (a) target processor(s).

Easy system validation The cost of post-design validation, performed by testing or simulation, is unacceptably high for contemporary systems [106]. The advantages of correctness built-in during the design and implementation phases becomes important. If it can further be achieved that a post-designed system corresponds one-to-one with its pre-designed version, then the post-implementation timing and functional validation should be easier.

Correspondingly, in this dissertation the following particular aspects will be addressed.

1. A data flow paradigm will be introduced that is suitable for describing systems which interact with an environment that exhibits parallel execution behaviour. The review in this chapter has shown significant progress in the development of data flow architectures for describing systems of increasing complexity. The key of this dissertation is to put time back into data flow systems, so that real-time systems can be succinctly modelled and engineered.

2. A second effort is to close the semantic gap in the description of a system between the functional specification step, the design step and the implementation step. This will assist in eliminating the sources of errors due to the translation between different architectures in the development process.

3. A third effort is to move away from a resource constrained paradigm to a resource adequate paradigm which is, in fact, what a hard real-time system requires by definition. New implementation media and assumptions are required to bridge the gap between squeezing

in the last function on a processor to being able to certify that (the) processor(s) have enough capacity to guarantee execution of all tasks under all known circumstances.

Chapter 3

Presentation of Visual Languages

The problem of developing and understanding real-time systems is that there is a multitude of languages used to implement real-time systems, but none of the standard languages has any notion of specifying and validating temporal properties of constructed systems. The current state of practice includes languages such as C, C++ and more recently Java. Moreover, the current languages are mostly derived from an efficient processor execution point of view and not from efficiently matching the application domain.

Further, the lack of a common computational architecture (other than the Von Neumann architecture) makes the communication between engineers developing different systems in different languages difficult. For example, it is pointed out in [110] that a well defined architecture assists in addressing the demands for the development of embedded software. Posix [56] compliant operating systems for task scheduling and inter-processor communication constitute a step into the direction of a standardised operating system interface, but the computational architecture chosen better matches that of the processor on which it executes than the application domain of real-time systems.

Shorter development cycles and a higher level of dependability are required of systems that contain a major software part. Real-time properties of a system only form one aspect of system requirements. It is, however, important to establish a baseline for working with real-time properties so that the focus can move towards functionality and mission critical system requirements imposed by an application.

The lack of a common computational architecture for real-time systems and the accompanying lack of a high level language to describe systems in the architecture contributes to long development cycles and long validation processes in the development of real-time systems. This results in high project costs, and limits the number of areas where software can be employed in mission critical systems.

It is impossible to capture all the subtle details of a complex software system at a high level in just one view [7, 22, 85, 69, 114]. Multiple views on the same system are required to capture all the relevant details. The use of visual languages is an important tool for conceptualisation during the engineering of any system. Visual languages represent inter-relationships between different components in a way which makes system structure and behaviour clear at a glance. Textual languages complement visual languages by conveying the detailed information which cannot be represented succinctly in a visual language. The use of textual languages is supplemented with visual formatting of text to convey additional meaning.

A notable effort in the area of open source software has made the computational architectures of such systems visible to anyone who cares to read the source code. The downside is that system are described in C source code and sometimes C++ source code. These language, being Von Neumann centric, make it difficult to comprehend a system's architecture in terms of its application domain.

The rate driven data flow architecture is chosen as departure point for a real-time computational architecture. The data flow architecture was selected as a generic architecture based on the fact that engineering systems are represented in a data flow format. Stating a rate of production and deriving the associated deadline by which the production must be consumed, is a natural way to express real-time properties of many systems.

A successful real-time systems engineering process entails specification, design, implementation, re-usable code, fast development cycles and least effort maintenance. This chapter describes a suite of languages, with particular emphasis on a real-time data flow language (RDF), striving towards meeting all the goals stated. The other four languages are: an hierarchical state machine language, an object oriented class structure language, a system structure language and a project development status language. Some of the early investigations of this work have been reported in [82].

Re-usability and system specification are two opposing driving forces. System specification deals with the interconnection of building blocks for a specific functionality, while re-usability deals with how generic building blocks are developed and stored. Software must, on the one hand, be described by components which are independent of their application, but the software components must also be described in the exact nature of their application as they interact to produce desired results. The multi-view approach proposed in this chapter addresses this fundamental problem by encouraging both representations to co-exist.

This chapter begins with a description of the views that are required to describe a real-time system completely. This is followed by a detailed description of a visual real-time data flow language (RDF) which includes a novel combination of feature properties to describe real-time systems. Other visual languages are briefly introduced and the interrelationship of the languages are discussed. The use of this suite of languages on a number of examples is finally shown.

3.1 Expressing Multiple Views

The multiple views on a system must capture all those properties which are required to specify, analyse and reason about the system and to construct it.

Most of the languages described in this chapter, viz. RDF, stategraphs, class diagrams, structure diagrams and project development diagrams, include a visual and a textual component to enable system description in a form most appropriate for the engineer. Visual representations allow a faster overall description, while textual representations are more compact for representing detailed information.

In the development process two model types are identified, viz. a physical model and a logical model. The physical model might represent a level of iteration which becomes the logical model for the next refinement iteration.

1. The **logical** model is used for a logical description of a system after the requirements

definition and during the specification phase of development.

2. The **physical** model is used for the physical description of a system during the design and construction phase of development.

Both the physical and logical model types need to be described in a number of views to capture all aspects of a model:

Behavioural view is used for describing the behaviour of a system.

Structural view is used for structuring a system.

Two other views describing properties which are orthogonal to both models and views are:

Project development The project development view is used for managing the development and re-use of system components.

Component organisation The organisation of code for re-use requires a classification scheme.

The inter-relationships between the views described above are one of the keys to a complete description of a software component. The views inter-relate in describing a software component.

The following set of languages is used to describe systems from functional specification to implementation. Each of the languages expresses one or more of the views required in the previous discussion. The languages and the views that they describe are:

- The real-time data flow language (RDF) describes functional communicating objects (tasks), and how they co-operate to perform a required function, i.e. aspects of the structure and dynamic behaviour of a system for both its physical and logical models.
- Stategraphs are derived from Statecharts [35] and describe the internal behaviour of each object in addition to the inter-object behaviour in the logical and physical models.
- A structural language describes physical relations between software components and hardware components, i.e. the relationship between logical and physical models.
- Project development information describes the development maturity of a system and its components.
- An object oriented class structure describes the software component organisation for software re-use. It is orthogonal to the other views because the components defined in the object oriented class structure are re-used in specific instances in the physical and logical models.

We assume a top-down approach for the specification and design of systems. At each level of a hierarchical decomposition of a system, it can be considered with any or all of the views. However, only the useful views need to be applied to a specific hierarchical level.

The logical model and physical model consist of communicating objects. The objects can either be passive or active. When passive, the objects are in a library which can be expressed as function blocks. Active objects have a different thread of control. Both passive and active

communicating objects respond to receiving a message with some processing and the output or not of another message.

The ports of the communicating objects have specific semantics to ensure predictable execution in the data flow architecture. The ports operate either as AND-triggered or OR-triggered communicating ports. In addition, the output ports have associated rate functions attached enabling timing analyses. With input ports worst case execution times are associated so that schedulability analysis (test for feasibility) can be performed.

This chapter describes the real-time data flow language (RDF) in detail. The other languages, i.e. object oriented class structure and Statecharts, are briefly introduced as they have already been defined in detail elsewhere [7, 35]. The Statechart language is adapted to allow for scope rules to be applied to event and variable names. This change allows hierarchical systems to be constructed without the complexity of propagating events on a global scale. The structural representation is simply the allocation of software components onto hardware components. The project development information view makes use of graphical (shading) and textual representations.

The next section describes the real-time data flow language that is suitable for structuring systems as well as describing communication objects and the time semantics of the interaction between communicating objects. This language is derived from the traditional data flow language, extended with real-time semantics first identified by [52]. The language of [52] is extended with conjunctive input port trigger semantics, synchronous data flow semantics, a rigorous definition of the execution rule semantics and output port guards.

3.2 The Real-Time Data Flow Language (RDF) - Components and Structure

The visual real-time data flow language (RDF) is suitable for requirements analysis (expressing time), design (refinement) and programming (in the small). This is one of the keys to the construction of dependable systems, i.e. that a rigorous notation can be used from the conceptual phase through to programming/implementation. The communicating objects, which form the basis for the architecture supporting real-time computations, are represented in a visual data flow language. This section describes the structure of the visual data flow language and its semantics. The language is described in terms of tasks, as they represent the instances of communicating objects.

A system consists of a number of task nodes, each with a number of ports connected by channels. The composite task block is an enclosure for further structuring of systems, i.e. a composite task node can enclose a larger system. Channels interconnect output ports of task nodes to input ports on other task nodes. The environment is represented by the environment interface. A channel is for one-way traffic and has no type. A system can be constructed of any number of nested composite task blocks which, in the final layer, must consist of primitive task blocks. This model supports embedding of fine grain detail using other languages (graphical or textual). The model supports synchronisation, concurrency, data communication, hierarchy and timing requirements. The control and the data flow structure is one and the same for RDF, simplifying the representation of larger models.

In order to support real-time analysis, design and construction in RDF, a system's time behaviour is indicated by a production rate associated with each message type of the output ports.

This rate is interpreted as the maximum rate at which the transmitting task will produce messages of the specific type and is, in general, a function of the input rates on the task. For correct functioning of the system the receiver must consume messages at a rate greater or equal to the transmitting port rate. This mechanism was first proposed in [52], where the rates were associated with each of the channels of a homogeneous data flow system. A single message can trigger the execution of a function associated with an OR port, or multiple ports must have messages waiting before the execution of a function associated with the conjunctive relationship of all designated AND input ports. The class of networks which can be realised is investigated in Section 3.6.3.

All diagrams define a scope. Every name is visible in the scope in which it is defined and in all lower level enclosed scopes. The same scope rules apply to all languages in the suite. This enables rigorous name checking in all views on the same level of a hierarchical decomposition.

3.2.1 Formal Definition

The fundamental components are tasks that can perform one or more functions and which can be interconnected. The other elements in the language are ports, communication channels, mailboxes, ropes, single shot and periodic timers, environment interfaces and data stores. Ports provide access to tasks. The communication channel is an essential feature for interconnecting ports and, hence, tasks. Persistent data storage is modelled by the data store element, which is not a unique symbol, but a task node that can only be accessed through bi-directional ports. It is required to support access to persistent and shared data. Refer to Figure 3.1 for the language elements.

The formal definition of an RDF graph reads as follows.

Definition 3 *An* **RDF** $\mathcal{G}$ *is defined as a 7-tuple* $(O, D, E, S, V, V_b, \Lambda)$ *where:*

O *is the set of all* **nodes** o_i. *A node* o_i *has a type, denoted by* $type(o_i)$, *with* $type(o_i) \in \{task, compTask, dataStore, enviri, enviro, pertimer, shotimer\}$.

D *is set of all* **data stores**, d_k *of* $\mathcal{G}$.

E *is the set of all* **asynchronous edges** $e_{i,j}$ *of* $\mathcal{G}$, *with* $E \subset O \times O$.

S *is the set of all* **synchronous edges** $s_{i,j}$ *of* $\mathcal{G}$, *with* $S \subset O \times D$.

$V = V_o \cup V_h$ *is the set of all* **data ports** p_i *of* $\mathcal{G}$. $V(o)$ *with* $o \in O$ *is the set of data ports for task* o, *with* $\forall o \in O | type(o) \in \{task, compTask, enviri, enviro, pertimer, shotimer\} : V(o) > 0$ *and with* $V_o = \cup_{\forall o \in G} V(o)$. V_h *is the set of composite ports on the border of* $\mathcal{G}$. *A port has one of the following directions:* $\{in, out\}$. *Orthogonal to the port direction is the port type:* $\{AND, OR\}$.

V_b *is the set of all* **bi-directional ports** *of* $\mathcal{G}$ *where* $V_b \in (O \times S) \cup (S \times D)$

$\Lambda : B \rightarrow N^+ \times N^+$ *is a* **time function** *associating an execution duration* $[\delta(B_i), \Delta(B_i)]$ *with each task, data store and task composite node* $B_i \in B$.

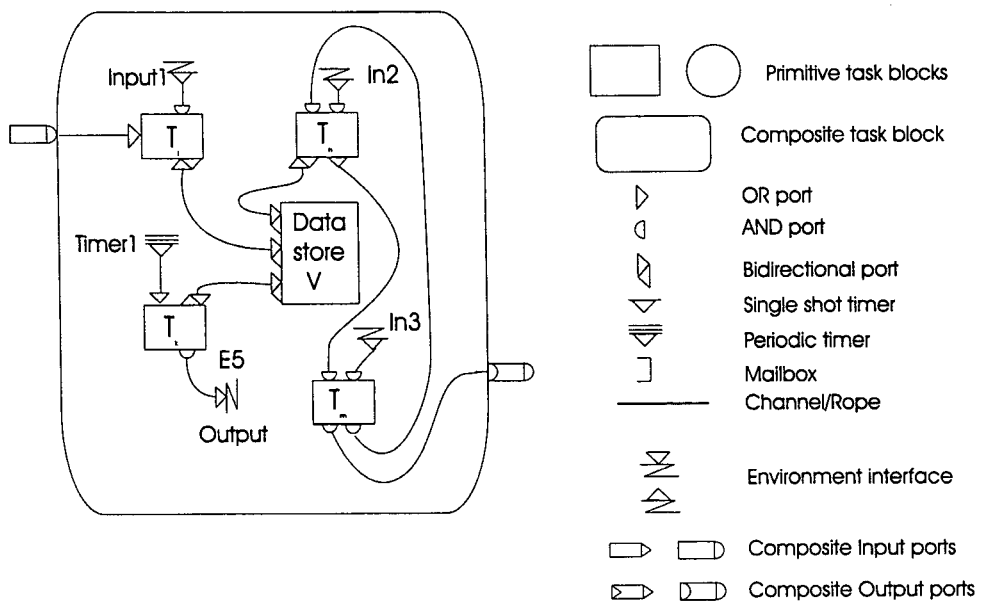


Figure 3.1: Elements of the visual language RDF

3.2.2 Nodes

A node $o_i \in O$ can be a task, composite task, data store, environmentInput, environmentOutput, periodicTimer or a singleShotTimer. Tasks, composite tasks, and data store nodes are called *behavioural nodes*, the other nodes are called *time control nodes*. The set of all behavioural nodes $B_i \in O$ is denoted by B. All nodes except the composite task form the set of operational nodes, see Figure 3.3.

The different nodes are distinguished from each other by node type:

Definition 4 (Node Type) *The* **type** *of a node is $o_i \in O$ or $type(o_i) \in \{$ task, compTask, dataStore, enviri, enviro, pertimer, shotimer $\}$.*

On its border, a node has zero or more input and/or output ports. The firing semantics of the ports determine a start point and exit point for each node.

Definition 5 (Entry Point) *The set of input ports that can fire to start the execution of a node is the entry point of an operational node $o_i \in O$, or $entry(o_i)$, and is at the start time (see Definition 35) t_i^s of the node.*

Definition 6 (Exit Point) *The exit point of an operational node $o_i \in O$, or $exit(o_i)$, indicates the end time t_i^e of the node and occurs when all computation and communication performed by that node is completed.*

Behavioural Nodes

Behavioural nodes $B_i \in O$ include task nodes, composite task nodes and data stores. These are nodes that consume processor resources to execute and, hence, have an execution duration associated with each of the nodes.

Task Nodes

Definition 7 (Task Node) *A task node performs one or more functions, has one or more input ports and has one or more output ports.*

The functions a task node perform are built either with the visual language or with a procedural language. Task nodes can be combined or decomposed given that the following condition is satisfied.

Definition 8 (Task Node Composition) *A* **task node** $B_i \in O$ *is a maximal set of connected operations with a deterministic execution duration* $\Lambda(B_i)$. *It may consist of one or more task nodes, composite task nodes and data stores.*

The deterministic execution requirement enables a separation of concerns. A task node can be statically compiled and partially ordered for execution, and can be described in most programming languages. The non-deterministic aspects of a system are, thus, raised to level visible for modelling and analysis, while the deterministic part is encapsulated.

Data Stores

Data stores represent significant persistent data, or data shared between two or more task nodes and accessed with bi-directional ports. Data stores are indicated with one or more bi-directional ports which provide access to them.

Definition 9 (Data Store) *A* **data store** d_k *is a persistent memory set with a different operation and reply message/token associated with each incoming message type and with a deterministic execution duration* $\Lambda(d_k)$.

Definition 10 (Data Store Access) *A data store is accessed through* **bi-directional ports** *(see Definition 22), only.*

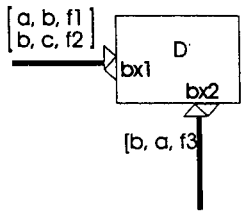
Referring to Figure 3.2 it can be seen that data stores are accessed with bi-directional ports. The protocol on a bi-directional port has two parts, the transmitted message and the return message.

A data store can implement any number of storage strategies. These could include circular buffers, queues, associative memory, simple registers or any more complicated database structure. A data store could even be a file system at an appropriate level of system decomposition. A data store thus represents a very flexible data modelling medium.

The execution duration of a behavioural node is the time it takes from entry until exit.

Definition 11 (Task Node Execution Duration (TNED)) *The* **task node execution duration (TNED)** $\Lambda : B_p \rightarrow R^+ \times R^+$ *is a time function associating an execution duration* $\Lambda(B_i) = [\delta(B_i), \Delta(B_i)]$ *with each task node* $B_i \in B_p$, *with* $[\delta(B_i)$ *and* $\Delta(B_i)]$ *being the minimum and maximum execution durations of the task node, respectively.*

The actual execution duration $\lambda(B_i)$ will vary from one execution instance to the next, where $\delta(B_i) \leq \lambda(B_i) \leq \Delta(B_i)$. The actual execution duration(s) can be derived from the underlying timed stategraph representation of the task node.




```
Task: D

Description: D stores the current satellite attitude matrix and
             the orbit orientation
Protocol definition:
INTEGER16; [1..8]REAL32 orientationMatrix;
INTEGER16; [1..3]REAL32 satAttitudeMatrix;
INTEGER16; size;

Port definitions:
(* portName (portType, msgRx, msgTx, inputPeriod) *)
D.bx1 (bidirectional, orientationMatrix, satAttitudeMatrix, wcetf1);
D.bx1 (bidirectional, satAttitudeMatrix, size, wcetf2);
D.bx2 (bidirectional, satAttitudeMatrix, orientationMatrix, wcetf3);

(* for each port - only shown for one port bx1 *)
Trigger event : Receive(D.bx1.msgRx(flags/fields of interest))
Assignments   : orientationMatrix := D.bx1.msgRx
```

Figure 3.2: A data store is a node with only bi-directional ports

A task node will continue execution until completion without any synchronisation with the environment or other task nodes, once it has been started. This makes the scheduling boundaries clear and allows a task thread to be written in a conventional language, if RDF or another data flow language is not used.

Tasks are represented by task nodes that are circular or rectangular to indicate tasks not being decomposed into other tasks, i.e. primitive tasks. Primitive tasks may be nested inside another task in which case they are collectively referenced with a composite task node. This composite task node reference symbol is used in the original diagram. The reference symbol is a rectangle with rounded corners. Both primitive and composite task nodes have input and output ports. See Figure 3.6 for a completely specified task node. The semantics of hierarchy is dealt with in Section 3.5.

Time Control Nodes

Behavioural nodes represent the functional behaviour inside a system and identify traditional code segments that could possibly run in parallel. Time control nodes model the interaction points at time instants, between the environment and the behavioural nodes. See Figure 3.3 for a set representation of the node types.

Periodic and single shot timers are provided in the architecture to express any time moment and time series succinctly. All timers have one output port which connects to the input port of one or more tasks in a particular RDF $\mathcal{G}$. A pre-defined task set in the RDF $\mathcal{G}$ can set or reset

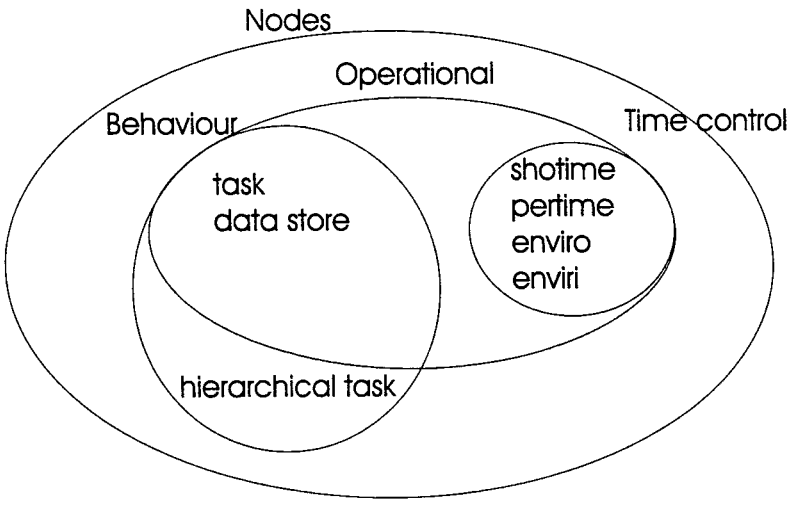


Figure 3.3: Node types and their classification

the timer.

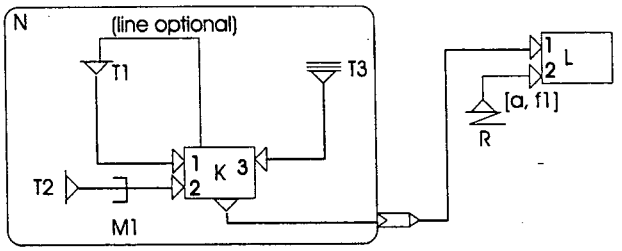
Definition 12 (Shotime Node) *A shotime node $o_{sho} \in O$ is an operational node with $type(o_{sho}) = \{shotime\}$ and $\Lambda(o_{sho}) = [0, 0]$. It has only an output port and indicates a possible starting point of an RDF $\mathcal{G}$. It has the following attributes $\{wctr(o_{sho}), period(o_{sho}), tasklist(o_{sho})\}$, where $wctr(o_{sho})$ indicates its worst case minimum inter-firing period, $period(o_{sho})$ indicates its period before firing, and $tasklist(o_{sho})$ is the task list that may set the timer.*

Periodic timers have only one period and one task which can adjust the period after initialisation. Periodic timers can serve as an input trigger source for a design graph. According to the rules of realisable design graphs (see Section 3.6.3), the system in Figure 3.4 is a valid graph as task K and task L do have a path from a trigger source. The full graphical description of a periodic and single shot timer is shown in Figure 3.4. The tasks that have access to set the timer are specified.

Definition 13 (Pertime Node) *A pertime node $o_{per} \in O$ is an operational node with $type(o_{per}) = \{pertime\}$, and $\Lambda(o_{per}) = [0, 0]$. It has only an output port and indicates a possible recurring starting point of an RDF $\mathcal{G}$. It has the following attributes $\{wctr(o_{per}), period(o_{per}), task(o_{per})\}$, where $wctr(o_{per})$ indicates its worst case minimum inter-firing period, $period(o_{per})$ indicates its period and $task(o_{per})$ is the task that may set the timer.*

Environment interfaces represent direct connections to the real world. This includes any interconnection to external timers, polled devices, interrupt devices and DMA devices. Environment interfaces represent the ultimate source and destination of messages. An input environment interface has only one output port, and the output rate is the worst case inter-arrival rate from the environment around the system. an output environment interface has only one input port, and the consumption rate is determined by the rate at which messages arrive at the interface. This rate must be less than or equal to the rate at which the environment can accept messages/tokens.

Definition 14 (Enviro Node) *An environmental output node $o_{enviro} \in O$ is an operational node with $type(o_{enviro}) = \{enviro\}$. It has only one input port and acts as a message injection*




```
Composite block: N

Description: N monitors three timers and sends a message to L
             to indicate certain timer combinations

Task nodes: K, L
Single shot timers: T1, T2
Periodic timers: T3
Environmental interface: R

Timer definitions:
T1 : Worst Case Trigger Rate; period; setting task list
T2 : WCTR; period; K, L;
T3 : 1000 micro seconds; L

Mailbox definitions:
M1 : No of entries, Size of one entry

Channel definitions:
T1.K, T2.K, T3.K, K.L, Rope.R.L

Interconnections:
T1.ox1 WITH T1.K TO K.ix1
T2.ox1 WITH T2.K VIA M1 TO K.ix2
T3.ox1 WITH T3.K TO K.ix3
K.ox1  WITH K.L TO L.ix1
R.ox1  WITH Rope.R.L TO L.ix2
```

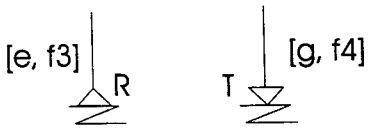
Figure 3.4: Channels, ropes and mailboxes are the glue which holds the tasks together

point from a system to its environment. It has the following attributes $\{triggerSource\}$*, where* $triggerSource$ *can be any one of* $\{interrupt, DMA, polled\}$.

Definition 15 (Enviri Node) *An environmental input node* $o_{enviri} \in O$ *is an operational node with* $type(o_{enviri}) = enviri$. *It has only one output port and acts as a message injection point from the environment. The output port's* production rate *attribute contains the maximum token production rate valid on the* enviri node. *The node has the following attributes* $\{triggerSource, actions\}$*, where* $triggerSource$ *can be any one of* $\{interrupt, DMA, polled\}$ *and actions are the message transmission actions.*

Referring to Figure 3.5 it can be seen that the input environmental interface emits messages of type $EVENT$ with an environment determined transmission rate of f_3 that is only dependent on the rate of external events.

When an enviro node's firing rule is true, it removes the specified number of tokens from its input




```
Enviri: R

Description: R is an asynchronous serial interface

Port definitions:
R.ox1.msg : E;              (* short for EVENT; *)
R.ox1.rate = f3;            (* environment dictated *)
R.ox1.noOfTokens = 1;       (* single token data flow *)
R.ox1.condition : true;     (* OR output port guard *)

(* for each enviri - only shown for enviri R *)
R.triggerSource : interrupt;
R.action : send(R.ox1.msg.e);
```

Figure 3.5: Environmental interface nodes and textual specification for environment interface R

port and emits them to the environment. Enviro and enviri nodes are typically implemented as device drivers in a software implementation.

The set of all environmental input nodes and environmental output nodes is denoted by O_{enviri} and O_{enviro}, respectively. The set of all time control nodes is denoted by $O_{timeControl}$:

$$O_{timeControl} = \{o_i \in O | type(o_i) \in \{shotime, pertime, enviri, enviro\}\}$$

3.2.3 Ports

The RDF $\mathcal{G}$ is a data flow graph and the execution of a node is triggered by a token whose contents has meaning. Such tokens are received and emitted on *data ports*. The data ports allow a task to interconnect through an edge that leads to ports connected to other tasks and time control nodes. The data ports are typed, follow an AND or OR trigger pattern and are unidirectional. Bi-directional ports connect only to synchronous channels and exhibit a synchronous activation trigger pattern.

Definition 16 (Data Port) *A* **data port** p_i *has the following attributes* $\{pr_i, rp_i, tk_i\}$, *which are respectively the* **message protocol**, **maximum rate of production/consumption**, *and* **number of tokens** *(messages) produced (output port) or consumed (input port) per execution cycle.*

Definition 17 (Input Data Port) *An* **input port** $ix_k \in V_o$ *of node* o_j *is connected by edge* $e_{i,j,k,n}$ *from node* o_i *and is the* $n'th$ *input data port of a node* $o_j \in O$ *such that* ix_n *transports the value on the edge* $e_{i,j,k,n}$ *to the internals of the operational node* o_j. *The input port* ix_n *is of type* $= \{or, and\}$.

Definition 18 (Output Data Port) *An* **output port** $ox_k \in V_o$ *between edge* $e_{j,n,k,m}$ *which interconnects node* o_j *with node* $o_n \in O$ *is the* $k'th$ *output data port of a node* $o_j \in O$ *such that*

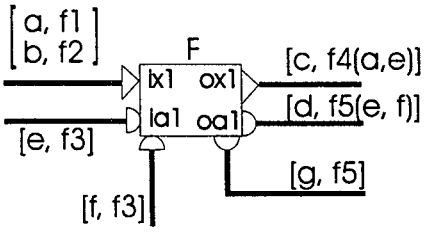
ox_k transports the value from the internals of the operational node to the edge $e_{j,n,k,m}$. The output port $ox_{k,j,n}$ is of type $= \{or, and\}$.

These input ports and output ports forms two sets of ports for a node.

Definition 19 (Input Port Set) *An* input port set *ipo_i for node o_i is $ipo_i = \exists o_i : \cup_{\forall j} ix_j$.*

Definition 20 (Output Port Set) *An* output port set *opo_i for node o_i is $opo_i = \exists o_i : \cup_{\forall j} ox_j$.*

Definition 21 (Composite Port) *A* composite port *p of an RDF $\mathcal{G}$ is a data port on the border of $\mathcal{G}$ which transports the value of p to or from the inside of $\mathcal{G}$. Here $V_h(\mathcal{G})$ denotes the set of all composite data ports of an RDF $\mathcal{G}$.*




```
Task: F

Description: F is the primary control_block

Port definitions:
(* vector of incoming message types, only one type at a time received *)
F.ix1.msg.1: CmdType;   (* abstract data structure *)
F.ix1.rate.1 = f1( processor, clock speed, architecture );
F.ix1.noOfTokens.1= 1;  (* single token consumption *)
F.ix1.type.1= or;       (* or input port *)

F.ix1.msg.2: int;
F.ix1.rate.2 = f2( processor, clock speed, architecture );
F.ix1.noOfTokens.2 = 1; (* single token consumption *)
F.ix1.type.2= or;       (* or input port *)

(*Port(direction, type, protocol, rate, noOfTokens)*)
F.ix2(in, and, float, sameRate, 1);
F.ix3(in, and, float, sameRate, 1);

F.ox1.msg.1: float; int[8];
F.ox1.rate.1 = (1/2 * F.ix1.rate.1) + 1/10 * F.ix2.sameRate);
F.ox1.noOfTokens.1= 1; (* single token production *)
F.ox1.type.1= or;      (* or output port *)
F.ox1.guard.1= true;   (* output port guard *)
```

Figure 3.6: Full RDF task node definition

A protocol specification is associated with each input and each output port. The format of the protocol specification is shown in Figure 3.6 for a completely specified task node with protocol specifications. The types are the data types supported by any strictly typed language. The maximum consumption rate of an input port is a function of the processor type, clock speed

and architecture of the hardware. The transmission rate function of an output port is linear in the input rates of the messages which trigger an output on the specific port. The rates are derived in Section 3.6.1. A $Rate = 0$ indicates that no rate is specified. The input port can accept any subset of message types sent to it via a channel from an output port of another task.

Bi-directional ports are used for task nodes to interact with data store nodes. A bi-directional port on a task node is defined as follows.

Definition 22 (Bi-directional Port) *A* **bi-directional port** $bx_k \in V_b$ *between edge* $s_{j,n,k,l}$ *which interconnects node* o_j *with data store* $d_n \in D$ *is the* $k'th$ *bi-directional port of a node* $o_j \in O$ *such that* bx_k *transports the value from the internals of the operational node to* $s_{j,n,k,l}$. *The port* bx_k *is of type* $= \{bidirectional\}$.

A bi-directional port bx_i *of node* o_j *has the following attributes* $\{pri_i, pro_i, rp_i\}$, *which are respectively the* **message protocol for a message produced, message protocol for a message consumed,** *and* **maximum rate of consumption** *per execution cycle.*

The bi-directional ports on a data store are each paired with the bi-directional ports on each of the task nodes connected to the data store.

Semantics

The set of input ports on a task can contain a number of OR ports and a number of AND ports. If the correct number of messages on any of the OR ports is available, the task executes the associated function. The correct number of messages must be received on each of the AND ports before the task starts executing. A set of AND ports implements the traditional data flow semantics [3], while the OR ports allow conditional expressions to be constructed in the data flow graphs.

3.2.4 Edges: Channels, Ropes and Mailboxes

Edges consist of asynchronous and synchronous edges. Asynchronous edges interconnect tasks so that the tasks can together perform a collective function.

Unsynchronised inter-node communication is performed via data stores to which one task node can write to in its execution cycle. The task node can even overwrite a previously written value if another task node has not retrieved the value in time. This is similar to communicating through registers.

An asynchronous edge $e_{i,j,k,n}$ between an output port of an operational node o_i and an input port of an operational node o_j enforces the precedence and data dependency between o_i and o_j.

Definition 23 (Asynchronous Edge) *An* **asynchronous edge** $e_{i,j,k,n} \in E$ *between the output port* p_k *of node* o_i *and the input port* p_n *of node* o_j *is an edge such that:*
$t^s(o_j) \geq t^e(o_i)$
The edge is of type$(e_{i,j,k,n})$ $= \{channel,\ mailbox,\ rope\}$ *and has an attribute* $\{count\}$ *which counts the number of tokens/messages sent on the edge.*

There are three different types of edges depending on modelling and implementation issues.

Together they are referred to as *asynchronous edges*. They are only differentiated into *channels*, *mailboxes* and *ropes* when additional knowledge about their behaviour is required. The form of communication they support differs according to whether messages are to be delivered timely (channel), whether messages are delivered with a possible burst at larger rate than the average rate (mailbox), or whether a timely indication-only is delivered which contains no more information than the fact that a transmission did take place (rope).

Definition 24 (Channel) *A* **channel** $ec_{i,j,k,n}$ *is an asynchronous edge* $e_{i,j,k,n}$ *between the output port* ox_k *of node* o_i *and the input port* ix_n *of node* o_j *with a storage capability of one token/message. It corresponds to the semantics of a homogeneous data flow system channel.*

A mailbox is a channel with a First In First Out (FIFO) buffer. Mailboxes are used to realise channels where the average transmission rate of an output port is known, but the peak rate is higher, or when there is an instantaneous rate difference between a producer and a consumer, but the average rate of production is known. The mailbox in Figure 3.4 is specified with the message size and the number of messages it can hold, as the source and destination information is implicitly contained in the diagram.

Definition 25 (Mailbox) *A* **mailbox** $em_{i,j,k,n}$ *is an asynchronous edge* $e_{i,j,k,n}$ *with storage capacity of size(m) tokens/messages. The* **size** *of a mailbox* $em_{i,j,k,n}$ *is a range* $[s_i..S_i]$ *with* $s_i, S_i \in N : s_i \leq S_i$*, indicating the storage capacity of the number of items in the mailbox;* $size(em) = S_i - s_i + 1$.

A rope is an asynchronous edge that cannot transfer any information. It can only deliver how many times it has been 'pulled' by the sender. It is useful for synchronisation functions and to indicate that an event did occur without incurring the message transmission cost of a channel. The environmental interface R is connected with a rope to task L in Figure 3.4.

Definition 26 (Rope) *A* **rope** $er_{i,j,k,n}$ *is an asynchronous edge* $er_{i,j,k,n}$ *between the output port* ox_k *of node* o_i *and the input port* ix_n *of node* o_j *with a storage capability of zero tokens/messages. The attribute* $\{count\}$ *is incremented on each pull of the rope* $er_{i,j,k,n}$.

The set of nodes executing before node o_i starts is defined as $pred(o_i)$. The set of nodes following node o_i is $succ(o_i)$.

Definition 27 (Predecessor and Successor Set) *For each node* $o_i \in O$,
$pred(o_i) = \{o_h \in O : \forall h \exists i e_{h,i,k,n}$ *and*
$succ(o_i) = \{o_j \in O : \exists i \forall j e_{i,j,k,n}$

denote the **predecessor** *and* **successor set**, *respectively.*

Synchronous edges interconnect the bi-directional ports of tasks to bi-directional ports on data stores. A synchronous edge has a storage capacity of one token in each direction and does not have any knowledge of the type of message it transmits. A synchronous edge $s_{i,k,j,n}$ between a bi-directional port dx_j of an operational node o_i and a bi-directional port dx_n of a data store d_k enforces that o_i must block on transmission and wait for a response message (token) from the data store d_k before continuing with execution.

Definition 28 (Synchronous Edge) *A* **Synchronous edge** $s_{i,k,j,n} \in S$ *between nodes* o_i *and* d_k *is an edge such that:*

1. $t^s(d_k) \geq t^s(o_i)$
2. $t^e(d_k) \leq t^e(o_i)$
3. $\Lambda(o_i)' = \Lambda(o_i) + \Lambda(d_k)$

3.3 RDF Operational Semantics

The system dynamics are required in addition to the structural information expressed by graph nodes, data ports and edges. The operational dynamics correspond to a *data flow* paradigm augmented with concepts of *elapsed time* and a *token state space*. The data flow paradigm operates on the number of tokens present at the respective input ports of a node. These tokens (messages) are carried on the asynchronous edges and, given the semantics of a mailbox, there could be one or more stored on each edge.

3.3.1 Semantics

An edge carries $\mu_{i,j}$ tokens from node o_i to node o_j and is the shorthand for $\mu(e_{i,j,k,n})$. For a channel holds $\mu_{i,j} = \{0, 1\}$, and for a mailbox $\mu_{i,j} \leq size(m_{i,j})$. The number of tokens on an edge are the number of messages in a mailbox. A rope holds $\mu_{i,j} \leq k \in N^+$, and it consumes the memory for one counter. The complete token state of an RDF graph can be described as the number of tokens on each of the edges in the data flow graph. The token state of an RDF graph $\mathcal{G}$ is useful to keep track of changes in the number of tokens on the edges, due to the firing of nodes.

Definition 29 (RDF Token State) *A* **token state** μ *of an RDF* $\mathcal{G}$ *is a mapping* $\mu : E \rightarrow N^+$.

The number of tokens on each edge, is a bounded number per edge. The semantics of the OR input and AND input ports can now be taken into account to describe when a node is *enabled* to fire.

Definition 30 (Multiple Port Firing Condition) *An operational node* o_k *can only start execution iff:*
1. $\forall ix_i \in ipo_k : \mu_{j,k} \geq ix.tk_i$, *if* $type(ix_i) = and$
or
2. $\exists ix_i \in ipo_k : \mu_{j,k} \geq ix.tk_i$, *if* $type(ix_i) = or$
The choice of which port fires if multiple OR ports are enabled is based on the input port with the earliest deadline first.

The execution rule for an RDF graph $\mathcal{G}$ can now be composed:

Definition 31 (Execution Rule) *Firing of an enabled node* $o_j \in O$ *in token state* μ *at time* t^s *results in the following phases with the associated token state for each phase:*

1. **Firing**: *at time* t^s *the number of* tk_n *tokens are consumed from input edge* $e_{i,j,m,n}$ *resulting in:*

$$\mu'_{i,j} = \begin{cases} \mu_{i,j} & if e_{i,j,m,n} \in E \setminus pred(o_j) \\ \mu_{i,j} - tk_n & if e_{i,j,m,n} \in pred(o_j) \forall n : ix_n = and \wedge \mu_{i,j} \geq tk_n \\ \mu_{i,j} - tk_n & if e_{i,j,m,n} \in pred(o_j) \exists n : ix_n = or \wedge \mu_{i,j} \geq tk_n \end{cases}$$

2. **Execution**: *takes* $\lambda(o_j)$ *time, with* $\delta(o_j) \leq \lambda(o_j) \leq \Delta(o_j)$, *and does not result in an RDF token state change.*

$$\mu'_{i,j} = \mu'_{i,j} : \forall t \in [t^s, t^s + \lambda(o_i)]$$

During execution, the **elapsed execution duration** *of* o_j, $\mathcal{EED}(o_j, t)$ *is equal to* $t - t^s$.

3. **Completion**: *until time* $t^e = t^s + \lambda(o_j)$, *tokens are produced on output edges. The tokens are only 'visible' on completion of node* o_j *which means an atomic action with duration* $\lambda(o_j)$ *has just taken place. The resulting RDF token state is then:*

$$\mu_{j,m} = \begin{cases} \mu_{j,m} & if e_{j,m,k,p} \in E \setminus post(o_j) \\ \mu_{j,m} + tk_k & if e_{j,m,k,p} \in post(o_j) \end{cases}$$

The start point t^s is defined in Definition 5, and an end point t^e is defined in Definition 6 for a node o_j. In between is the *elapsed firing time* (EFT) which linearly increases from 0 to $\lambda(o_j)$.

Definition 32 (State of $\mathcal{RDF}$) *The state* $\mathcal{S}$ *of an RDF* G *consists of:*

1. *the RDF token state,*
2. *the Elapsed Execution Duration* $\mathcal{EED}$, *and*
3. *the RDF control state (RDF control state is defined later, essentially it is a state vector representing all the parallel states in which the stategraph is in).*

At any time instance t, an RDF can be fully characterised by its state $\mathcal{S}$. The transition from one RDF token state to another forms an execution sequence leading from one RDF token state to the next.

3.3.2 Output Port Guards

The semantics of the output ports require a further investigation. The semantics of the input ports have been described in detail and led to the definition of the execution rule in Definition 31.

Formal Definition

Each output port has a condition that must be *true* before the messages are emitted on that port.

Definition 33 (Output Port Guard) *The* **output port guard** $g_k = guard(ox_k)$ *of an output port* ox_k *is a function* g *on port* k *of node* j *which generates a Boolean result* $\{true, false\}$, *and is an expression which can include the value of one or more input tokens or retrieved data store values. If the Boolean result is true, the tokens/messages available for transmission on that port are emitted.*

The values of the input tokens evaluated as part of the guard expression are the values received in that particular execution sequence of the task node. This means that the task node cannot store state, it can, however, store and share state information through a data store.

Operational Semantics

The execution rule is amended to include the restrictions implied by the Boolean conditions (guards) on the output ports.

Definition 34 (Extended Execution Rule) *Firing of an enabled node $o_j \in O$ in token state μ at time t^s results in the following phases with the associated token state for each phase:*

1. **Firing**: *at time t^s the number of tk_j tokens is consumed from input edge $e_{i,j,m,n}$ resulting in:*

$$\mu'_{i,j} = \begin{cases} \mu_{i,j} & if\ e_{i,j,m,n} \in E \setminus pred(o_j) \\ \mu_{i,j} - tk_n & if\ e_{i,j,m,n} \in pred(o_j) \vee \forall n : type(ix_n) = and \wedge \mu_{i,j} \geq tk_n \\ \mu_{i,j} - tk_n & if\ e_{i,j,m,n} \in pred(o_j) \vee \exists n : type(ix_n) = or \wedge \mu_{i,j} \geq tk_n \end{cases}$$

2. **Execution**: *takes $\lambda(o_j)$ time, with $\delta(o_j) \leq \lambda(o_j) \leq \Delta(o_j)$, and does not result in an RDF token state change.*

$$\mu'_{i,j} = \mu'_{i,j} : \forall t \in [t^s, t^s + \lambda_j)]$$

During execution, the **elapsed execution duration** *of o_i, $\mathcal{EED}(o_i, t)$ is equal to $t - t^s$.*

3. **Completion**: *until time $t^e = t^s + \lambda(o_j)$, tokens are produced on output edges. The tokens are only 'visible' on completion of node o_j. The resulting RDF token state changes on the output edges are:*

$$\mu_{j,m} = \begin{cases} \mu_{j,m} & if\ e_{j,m,k,l} \in E \setminus post(o_j) \\ \mu_{j,m} + tk_k & if\ e_{j,m,k,l} \in post(o_j) \wedge \forall k : type(ox_k) = and \wedge g_k = true \\ or & \\ \mu_{j,m} + tk_k & if\ e_{j,m,k,l} \in post(o_j) / \exists k : type(ox_k) = or \wedge g_k = true \end{cases}$$

The output port now only produces the said number of tokens/messages if the guard (Boolean condition) evaluates to true. The port chosen is the first one whose guard evaluates to true. If more than one output port is enabled then, in principle, any of the enabled ports are selected to fire. In practice, a next port algorithm can be implemented to ensure a fair distribution of firing. For a set of AND output ports, all the ports must have the same condition as they all fire on the same condition. The guarded output ports can be used to model the traditional programming constructs such as if statement and conditional loop.

3.4 Temporal Properties

This section defines a number of temporal properties in general and then continues to define the temporal properties for a behavioural node B_i, a time control node o_i and an RDF graph $\mathcal{G}$.

1. **Definition**

   Real-Time: 1. Pertaining to the actual time during which a physical process transpires. 2. Pertaining to the performance of a computation during the actual time that the related physical process transpires in order that results of the computation can be used in guiding the physical process.
   [IEEE Dictionary of Electrical and Electronic Terms]

2. **Accuracy** The accuracy of a time value relates to how close to the actual value the estimated value is.
   (a) A time property can be represented by a **scalar** value, e.g. $\lambda(o_i)$.
   (b) A time **interval** might describe a particular situation more accurately, e.g. $[\delta(o_i), \Delta(o_i)]$.
3. **Time constraints** The limits imposed upon the application by the physical process to guide.
   (a) **Duration** is the amount of time a computation takes associated with two particular points in a system. Usually it is the time between the start and the end of task execution, and is expressed as $\lambda(o_i) = t^e - t^s$.
   (b) **Rate** refers to how many times per time unit something is done. This is of particular interest in the interaction with a physical process.
4. **Functional time**; the time that is transpiring in the system as implemented.
   (a) The **duration** is best measured and represented as an interval of minimum and maximum execution time, viz. $[\delta(o_i), \Delta(o_i)]$.
   (b) The worst case **rate** at which a task is executed is best represented by a scalar value, viz. rp_i.
5. **Temporal properties** are properties inherent to time.
   (a) The **inverse** of duration (period) is rate through the function $1/rate = duration$. This property is useful, because it is convenient to measure the execution **duration** of a task and to work with the **rate** at which a system can handle events.
   (b) **Jitter** is a known bound on the accuracy of a scalar value of time with an unknown instantaneous value.

The functional time properties of an RDF graph are those properties characteristic of the implemented system. A behavioural node was defined as a connected set of operations with a deterministic execution duration. The execution duration for a number of task nodes with deterministic execution time can be summed as follow $o_j = \sum_{i=0}^{k} SED_i$. The messages that trigger the execution of the behavioural node are available before the node is triggered to execute. The messages produced are written into each of their respective ports during execution of the behavioural node. The messages are, thus, available on the asynchronous edges *after* execution of the behavioural node, and the next behavioural node to execute can be scheduled (assuming a single processor).

Definition 35 (Start Time and End Time) *The* **start time** $t^s(o_i)$ *of an enabled behavioural node* $o_i \in O$ *is defined as the moment that the trigger rules fires and the nodes start execution* : $t^s(o_i) \in N^+$.

The **end time** $t^e(o_i)$ *of a behavioural node* $o_i \in O$ *is defined as the moment that the node* o_i *completes execution* : $t^e(o_i) \in N^+$.

Time is expressed as a natural number that denotes any granularity of a basic cycle in which time is measured. The basic cycle could also represent processor cycles. A node fires as soon as its firing condition is satisfied. This corresponds to as-soon-as-possible firing and uses all parallelism as expressed in the model.

The functional instantaneous execution rate is a useful parameter to compare with the rate at which a node is to execute based on the RDF analysis that is described later.

Definition 36 (Instantaneous Execution Rate) *The instantaneous execution rate $R_i(t)$ of an operational node o_i at time instant t is the number of executions n per time:*

$$R_i(t) = dn/dt = \lim_{\Delta t \to 0} \Delta n/\Delta t$$

Given that $n \in N^+$ is discrete:

$$R_i(t) = \frac{1}{t^s_{k+1}(o_i) - t^s_k(o_i)}$$

with k denoting the execution cycle index and $t^s_k(o_i)$ the start time of the $k'th$ invocation of o_i.

Time control nodes have a specific timing semantics. The timers and environmental interfaces can exhibit *periodic* or *aperiodic* occurrence patterns. In both cases, the start time and end time of an occurrence is instantaneous, as all execution duration is accounted for in behavioural nodes. In particular:

Definition 37 (Event Occurrence Instant) *For every node $o_i \in O_{timeControl}$ holds: $(t^s_k(o_i)) = (t^e_k(o_i))$, which is represented by a single symbol t_k — the $k'th$* **occurrence instant** *of the event.*

An event is **periodic** if $\forall k : t_{k+1} - t_k = period$ with *period* the constant inter-occurrence time. An event is **sporadic** if $\forall k : t_{k+1} - t_k \geq mintime$ where *mintime* is the **minimum inter-arrival time** between any two event occurrences.

The execution duration for an RDF $\mathcal{G}$ is the time from the start of the first operational node to the finish of the last operational node in RDF $\mathcal{G}$.

Definition 38 (Execution duration of an RDF $\mathcal{G}$) *The* **graph execution duration** *of an RDF $\mathcal{G}$ is $\Delta(\mathcal{G}) = max(t^e) - min(t^s) \in N^+$.*

3.5 Hierarchy

The ability to hide the detail not required at any stage of the development of a complex system is essential to aid in simplifying and, hence, understanding such a system. The RDF language was designed with hierarchical application in mind. The stategraph language with its Statechart heritage, is inherently hierarchical.

Complex systems can only be understood and constructed if they are decomposed into smaller parts and abstracted into the relevant features at each level of decomposition. A decomposed system consists of hierarchical layers representing increasing knowledge about the system as one goes lower in the layers. Two forms of hierarchies exist, i.e. vertical hierarchies and horizontal ones. Vertical hierarchical decomposition corresponds to black box decomposition and is defined in Definition 39. Horizontal hierarchical corresponds to white box decomposition where all aspects of the replaced component are visible and is defined in Definition 40.

Definition 39 (Vertical Hierarchical Decomposition) *Vertical hierarchical decomposition is given when a composite block can be viewed as a black box, with sufficient information available to validate its function in a system without knowing anything more about its inner functioning.*

Definition 40 (Horizontal Hierarchical Decomposition) *Horizontal hierarchical decomposition occurs when the whole system representation must be flattened to primitive blocks to validate the function of a complete system.*

3.5.1 RDF Hierarchy

The visual RDF notation has been described in great detail in Section 3.2. An RDF graph $\mathcal{G}$ can contain composite task nodes that are placeholders in the RDF $\mathcal{G}$ for another RDF $\mathcal{G}'$ which can be substituted into the composite task node. The task nodes and composite task nodes form the basis of the inter-relationship between RDF and the object oriented programming view which is used to organise and manage RDF components. Each composite task node and task node has an associated stategraph which provides information about timing and message sequence processing, and is an aid in the design refinement process. A system can be decomposed into a hierarchy in one of two ways. Vertical decomposition is the preferred mechanism as system trade offs are possible at any point in the design refinement process. This section will develop the hierarchical composition of systems with the RDF notation.

Composite task nodes have input and output ports allowing the nodes to communicate with other task nodes. The ports on composite task nodes support messages of multiple protocols. On refinement of a composite task node the different messages can be accepted by different tasks inside the composite task node.

Composite Task Nodes

Definition 41 (Composite Task Node) *A composite task node is a behavioural node* $o_i \in B^h$ *with:*

1. $type(o_i) = compTask$,
2. $rdf(o_i) = \mathcal{G}'$, *i.e. it instantiates a proper RDF which terminates with no messages left on edges and the only state information stored being kept in data stores, and*
3. *execution duration* $\Lambda(o_i) = \Delta(\mathcal{G}')$

The execution of a composite task node has to be triggered from an environmental interface or timer and, thus, either should contain an environmental interface, or it is assumed that the composite task node will be activated through its incoming data ports which, when traced back, will eventually find an environmental trigger.

3.5.2 Operational Semantics

A composite task node can have any number of input ports. The firing rule of Definition 43 is valid to start the trigger for a combination of AND ports and OR ports that leads into the composite task node.

Definition 42 (Composite Input Ports) *A* **composite input port** *of an RDF* $\mathcal{G}$ *is the* $i-th$ *composite input port of node* o_j. *It is called* cix_i *as a data port and cannot have a guard as defined in Definition 33.*

Definition 43 (Composition Firing Rule) *The* composite task node *fires, if the following holds on its composite input ports* $\forall i : type(cix_i) = \{in\}$:

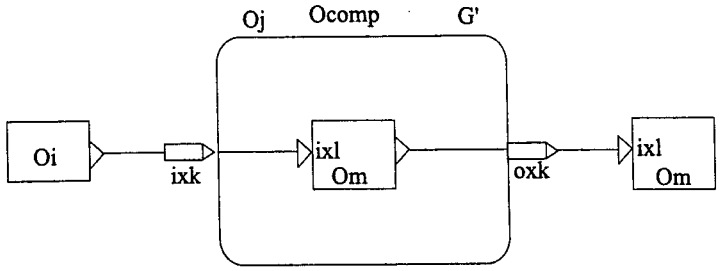


Figure 3.7: The composite task node o_j is an RDF $\mathcal{G}'$

A composite node o_k can only start execution iff:
1. $\forall cix_i \in cipo_k : \mu_{j,k} \geq cix.tk_i$, *if* $type(cix_i) = and$
or
2. $\exists cix_i \in cipo_k : \mu_{j,k} \geq cix.tk_i$, *if* $type(cix_i) = or$

The choice of which composite port fires if multiple OR ports are enabled is the port with the earliest deadline.

With the firing rule for composite ports established, the execution rule for a composite task node can be formulated. The internals of a composite task node $o_j \in O$ of an RDF $\mathcal{G}$ is also a valid RDF $\mathcal{G}'$; see Figure 3.7. This leaves only the initialisation and termination of the RDF $\mathcal{G}'$ and the handover of tokens associated with the two phases to be described.

Definition 44 (Compositional Execution Rule) *Firing of an enabled composition node $o_{comp} \in O$ with $rdf(o_{comp}) = \mathcal{G}'$ in token state $\mu(\mathcal{G})$ at time t^s results in the following phases with the associated token state for each phase:*

1. **Firing of o_{comp} and initialisation of RDF $\mathcal{G}'$:** *at time t^s the number of tk_n tokens is consumed from input edge $e_{i,j,m,n}$ resulting in:*

$$\mu'_{i,j}(\mathcal{G}) = \begin{cases} \mu_{i,j}(\mathcal{G}) & if\ e_{i,j,m,n} \in E \setminus pred(o_j) \\ \mu_{i,j}(\mathcal{G}) - tk_n & if\ e_{i,j,m,n} \in pred(o_j) \wedge \forall n : type(cix_n) = and \wedge \mu_{i,j} \geq tk_n \\ \mu_{i,j}(\mathcal{G}) - tk_n & if\ e_{i,j,m,n} \in pred(o_j) \wedge \exists n : type(cix_n) = or \wedge \mu_{i,j} \geq tk_n \end{cases}$$

and generate tokens in underlying RDF ($\mathcal{G}'$)

$$\mu_{j,m}(\mathcal{G}') = \begin{cases} \mu_{j,m}(\mathcal{G}') & if\ e_{j,m,k,l} \in E \setminus (pred(o_j) \cup post(o_j)) \\ \mu_{j,m}(\mathcal{G}') + tk_k & if\ e_{j,m,k,l} \in post(o_j) \wedge \forall i : type(cix_i) = and \\ or & \\ \mu_{j,m}(\mathcal{G}') + tk_k & if\ e_{j,m,k,l} \in post(o_j) \wedge \exists i : type(cix_i) = or \end{cases}$$

2. **Execution** *of RDF $\mathcal{G}'$: takes $\Delta(o_{comp})$ time and does not result in an RDF $\mathcal{G}$ token state change.*

$$\mu'_{i,j}\mathcal{G}' = \mu'_{i,j} : \forall t \in [t^s, t^s + \lambda(o_comp)]$$

After initialisation, the token state for RDF $\mathcal{G}'$ changes according to the execution rule Definition 34.

3. **Termination** *of RDF $\mathcal{G}'$ and the* **completion** *of o_{comp}: at time $t^e = t^s + \Delta(\mathcal{G}')$ tokens are produced on the output edges of RDF $\mathcal{G}'$. The tokens are transferred to the output*

edges of o_{comp} and RDF $\mathcal{G}'$ is terminated. The resulting RDF token state changes on the output edges of o_{comp} are:

$$\mu_{j,m} = \begin{cases} \mu_{j,m} & if\ e_{j,m,k,l} \in E \setminus (pred(o_j) \cup post(o_j)) \\ \mu_{j,m} + tk_k & if\ e_{j,m,k,l} \in post(o_j) \wedge \forall i : type(ix_i) = and \\ or & \\ \mu_{j,m} + tk_k & if\ e_{j,m,k,l} \in post(o_j) \wedge \exists i : type(ix_i) = or \end{cases}$$

The node o_{comp} completes with only the AND output port set firing or one of the OR output ports.

The number of ports visible on the high level can be much less than the number of ports implemented in the composite task node. The ports visible on the high level are shown in the decomposed level as **composition input ports** and **composition output ports**.

Definition 45 (Port Composition Substitution) *When replacing a composition task node with its internal representation, the following substitution rules have to be observed.*

1. *Composite input ports:*
   (a) *The* **destination** *ports inside the RDF must be of the same type as the composite input ports.*
   (b) *The* **source** *ports outside the RDF connected with asynchronous edges to the composite input ports on the border of the RDF may be of any type $= \{or, and\}$.*
2. *Composite output ports:*
   (a) *Composite AND output ports must all originate from the same task node in the underlying RDF.*
   (b) *Composite OR output ports can originate from different task nodes in the underlying RDF, and the set of OR composite output ports must be structured in such a way that only one produces tokens on the termination of the RDF.*
   (c) *The* **source** *port inside the RDF must be the same type as the composite output port to which it is attached.*
   (d) *The* **destination** *port outside the RDF connected to from any composite output port can be of type $= \{and, or\}$.*

Multiple Channels

Multiple input and output channels are often required to model/describe an actual system. A task that has multiple input ports can be viewed in two ways as shown in Figure 3.8 a and b. The sender's view is outputting to a single port, connected by a single channel leading to a single input port connected on task M. The receiver can view messages as arriving on a single logical channel which simplifies the timing analysis of the system. See task M' in Figure 3.8 b. A single port leading to multiple output channels can be represented by the number of AND ports outputting the same message at the same rate to the destinations via individual channels. See Figure 3.8 c and d for an example where task Q and task Q' are equivalent. The example is shown for an OR output port, but is valid for an AND output port, too.

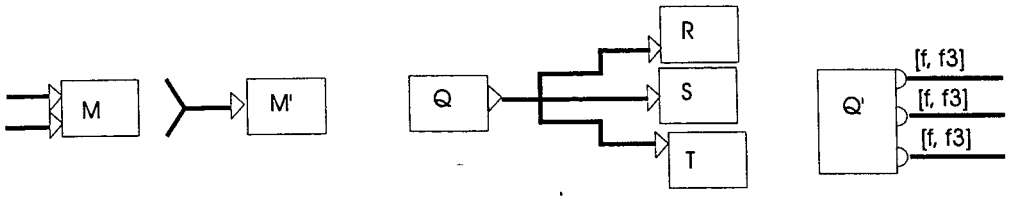


Figure 3.8: Representing multiple ports and channels

3.6 Message Passing Semantics — Timing

An interconnected set of operational nodes is functioning correctly in the real-time data flow time domain iff: every message (token) that is produced by an operational node is consumed before the next message (token) is emitted. For the interconnected system of message producers and consumers to function correctly, each pair of communicating objects must obey the requirement that the receiver must consume the incoming messages at least as fast as they are produced by the sender. This corresponds to an infinitely fast receiver as seen from the transmitter.

In practice a fast enough receiver is all that is required. Each input port has a maximum message consumption rate that is a function of the different types of hardware resources which can execute the task when triggered with this message on the input port. The consumption rate function is, thus, a function of processor type, clock speed, and architecture of the hardware. The maximum consumption rate of an input port can only be computed in the implementation phase of a project, and is discussed further in Chapter 4. The actual input rate is determined when the task is connected in a larger system from the output rate of the task connected to the input port. This actual input rate must be verified against the maximum consumption rate of the receiving port.

For scheduling a set of tasks on a single processor, another step has to be performed where the feasibility of the task set to execute on one processor is determined. This is an implementation issue and is covered in Chapter 4. In this section we determine the real-time behaviour of tasks symbolically in terms of output rates. In the next section we show how the rates are derived. For timing analysis it is assumed that the messages can be consumed fast enough. The maximum consumption rate for each input port is computed in the implementation phase and enough resources are allocated to each task to satisfy the rate at which it must consume messages.

3.6.1 Message Transmission Rates

For each output port we define the message transmission rate in terms of the worst case minimum inter-message transmission time; see Figure 3.9.

Definition 46 (Worst Case Minimum Inter-message Transmission Period) *If A and B are two interconnected tasks and P_B is the shortest inter-transmission period of messages emitted by task B, then the message transmission rate r on the output port of task B is*

$$r = 1/P_B$$

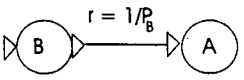


Figure 3.9: Two interconnected tasks with a channel transmission rate function

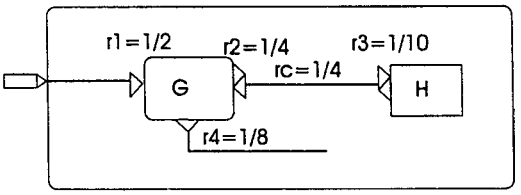


Figure 3.10: Use of a bi-directional port

The time basis is built on discrete time which means that P_B is a positive integer. It is then assumed that there exists a basic indivisible time unit which is small enough that the values of all time related constants and variables can be directly expressed or reasonably approximated by integer multiples of this unit.

Unidirectional OR Ports and AND Ports

Each OR input port on a task node has an individual rate. All AND input ports on a single task node must have the same rate. All AND output ports related to a simultaneous output of a result must have the same rate. This ensures that it is possible to abstract all AND ports in a set as one OR port for timing analysis. The OR input ports and the equivalent AND input port together from the independent input port set.

Definition 47 (Independent Input Port Set) *The* **independent input port set** *of node o_i is the set ipi_i:*

$$ipi_i = \exists o_i : \cup_{\forall j} ix_j \; for \; type(ix_j) = or \cup \exists ix_j : type(ix_j) = and$$

The AND input port set would correspond to one equivalent OR port for timing analysis and function analysis purposes. Likewise the AND output ports are grouped together based on the same output rate.

A channel connects an output port on one task to one or more input ports of different tasks. The channel derives its actual transmission rate from the sender. Referring to Figure 3.10 it can be seen that the channel rate function $rc = 1/4$ is dominated by the sender transmission rate function.

Bi-directional Ports

Before a task can complete processing a message, it must wait for a response from each bi-directional port to which a message was sent out. The transmission rate function of the return port of a bi-directional port is by definition $f(r) = r$, because for every incoming message a return message must be produced. The physical consumption rate of a bi-directional port on a data store is a function of the available resources and can be computed in the implementation

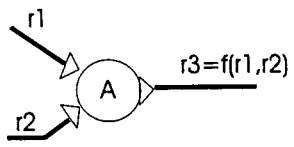


Figure 3.11: Computing an output rate

phase. The receiver on the other side of a bi-directional port must process messages and return results at a faster rate than the output rate on the bi-directional port of the sender. See Figure 3.10 for an example. Task G receive messages at a rate $1/2$ and for every two messages it receives it makes one request to data store H.

3.6.2 Computing Rates

This section describes how to compute the rate at which messages are emitted from an output port for homogeneous data flow. The synchronous data flow extension is described in Section 3.7. This section assumes that the maximum consumption rate on the input ports is always greater than the transmission rate.

Message Output Rates

The rate at which a task outputs messages on a specific port is a function of the rate at which it inputs on specific input ports. A transmission rate function is defined for each output port in terms of the input rates of all its corresponding input ports. Referring to Figure 3.11 it is shown that the output rate is a function of two input rates.

If the transmission rates are defined in terms of the worst case minimum message inter-arrival time, then for the computational model described so far in [52] it was proven that this function is always a simple linear function through the origin whose slope is a harmonic number. This function expresses the fact that messages cannot be sent faster than they are received.

Definition 48 (Transmission Rate Function) *The transmission rate function is of the form*

$$f(r_k) = \frac{1}{x} r_k,\ x \in \mathcal{N}$$

where r_k is the incoming rate on port k and x is the number of messages consumed before a message is emitted.

The logic of a task is evaluated to determine the transmission rate on a specific port. The logic is described with a stategraph on the same level of decomposition as the hierarchical decomposition of the data flow description. The stategraph contains all events/messages relevant to that level. All states on this level of decomposition are searched for message transmission states. The closest states are computed and the minimum of this distance will be the denominator of the slope in the transmission rate function.

The requirements on transmission rates for OR ports and AND ports are different. The message transmission rates on the set of AND output ports on a specific task are all the same. The message transmission rates for each OR output port on a specific task can be different. The

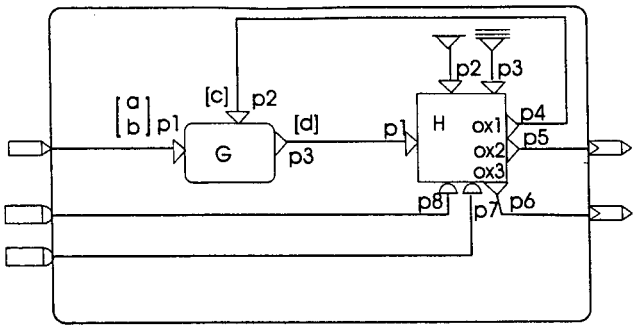


Figure 3.12: The output port rate functions p4, p5 and p6 on ports $\{H.ox_1, H.ox_2, H.ox_3\}$

```
Task: H

Partial port definition:
H.ox1.msg : STRING10;
  H.ox1.rate = 1/4 * H.ix1.rate;
H.ox2.msg : FLOAT64;
  H.ox2.rate = f(H.ix1.rate, H.ix4.rate)
```

output port transmission rate can be a function of any combination of OR input ports and the set of AND input ports. See the output port rate specification for p4 and p5 in Figure 3.12. Each AND input port must have the same message arrival rate. The transmission rate function is generally given as:

$$r_{out} = f(ix_i, ia_j)\ \forall i : 1 \leq i \leq n\ \wedge \exists j : 1 \leq j \leq m$$

where ix_i is the $i-th$ OR input port rate of n OR input ports and ia_j is the j_{th} AND input port set rate of the AND input port set.

Message Input Rates

We distinguish between the maximum consumption rate and the actual input rate as dictated by a sender. The maximum consumption rate is the rate at which messages on a port can be consumed. This rate depends on the algorithmic complexity of the function executed and the speed of the executing processor. The actual message input rate is the sender rate that could be smaller or equal to the maximum consumption rate of the receiver. For the purposes of timing analyses and design, the actual message rate is used. For a task with a single input port, the actual input rate is the known rate at which the task is supplied to process messages on that physical port. For a task with multiple input ports we consider a logical input port. The actual logical input rate is the worst case aggregate rate of all incoming messages on the independent input port set, see Definition 47. This is illustrated in Figure 3.13.

Definition 49 (Logical Input Port) *The* **logical input port** *of a task is a virtual port for which the input rate is the sum of the rates rp_i of all n independent input ports ix_i of node o_j*

$$r_{in} = \sum_{i=1}^{n} r_i$$

Two tasks can send messages at the same time to a receiving process implying a reception rate of infinity. However, it is only on the physical ports where the receiver's consumption rate must be greater or equal to the sender's rate. In practice each asynchronous edge is one or more buffers deep, which means that all incoming messages can arrive at the same time, but trigger the corresponding task node in an orderly manner.

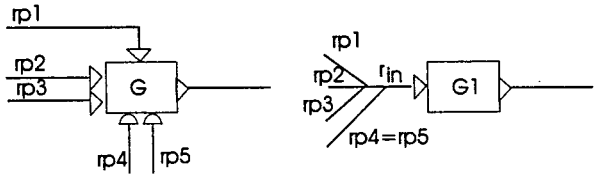


Figure 3.13: Input rate definition for multiple input ports

Multiple AND ports require a slightly different approach. All AND input ports on a specific task must have a message waiting before the processing of these messages take place. This means that the AND input ports must all have the same consumption rate to exclude one input port from receiving more messages than the other AND ports on the same task. The worst case consumption rate for any number of AND input ports in an AND port set is the same regardless of how many AND input ports are connected in the set.

3.6.3 Realising Designed Data Flow Networks

A design graph is realisable if all the channel rate equations can be solved. In order to ensure that the channel rate equations can be solved, the following design rules must be followed. The rules are based on those derived in [52] and are extended to allow for OR ports and AND ports.

Definition 50 (RDF Graph Design Rules) *The rules are as follow.*

1. *A task may have any number of unidirectional/bi-directional input/output ports. No two unidirectional OR output ports of a task may have the same receiver. Two AND output ports of a task may have the same receiver, receiving on two AND input ports as the semantics of the AND output ports ensure that no messages can be received other than at exactly the same rate and phase, i.e. at the same time instant.*
2. *There must exist a path from an active environmental interface or periodic timer to each task in the system. This requirement stems from the fact that the whole system is a reactive data flow system. This means that if a task cannot be reached from an execution trigger, it will never execute.*
3. *At most one message may be sent during the processing of an incoming message on each unidirectional output port. This preserves the semantics required to solve for the transmission rates on a designed system.*
4. *If the design graph is acyclic and adheres to requirements 1, 2 and 3, then in [52] it was proved that it is always possible to solve for the transmission rate equations and label each unidirectional channel in the design with a worst case transmission rate.*
5. *The transmission rate equations can be solved in a disjoint cycle if at least one channel in the cycle has a non-identity transmission rate function. That is, some task in the cycle must always delay for at least two messages before emitting a message. Formally it reads:*

Let G be a designed graph with a cycle C of n distinct tasks and unidirectional channels. Let $1/x_1, 1/x_2, \ldots, 1/x_n$ be the slopes of the transmission rate equations for the channels n C. If C is disjoint from other cycles in the graph, then the transmission rate equations for the channels in C can be solved if and only if

$$\prod_{i=1}^{n} x_i > 1$$

6. *The transmission rates can be determined for simple non-disjoint cycles in a design. A simple cycle is a cycle in which all the vertices are disjoint. The condition is that each cycle of two non-disjoint cycles must have at least one channel with a non-identity transmission rate function each. That is, some task in each cycle must always delay for at least two messages before emitting a message. In addition, there must be a task or tasks that have a cumulative delay for at least three messages.*

   Let G be a design graph with non-disjoint cycles. If every pair of cycles in G contains at most one task that receives messages from 2 distinct tasks in the 2 cycles, then the condition in the previous paragraph is sufficient for solving transmission rate equations. If the condition does not hold, then a set of simultaneous linear equations must be solved to determine the transmission rate functions of a design graph with non-disjoint cycles.

3.7 Synchronous Data Flow Extension

The constraints on the transmission rate function form in Definition 48 are lifted to include multi-token production and multi-token consumption per execution cycle.

Theorem 1 (Extended Transmission Rate Function) *The* **extended transmission rate function** *$f_k(r_j)$ of port ox_k of node o_i is a function of the number of tokens produced $y_k = tk_k$ by the output port k of node o_i and the number of tokens consumed $x_j = tk_j$ by the input port j of node o_i:*

$$f_k(r_j) = \frac{y_k}{x_j} r_j$$

where r_j is the rate at which tokens arrive at node i.

Proof 1 *Let o_i and o_m be two nodes connected to a channel. Let r_j be the worst rate at which node i receives messages. Thus node o_i receives a message every $1/r_j$ time units. Node o_i can only emit one or more messages y_k when it receives a message, or after it has received x_j messages. Message emissions are discrete events, they either occur or not. In the worst case, the minimum time separation between messages from node o_i to node o_m will be a fractional constant x_j/y_k times $1/r_j$. Therefore, the worst case transmission rate is $(y_k/x_j)r_k$ with y_k and x_j both positive integers.*

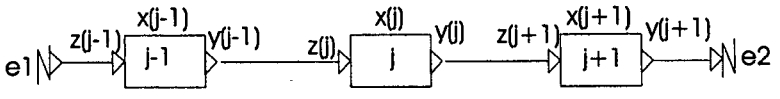


Figure 3.14: Multiple token production and consumption per cycle

The additional tokens arriving require a refinement of the token consumption rate definition. See figure 3.14 for an example system with a node o_{j-1}, node o_j and node o_{j+1}.

Theorem 2 (Multiple Token Execution Rate) *The rate at which node o_j must be scheduled to consume tokens to adhere to the RDF time correctness criteria is:*

$$\frac{y_{j-1}}{z_j}$$

where y_{j-1} is the number of tokens produced by a single producer connected to node o_j and z_j is the number of tokens required to trigger the execution of node o_j and subsequently consumed when node o_j fires.

Proof 2 *It is to be proved that messages must be received and processed before the sender sends again. Given that y_{j-1} messages are produced by node o_{j-1} sending them to node o_j in a single execution cycle, and node o_j produces only one message and consumes z_j messages, the transmission rate function for node o_j is:*

$$f_j(r_j) = \frac{1}{x_j} r_j$$

If node o_j emits y_j messages, the transmission rate function becomes:

$$f_j(r_j) = \frac{y_j}{x_j} r_j$$

The next node o_{j+1} must consume y_j messages before another y_i messages are produced, and node o_j must consume y_{j-1} messages before it can receive a new message vector. This is only possible if node o_{j+1} consumes y_j messages and node o_j is scheduled

$$\frac{y_{j-1}}{z_j}$$

times to execute before node o_{j-1} is scheduled again.

With multiple input ports into a node, the rate at which the consumer must be scheduled, becomes the sum of the input rates. See Figure 3.15 for an example with the notation used in Definition 51.

Definition 51 (Multi-rate Logical Input Port) ***A multi-rate logical input port** is an equivalent input port when all n data streams from the independent input port set are combined for node o_j. The combined rate at which the node must execute to consume all messages for a system to be correct from the timing point of view is:*

$$rm_{in} = \sum_{i=0}^{n} \frac{y_i}{z_i} r_i$$

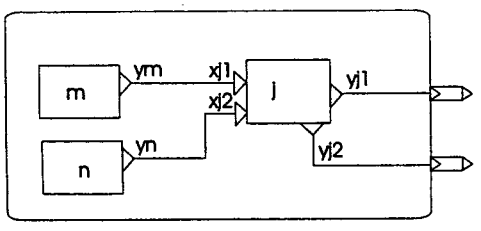


Figure 3.15: Multiple input rate definition for multiple input ports

With the Theorem 2, all other results valid for homogeneous RDF data flow graphs can now be extended to accommodate multi-rate RDF graphs. This is a significant improvement in the analysis capability of RDF systems.

3.8 Stategraphs

The behaviour of each of the task nodes can internally be described by a stategraph. These parallel stategraphs for all task nodes in a system together constitute the system stategraph vector. Statecharts [35] is a compact rigorous notation for representing parallel and nested state machines. It is, therefore, used as stategraph representation in conjunction with RDF for the expression of task node internal state transitions based on incoming messages (tokens) and for the resulting output messages (tokens) generated. In addition, it also represents the execution duration interval of each task node. The structure of Statecharts is amended as follows for the application of stategraphs.

Definition 52 (Timed State Machine) *A* **Timed State Machine** *(TSM)* Q_i *is state machine corresponding to the internal transitions of behavioural node* o_i. *It consists of* state nodes q_i *and* interstate edges w_i.

Definition 53 (State Time Function) *A* **State Time Function** *for a node* o_i *is the interval* $[\delta_{i,j}, \Delta_{i,j}]$ *where* j *is the* j*-th state of node* o_i. *The actual execution duration for the state* j *is* $\lambda(o_{i,j})$.

The set of timed state nodes for a behavioural node o_A is shown in figure 3.16 as $\{q_1, q_2, q_3\}$:

Definition 54 (Timed State Nodes) *A* **timed state node** *of a TSM* Q_i *of node* o_i *is a state node* q_j *and* $\forall j : Q_i = \cup q_j$.

Each timed state node represents an execution duration which contributes to the time function for the operational node. In figure 3.16 the state time execution duration for q_2 is $[2, 5]$.

Definition 55 (State Execution Duration (SED)) *The* **SED** *is the time spent in the state after entering it, to before the test for exiting the state is enabled. The SED for state* q_j *of node* o_i *is a state time function* $[\delta_{i,j}, \Delta_{i,j}]$.

In the stategraph in figure 3.16, the transitions across edges between states q_i are fired with the same rule as in RDF. The actions on the edges leaving the state node q_j correspond to the token/message sent in the RDF. The edges that interconnect the states in a stategraph are called **interstate edges**.

Definition 56 (Interstate Edge Firing Rule) *An* **interstate edge** *is triggered when the firing rule according to Definition 30 is satisfied for an* enabled input port set *of node* o_i. *The action on a transition corresponds to a message emitted in the corresponding RDF graph.*

The information that was required to calculate the time function $\Lambda(o_i)$ defined in Definition 11 for the behavioural node o_i is now available.

Definition 57 (Time Duration Lower Bound) *The time duration lower bound is the sum of the state time functions' lower bounds for the shortest path between entry*(o_i) *and exit*(o_i) *of a behavioural node* o_i.

Definition 58 (Time Duration Upper Bound) *The* Task Node Execution Duration *(Definition 11), upper bound for node* o_i *is the sum of the upper bounds of each timed state node for the longest path from entry*(o_i) *to exit*(o_i).

The stategraph as described by the corresponding notation does not control the data flow machine. It tracks the data flow machine and maintains state information that is not available in the RDF graph. In particular, the following information is available in a stategraph:

1. the current state of the task node, corresponding to the token (message) input sequence,
2. the execution duration (or elapsed time) for the task node.
3. The task node transfer rate function can be deduced from the minimum path length between corresponding output actions of the same message on the same output port.

See Figure 3.16 for a stategraph from which the time duration, the node rate transfer function and the current state of execution can be derived. For state q_2 the state time duration $[\delta_{A,2}, \Delta_{A,2}]$ is the range $[2, 5]$.

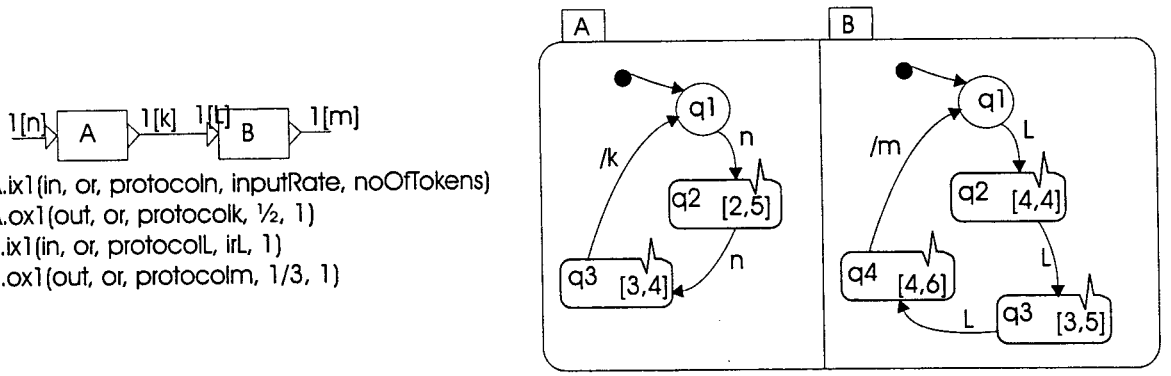


Figure 3.16: An RDF and accompanying stategraph from which the transfer rate functions and time durations can be derived

Considering the use of variables in stategraphs. An integer variable in a stategraph represents a very compact way of describing a large number of states which only differ in the value of the variable. Thus, Boolean and integer variables are recommended in stategraphs to succinctly represent a large number of states in a very compact and intuitive way.

3.8.1 Stategraph Hierarchy

Stategraphs are state machines expressed in hypergraph format [35]. All events in the original definition of Statecharts are global which enables only horizontal decomposition possibilities. Stategraphs can express AND relationships and OR relationships between states. An AND relationship indicates that the state machine is in a state A and a state B simultaneously implying parallel execution. An OR relationship indicates that a machine can be in one and only one state within the Stategraph.

For vertical hierarchical system decomposition one requires events visible from a certain level down. This adaptation of Statechart semantics is assumed for our use in Stategraphs. During the design/specification of a system, the important decision is which events are visible at which level of the Stategraph, and which events can cause a state change. The restricted visibility events as defined in the previous paragraph aid in reducing the complexity of having all events visible at all times at the upper level decomposition of the system.

A stategraph corresponds to the same level of decomposition at which an RDF graph $\mathcal{G}$ exists and must obey the data flow actor semantics. Thus, it must correspond to the consumption and production of the number of messages as indicated in the RDF graph $\mathcal{G}$. The events which are visible on the stategraph thus correspond to messages in the RDF graph $\mathcal{G}$. See Figure 3.17 for an example system described with the visual data flow language and a stategraph corresponding to the same level of decomposition with task nodes $\{o_A, o_B, o_C\}$.

Each state in a stategraph can execute with no processor resources consumed (*idle state*) or it can indicate execution time (*behavioural state*). See Definition 11 for the task node execution duration $\Lambda(o_i)$.

The actual execution time will vary from execution instance to execution instance depending on non-determinism such as hardware optimisations. The actual execution time for the state j of node o_i is $\delta(o_{i,j}) <= \lambda(o_{i,j}) <= \Delta(o_{i,j})$. It must be the result of statically compiled code, of which the execution time can be measured and the bounds of the execution time defined.

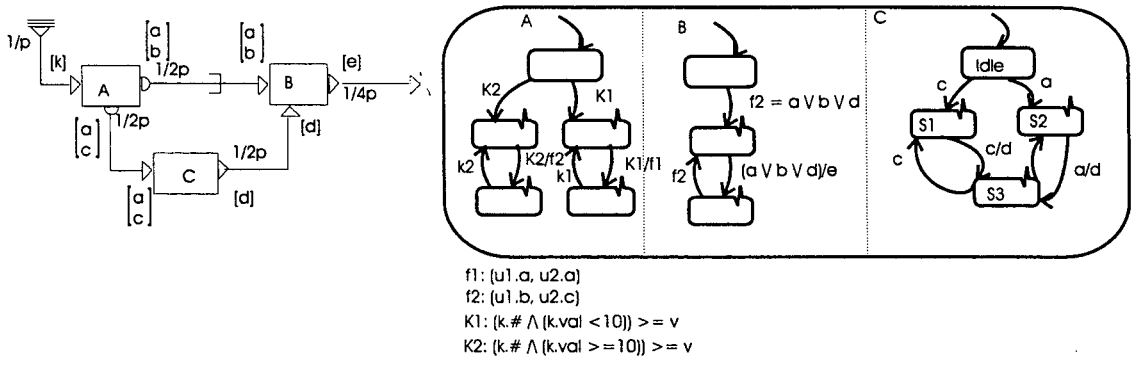


Figure 3.17: Example system and Statechart describing the behaviour

3.9 Multi-view Building Blocks

All aspects of a software component can only be described by using multiple views. These views were identified as behaviour, structure, and project development information. Each of these views exists in a logical or physical model depending on the maturity of the software/hardware component. Representing each of these views requires a language for each one. The languages adhere to the principle that a visual language provides a better overall impression, and that a textual language is better at representing finer detail. The RDF language has been described in some detail in Section 3.2. Stategraphs were briefly explored in Section 3.8 as the semantics associated with the stategraphs graphical formalism is somewhat different than Statecharts. The other three languages are either simple in semantics, i.e. project development language and structure diagrams, or are existing languages/representations, i.e. object oriented class structure diagrams. Only a brief description of the three languages will be given.

The next section will give a brief description of the three languages other than the RDF and stategraph languages. Particular emphasis will be put on the hierarchical combination supported and how the languages inter-relate to each other. The closing section gives an example showing how all the views are applied to an example on one layer of decomposition.

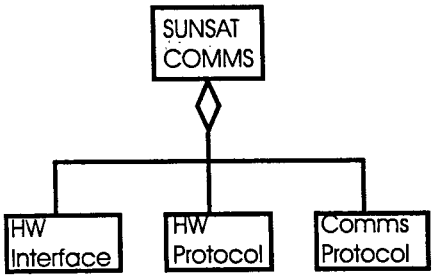


Figure 3.18: UML notation for class structure and an example of a communication system

3.9.1 Object Oriented Class Structure

The object oriented class structure forms the basis for organising software for development and re-use. Software is organised in the analysis and design phases such that the organisation structure leads to more efficient re-use of the developed components. All relationships among classes in the UML class diagram notation [7] are supported. This enables to specify and design with all advantages of the object oriented approach for component structuring and re-use.

Within the architectural model we chose, i.e. communicating tasks, there is a strict guideline to the application of the object oriented class structure in the physical model. Tasks corresponds directly to instances of classes. There is, thus, a natural transition within the architectural model from the logical class description to the physical task description. This advantage is only gained by having chosen a specific architecture for all the software components. Figure 3.18 shows the notation used to describe the class structure and an example of the communication software on a space craft. Figure 3.19 shows the class category icon and an example of a communication system's top level class diagram.

The ways to describe the hierarchy of object orientation are far richer than that of RDF, and relate to different ways of re-using code. See Figure 3.20 for some examples. They are, however, orthogonal to the RDF hierarchy, allowing to use the hierarchy or relationship possibilities of UML to its full extent and, finally, show the correspondence between classes and operational nodes of an RDF. The stategraph description associated with a task node, corresponds to the behaviour described inside a class. The hierarchy of the RDF data flow language is strictly that of composition in the context of object orientation. That is, in RDF a task node can *contain* one or more RDF task nodes and, then, it becomes a composition task node.

Packages represent one particular way in which the classes (which correspond to operational nodes) can be organised for storage and retrieval. Thus, the object oriented characteristics are used to organise code for re-use.

Some of the different ways in which inter-relationships between elements can be described are summarised below and shown in Figure 3.20:

1. **Association** is a relationship between elements and refers to how two elements are related.
2. **Generalisation** relates a more general element and a more specific element; implemented with inheritance.
3. **Dependency** between elements leaves the one independent and the other dependent.

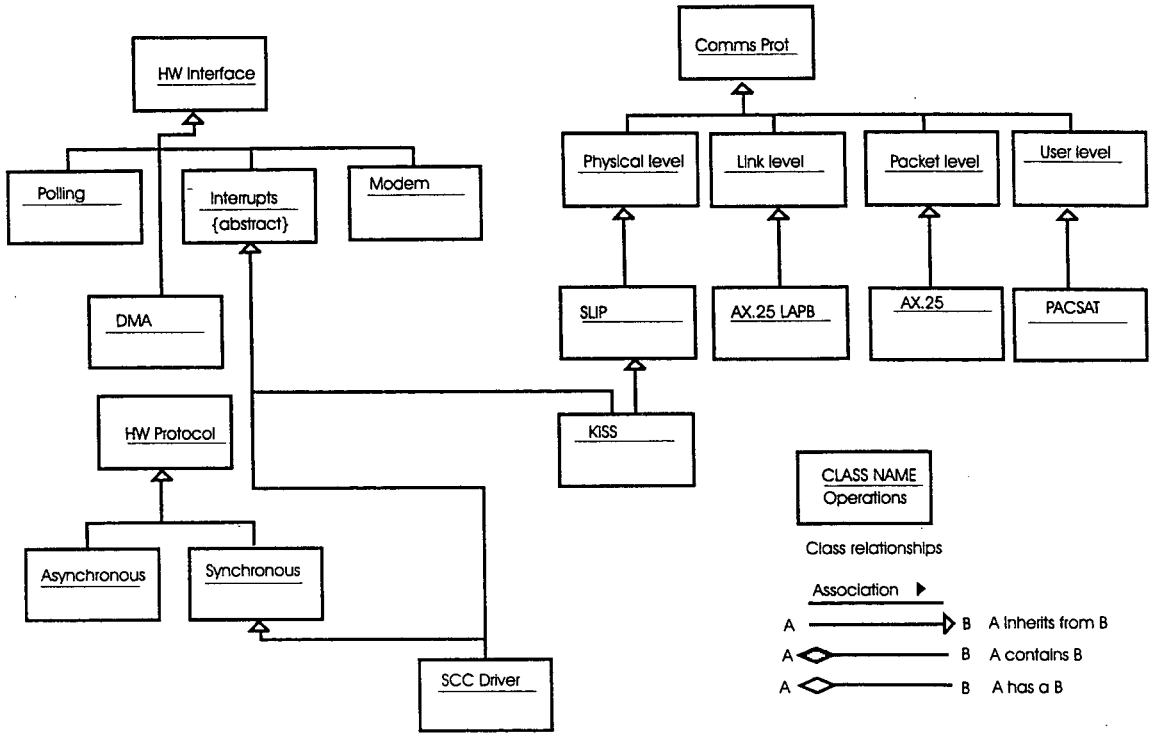


Figure 3.19: Top level class diagram of a microsatellite communication system

4. **Refinement** is a relationship between two descriptions of the same element, but at different levels of abstraction.

5. **Packages** are groupings of related elements into libraries.

3.9.2 Structural Description

Structural description of hardware is well established with many graphical and textual languages. We shall use a graphical language as described below. The hardware elements of a system structure are shown with rectangles with the names of elements in a convenient location of the rectangles. If the software function of a hardware element is of interest, that part of the block is shaded. The interconnections between elements are shown in three levels of detail: a connection of any kind between two elements, a composite connection between two elements (also called a bus), and a specific connection between two elements which must contain a detailed signal name description with each connecting line. These graphic elements are shown in Figure 3.21 for a very high level description of a typical microsatellite ground station. If additional information on a link or in a block is required, an edge or the block is labelled, and a separate table is included with the additional information.

3.9.3 Project Development Properties

Development information provides project management information needed during the engineering of a system, and product maturity information useful for re-use of the individual components. To engineer a system, it is decomposed into components which are developed within a

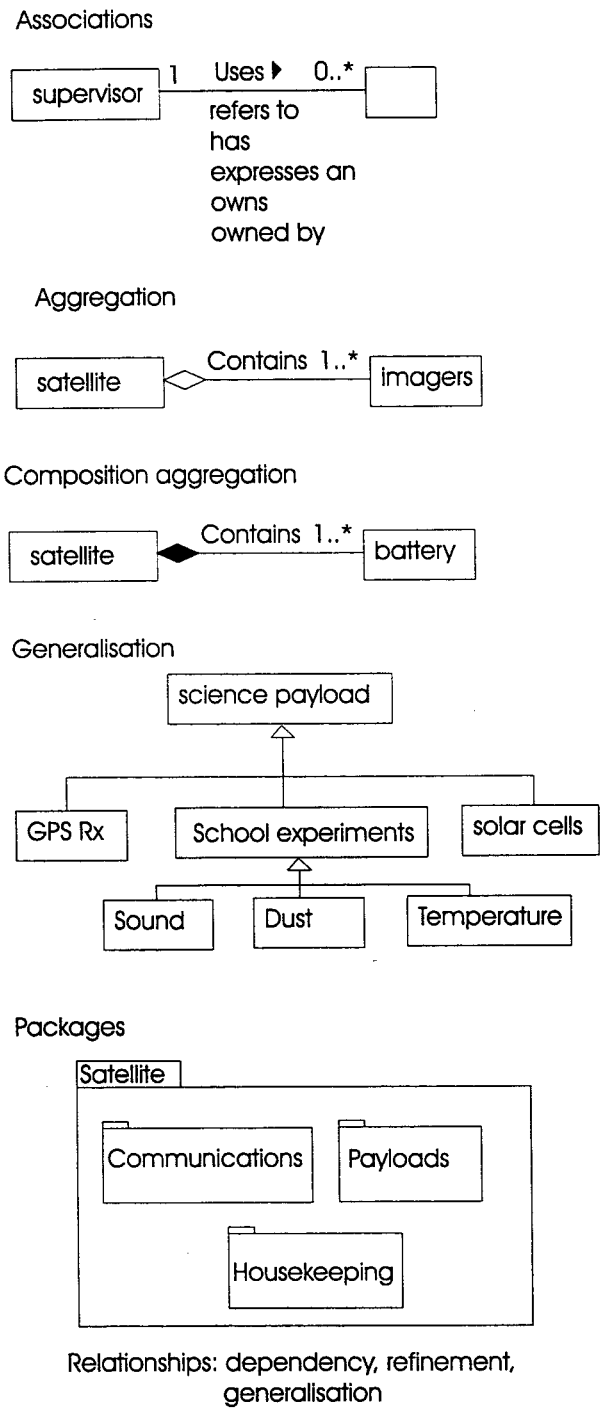


Figure 3.20: Essential class diagram notation in UML

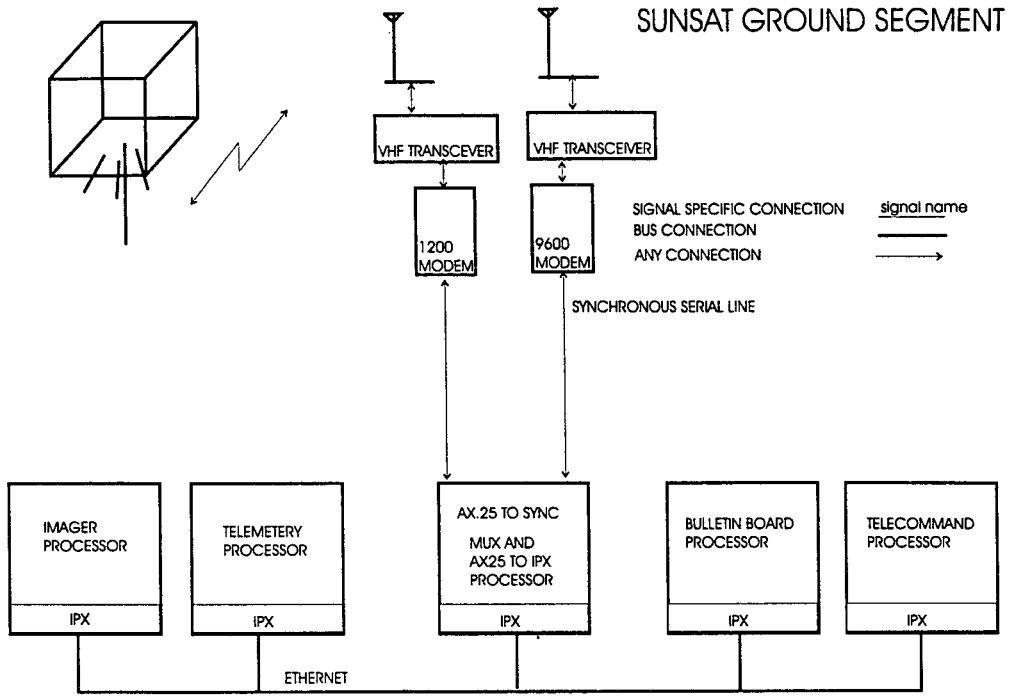


Figure 3.21: Definition of structural language and example system

development framework. The development framework describes each software component with the following parameters:

1. development plan and schedule,
2. development record and status,
3. specification and implementation languages and tools,
4. configuration control of the component including the storage location of the component and its version number, and
5. the responsible person.

Within the top-down specification and design process, the project development properties form a natural vertical hierarchical structure, i.e. the complete system is not complete until every component inside is not complete. However, although correct this global information is not useful on the microscale. Thus, each component on each decomposition level is described by the set of parameters. For a vertical decomposition process to function correctly, the top level component adopts the status of the least developed component in its hierarchical structure.

Within the bottom-up implementation process, the hierarchical convention is that a component being engineered has its status updated, while all other stati are unaffected. Furthermore, all lower components inherit the status of the higher component unless the component had its status explicitly changed. This ensures a valid conservative development status for each component across a project. To enable easy identification (at a glance) of the status of tasks in a development environment, the tasks are shaded according to their development status. The four stati that have proven useful are:

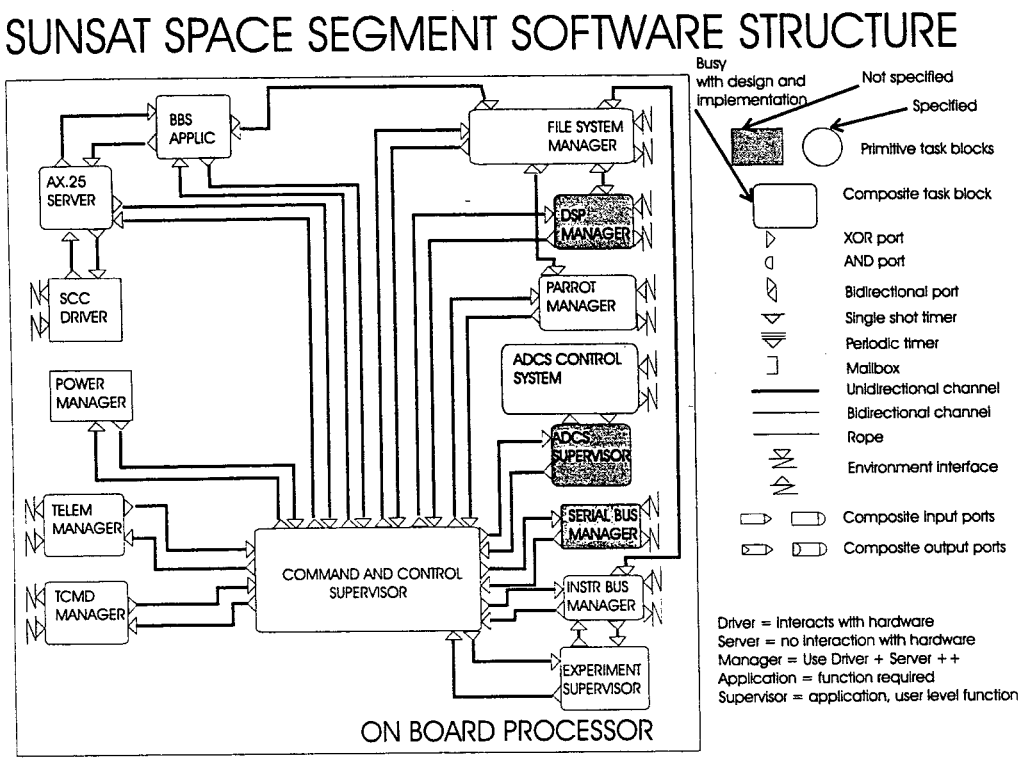


Figure 3.22: An example of the development status of the space segment software of SUNSAT

1. not specified — very dark grey,
2. specified — dark grey,
3. busy with design and implementation — light grey,
4. released — white.

Inter-relationships

A multi-view description of a system reaches its full potential when the views are inter-related. What is important is how each of these languages support hierarchical decomposition, and how each of these languages inter-relate to each other on each level of hierarchical decomposition. The five languages RDF, stategraphs, system structure, class structure and project development, have a very close inter-relationship which enhances their joint use. The inter-relationship can be summed up as follow.

1. The communicating tasks are objects that are instances of the class structure of the system.
2. The stategraph description on a decomposition level corresponds directly to the behaviour of the system described by the data flow language decomposition on that level.
3. The structure of the system encompasses any number of tasks and stays the same or can be refined throughout the decomposition process.

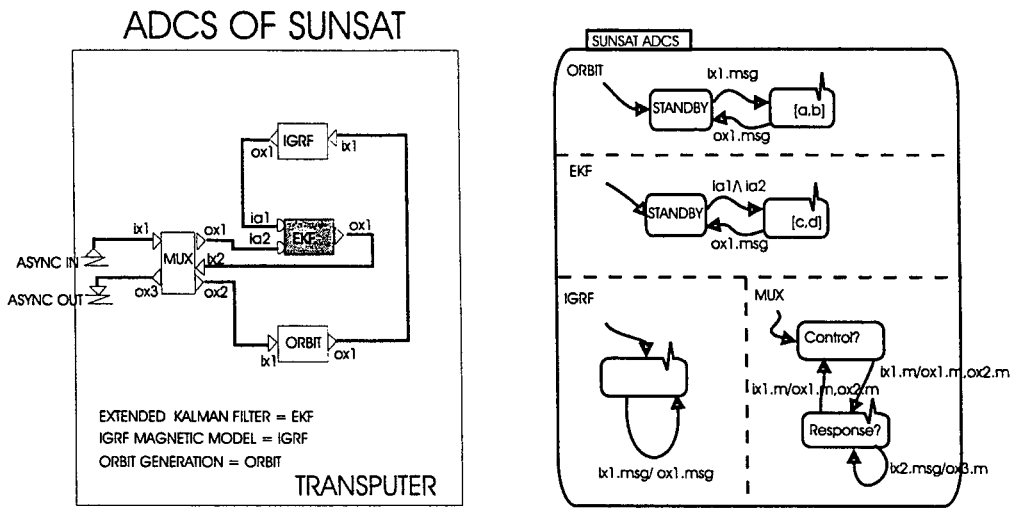


Figure 3.23: Designing the space segment of the ADCS demonstration system

4. The project development view is usable on all levels of decomposition. The visual development status indicator is a conservative indicator of the development status of the system.

3.10 From Specification to Programming

This section contains two small examples which, in the first instance, show one step in the design refinement process and secondly, how to use the visual data flow language for programming in the small. These two examples show that the languages are applicable from functional specification to implementation.

The example to show a refinement step in the multi-view software suite builds on the ADCS demonstration example described in Chapter 5 Section 5.6. Four of the five views are shown. The RDF graph with its corresponding stategraph is shown in Figure 3.23. The project development status is clearly shown in addition to the resource allocation on a Transputer processor. The class structure for the communication tasks is shown in Figure 3.18, of which the asynchronous in and asynchronous out are two objects instantiated as tasks for this application.

It is possible to express notions of control flow with the real-time data flow language. Figure 3.24 shows how a control construct can be implemented in an iterative language. The notion of a remote or normal procedure call is expressed with the bi-directional ports as shown in the lower left hand side of Figure 3.24.

3.11 Summary and Discussion

This chapter presented a multi-view approach to software component description and introduced a real-time data flow language, RDF. Such a multi-view approach is essential for effective communication during software development in addition to a rigorously described computational architecture.

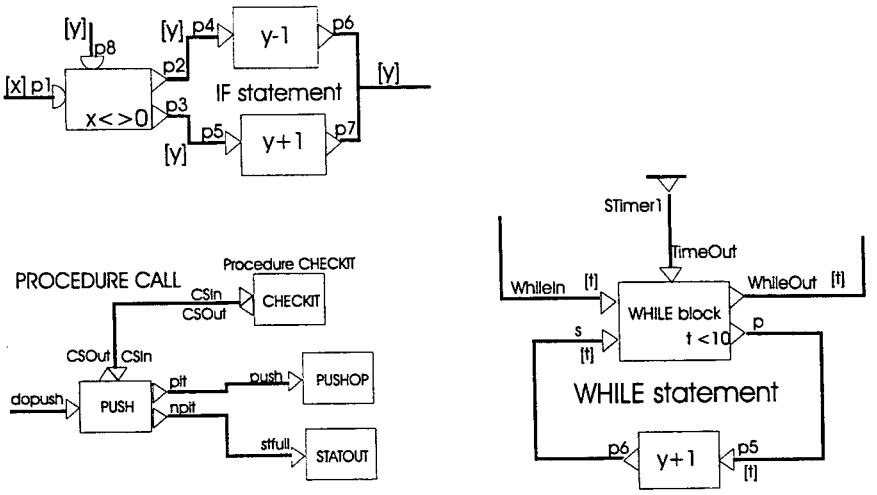


Figure 3.24: Programming constructs to describe control flow with the visual data flow language

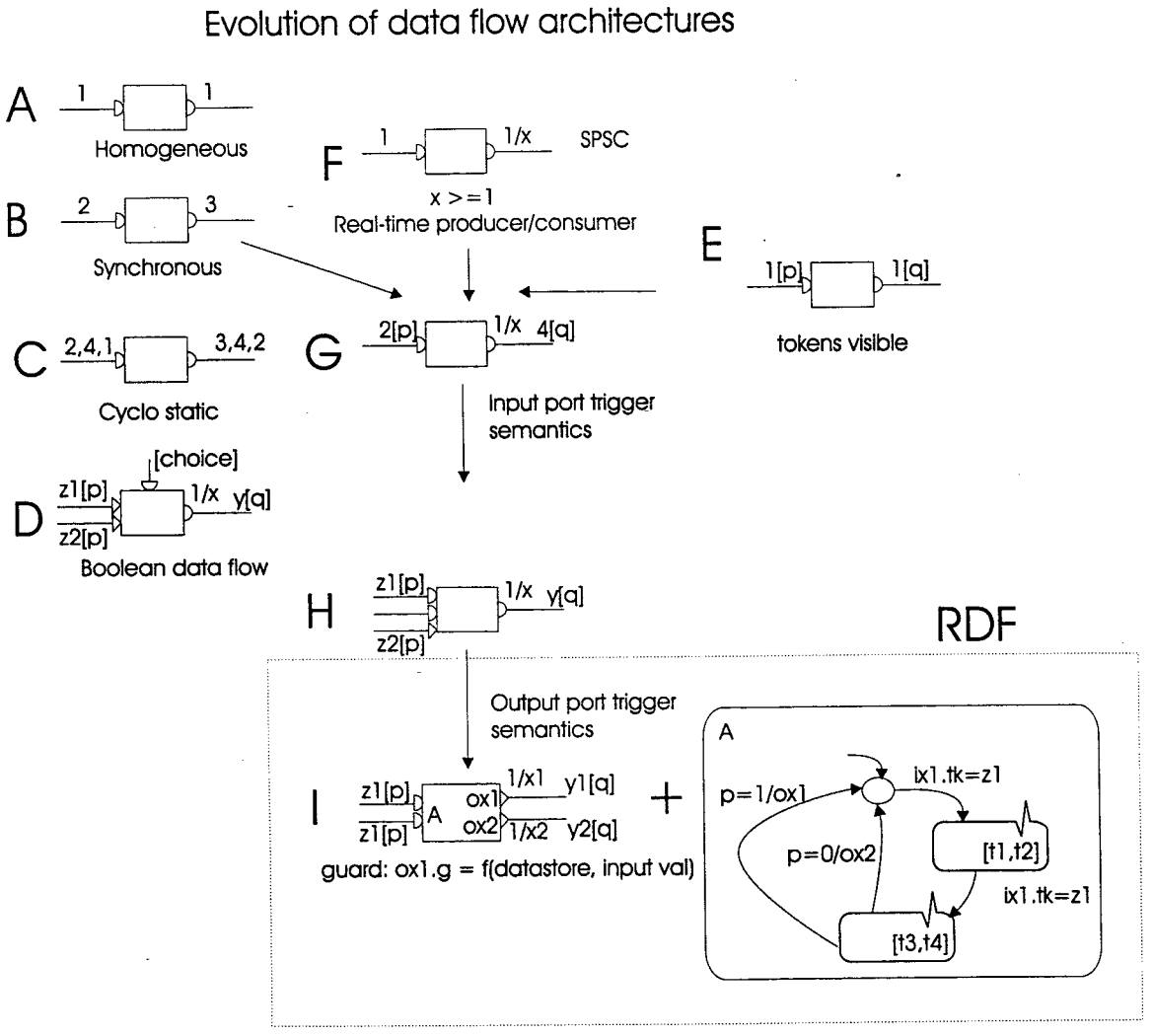


Figure 3.25: The evolution of data flow languages

The real-time data flow language RDF allows to express the real-time behaviour of systems. RDF extends the Real-Time Data Flow (RTDF) paradigm in numerous ways. The trigger semantics of ports were expanded to allow a larger class of applications than just RTDF to be described by this language. Other extensions enable to describe synchronous data-flow communication.

The RDF architecture provides a platform for a task set of which an advance static schedule is unknown due to the unpredictability of environmental interactions. The semantics expressed in the RDF language assume a worst case message arrival rate and hence hard real-time properties of the system can be guaranteed.

The following are significant contributions made with the definition of the RDF language suite:

1. the merging of the properties offered by the RTDF paradigm with the developments in synchronous and other data flow extensions while maintaining the analysability of the RTDF paradigm. See Figure 3.25 for a diagrammatic representation of the evolution of data flow languages with RDF one of the outflows of this evolution.
2. the inclusion of AND port and OR port trigger semantics on the input ports and output ports of a task node, which together offer the possibility to express control flow constructs with a data flow network (see Section 3.10),
3. the AND/OR port semantics on the output ports which offer data dependent execution to be visible at the RDF modelling level (see Figure 3.24),
4. the extension of RDF with a state description to allow a richer number of properties to be described by the notation (see Section 3.8),
5. the matching of object orientation at an appropriate level of abstraction to offer the RDF language suite the best mechanism available in program code organisation and re-use (see Section 3.9.1),
6. the specific ability to construct a system hierarchically with a multi-view language suite including the semantics with regard to the timing properties (see Section 3.5).

Establishing the inter-relationships of the five languages is important for the description of real-time software components. Each language on its own can describe a larger class of systems. However, the inter-relationships allow one to say far more in a rigorous way about a software component, meaning that it is more re-usable in the long term due to the improved knowledge about it.

Languages are but one set of tools to engineer real-time systems. The data flow language is motivated for its ability to express parallel operation of tasks as is found in the environments in which real-time systems operate. This means that if the architecture assumed by RDF can be realised in hardware, there is no transformation required from design specification to hardware implementation, which aids in simplifying the validation step.

The next chapter describes the implementation of systems described with RDF in a way that maintains the architectural knowledge expressed in RDF. This is one of the keys to a better understanding of a system, leading to shorter development time, shorter testing time, and a smaller maintenance effort for real-time systems. Implementations of the RDF computational architecture in software and hardware are described. A simple processor architecture which maps exactly to the tasks, ports, timers, deadlines and channels is going to be introduced.

Chapter 4

Implementing the RDF Architecture

The previous chapter introduced the structure and semantics of the real-time data flow language set. The Real-Time Data flow language (RDF) is the base language with which a system is described. The other languages support views that are difficult to express in RDF. This chapter focuses on implementing systems described in RDF, and uses the other languages where appropriate.

The architecture (structure and computations) embodied in the RDF language needs to be available in an implementation target matching exactly the features of RDF, if a simple validation process is desired. The premise of this dissertation is that once the implementation architecture matches the design architecture, it will lead to shorter development cycles and simpler (and hence shorter) validation operations. This chapter shows three ways to realise the architecture described in Chapter 3. In each case the implementation supports the computational features required by RDF.

The traditional way of realising an architecture is to implement it in software on another architecture. Such an implementation makes the RDF architecture available on any existing single processor or multiple processors. Verifying the timeliness properties of a task set on a single processor is a subject of the field of schedulability analysis. Results from the research area of schedulability analysis are used to verify the feasibility of an RDF task set executing on a single processor.

Two other ways of realising the architecture use dedicated hardware. In particular, a simple processor is introduced which supports the process execution model of RDF. This simple processor is unique in that it can be expanded with the correct amount of resources to guarantee the execution of one task in the task set. With System-On-a-Chip (SOC) technology becoming more accessible, a number of these simple processors can be integrated into one package.

Two traditional processors in parallel lead to a dedicated hardware implementation with commercial-off-the-shelf (COTS) hardware technology. One processor performs all RDF related computation, while the other one is optimised to execute functions for a particular application. The execution of the functions on the function processor is under the management of the RDF processor.

The combination of the three architecture realisations is unique as it offers an immediate software solution, a resource adequate hardware solution with conventional processors, and a resource adequate hardware solution with RDF optimised simple processors. The final validation

of real-time properties is supported by the one-to-one correspondence of an implemented task with a designed task in all three realisations.

This chapter starts with the software realisation of the architecture, and explores the additional schedulability tests required for a resource adequate single processor implementation. The simple processor with its architectural features supporting RDF is described next. The third implementation with a processor pair is introduced in the context of a microsatellite case study. The actual use of the software implementation of RDF on the SUNSAT project is discussed before concluding the chapter.

4.1 Establishing Task Boundaries

In all implementations of a system based on the RDF paradigm it is required to decide on which processor a task will execute. In the one extreme of a resource constrained implementation such as the SUNSAT microsatellite, with only one processor for the complete task set, it is useful to know which tasks can be invoked in the same task boundary without a re-scheduling operation. In the other extreme, to each task can be allocated its own dedicated processor. The question remains where the boundaries for this task are. For all task-to-processor allocations, the task boundaries determine points where the task set can be split to run on different processors. In many implementations of telecommunication and control systems, there are specific hardware interfaces available on certain processors, only. Access to the hardware interfaces pre-determines where certain task sets will execute. The task execution boundaries need to be determined for single processor scheduling or multiprocessor allocation. The rest of this section gives a brief overview on a decision criterion that can be used to determine the boundaries of a task.

The RDF output port rate functions indicate the worst case production rates at which tokens are produced and, hence, the worst case rates at which they must be consumed by the receivers. The state transition path in the state graph of a receiving node gives the execution time for processing a token. The execution time for processing a token is the sum of the time durations in each of the states in the state transition path in the state graph. Each time duration for a state indicates the execution duration of the functions that are executed while in that state.

The only non-deterministic time synchronisation is restricted to the *environmental interface* nodes in an *RDF*. There are other factors contributing to the non-deterministic time durations that include 1) data dependent loops, 2) conditional paths inside a node, and 3) hardware effects such as caches, speculative execution etc. The execution duration uncertainty for the latter three influences is modelled in the worst case duration that is spent in each state.

Definition 59 (Schedulable Unit) *A schedulable unit is a maximal set of connected nodes with a deterministic execution time.*

The deterministic execution time relates to a predictable execution order of the task nodes. The inter-*schedulable unit* boundaries will provide the points where scheduling decisions can be made and are, hence, good boundaries for determining the allocation of tasks to different processors.

Definition 60 (Schedulable Unit Execution Time) *The* **schedulable unit execution time** *is the sum of all execution durations of the states in a state transition path representing a schedulable unit.*

A schedulable unit represents a code unit that can be compiled without any regard for the system level issues. All system level issues, such as synchronisation with the environment and process level concurrency, are modelled in the system level layer. One of the results of identifying a schedulable unit is that inter-task communication inside a schedulable unit can be compiled/implemented as procedure calls.

In the design refinement process with RDF it is useful to distinguish between the following two levels.

1. The system level layer. On the system level layer, aspects such as synchronisation with the environment and inter-process concurrency and synchronisation are represented.
2. The process level layer. On the process level layer, all aspects that can be scheduled statically can be represented.

On the process level layer a process function will be implemented with a sequential program code or with a data flow code.

In all the implementations that follow it is assumed that the task boundaries have been established, either through the cognitive partitioning process of the design engineer or with an automated system applying the above definitions without regard for other influences on the allocation of tasks.

4.2 Software Kernel Implementation

The architecture described in this dissertation can be implemented by one processor per task to ensure a resource adequate implementation guaranteeing hard real-time constraints. However, for the architecture to be universally usable on existing processors, it must be able to used on a single processor system. Scheduling tasks on a single resource constrained processor is inevitable in situations where a standard hardware processor is prescribed. Such a case is the SUNSAT microsatellite where a set of processors was prescribed due to limits on mass, power and a space legacy.

Scheduling a set of tasks on a limited set of resources while maintaining real-time guarantees has received a lot of attention in the research community [71, 95, 40]. To support the thesis of an architecture that can be used from design to implementation for dependable real-time systems, the earliest deadline first (EDF) scheduling algorithm is selected. The maximum rates at which messages must be consumed translates to an expression as a deadline.

Systems with hard real-time components require guarantees with regard to the timeliness of task executions by the processor. In the case of a complex system such as a multi-mission microsatellite, it can be expected that the processing load will vary over time. For a resource constrained processor implementation, this might result in a processor overload. This suggests that the processing load must be measured and corrective steps taken if the task set exhibits transient processor overload behaviour. An algorithm to address this particular problem is proposed.

4.2.1 Scheduling Results

A system described in RDF is represented as a set of communicating tasks. To ensure that the implementation is temporally correct, the scheduling algorithm must schedule the tasks so that they conform to the RTDF paradigm.

A task repeatedly accepts one or more messages, consumes the messages, emits one or more messages, and waits for the arrival of the next messages. This repetitive behaviour indicates a *cyclic* task.

Definition A *cyclic* task T is a 3-tuple (s, c, p) with

1. s = start or release time: the time of the first request for execution of task T,
2. c = computational cost: the time to execute task T to completion on a dedicated single processor, and
3. p = period: the interval between requests for the execution of task T,

where s, c, p are multiples of some indivisible time unit. In general, the release times of tasks are unknown and do not influence the scheduling results or decision procedure that follows.

Cyclic tasks can be periodic (time driven) or sporadic (event driven). The behaviour of a sporadic task T is slightly less constrained than that of a periodic task and its correct real-time behaviour is described by the following rules:

1. Task T makes its first request for execution at time $t_1 = s$.
2. If T has period p, then T makes its $(k+1)^{st}$ request for execution at time $t_{k+1} >= t_k + p >= s + kp$.
3. The k^{th} execution request of T must be completed not later than the *deadline* $t_k + p$.
4. Each execution request of T requires c units of execution time.

The period of a sporadic task is the minimum time between any two successive execution requests of the task. In order to determine schedulability for a set of task, we need to verify that no task will miss a deadline. The property of interest is called feasibility.

Definition 61 Feasibility
A set of sporadic tasks τ is said to be feasible *on a single processor if it is possible to schedule τ on a single processor, such that every execution request of every task T is guaranteed to have completed execution before its deadline.*

Feasibility of a specific task set is relative to a specific scheduling algorithm. An optimal scheduling algorithm can schedule any feasible task set. A feasible scheduling discipline is, thus, as good as any other discipline that can correctly schedule any task set that is feasible, which leads to the following definition.

Definition 62 Feasible Scheduling Discipline *A scheduling algorithm is feasible, if it produces a feasible schedule for any feasibly schedulable task set.*

The Earliest Deadline First (EDF) [71, 40] scheduling policy is chosen as it has been shown to be optimal on single processors. The EDF policy also represents a close match to the way in which time constraints are specified in RDF. There is a number of decision procedures for establishing the feasibility of a task set to be scheduled with the EDF scheduling algorithm. In [53] the following results for feasibility and optimality of the earliest deadline first scheduling discipline were proven. The results can be summarised as follow:

1. For pre-emptive scheduling
   (a) EDF is an optimal discipline if pre-emption is allowed at arbitrary points in a process, and
   (b) feasibility of a task set can be determined analytically for arbitrary release times of both sporadic and periodic tasks.
2. For non-pre-emptive scheduling
   (a) EDF is an optimal discipline for sporadic tasks, but is not an optimal discipline for periodic tasks with arbitrary release times, and
   (b) feasibility can be determined efficiently for sporadic tasks with arbitrary release times.

A non-pre-emptive scheduling mechanism is selected for the following reasons:

1. Shared resources and data stores do not require semaphores as mutual exclusion is guaranteed through the run-to-completion mechanism.
2. Most tasks are short in duration and do not block any other task past its deadline. The decision procedure described in the next section checks for this.
3. The events that do require pre-emptive processor attention are mapped on the processor interrupts.
4. The tasks which do not require real-time guarantees can be time sliced when no other activities are present in the non-pre-emptive task set.

Decision Procedure

The *decision procedure* to determine whether a set of sporadic tasks is feasible can be deduced from the necessary conditions for the feasibility of a set of sporadic tasks which is scheduled non-pre-emptively with the EDF discipline. We assume for convenience that the set of tasks is sorted in non-decreasing order by period $p_i \geq p_j$ if $i > j$. The *index* of a task refers to its position in this sorted list. The following definition states the necessary conditions of which the proof can be found in [52].

Definition 63 *A set of periodic tasks $\tau = \{T_1, T_2, ..., T_n\}$, sorted by period in non-decreasing order, can be scheduled non-pre-emptively without inserted idle time for all possible release times only if:*

$$(1)\ \sum_{i=1}^{n} c_i/p_i \leq 1,$$

$$(2)\ \forall k, 1 \leq k < n; p_k \geq MAX_{i:p_i>p_k}\left(c_i + MAX_{0<l<p_i-p_k}\left(-l + \sum_{j=1}^{i-1} \lfloor (p_k + l - 1)/p_j \rfloor c_j\right)\right)$$

Condition (1) is the non-overload requirement, viz. the complete load on the processor may not exceed the capacity of the processor. Condition (2) states that for each task $T_k, k < n$, the right hand side of the inequality is the sum of the execution cost c_k plus the worst case delay that can occur between the time task T_k makes a request for execution and the time when it is scheduled. The worst case delay is primarily a function of the relative sizes of tasks periods. If all tasks have the same period, condition (2) is empty and condition (1) is the only necessary condition.

Since the non-pre-emptive EDF discipline is optimal for sporadic tasks, in order to decide if a set of tasks is feasible on a single processor we only need to consider if conditions (1) and (2) of the previous definition hold. Condition (1) is computable in $O(n)$ time. For condition (2) the following computational procedure is recommended in [52]. For each task T_k, we can compute

$$MAX_{i>k}\left(c_i + MAX_{0<l<p_i-p_k}\left(-l + \sum_{j=1}^{i-1}\lfloor(p_k + l - 1)/p_j\rfloor c_j\right)\right)$$

as follows. Let

$$f_{k,j(l)} = \lfloor(p_k + l - 1)/p_j\rfloor c_j$$

and

$$S_{k,i(l)} = c_i - l + \sum_{j=1}^{i-1} f_{k,j(l)}$$

To determine whether condition (2) holds, we compute $S_{k,i}(l)$ for $1 \leq k \leq n-1$ and $k < i \leq n$. The values $S_{k,i}(l)$ are tabulated with i as the column header from $k+1$ to n and l as a row index from $l = 1$ through $p_{k+1} - p_k$ to $p_n - p_k$. The largest value in each column represents the maximum delay that a task can experience due to a task with a longer period executing. If any of the maximum delays imposed by tasks $k+1$ to n is larger than the period of task k, viz. p_k, then condition (2) is not satisfied.

4.2.2 Implementation Issues

Up to this point, the description of a system in RDF does not contain enough information for the final mapping to a software implementation. A number of implementation issues must be resolved.

Multiple Input Port Implementation Strategy

Tasks with multiple input ports are modelled as having a logical input rate that is the accumulation of the individual rates. This is, however, not useful in the feasibility analysis as each input port must have one trigger period and one computational time period. Applying the conditions in Section 4.2.1 on a sporadic task set means that each channel can be treated as if it connects to a dedicated task which processes only messages from that channel. Condition (1) is verified with a spreadsheet and condition (2) is verified with a program in Matlab given in Appendix C. An implemented task with more than one port which is enabled to fire, chooses the one with the closest deadline to execute. This is called earliest deadline first choice.

Feasibility of Execution

A system described in the RDF notation can only be checked against the design paradigm, i.e. whether every pair of nodes connected with a unidirectional channel adheres to the RTDF paradigm. The maximum consumption rate for each input port is a function of the processor which executes the task triggered with a message on the input port. The consumption rate is, thus, a function of the processor type, clock speed, and the architecture of the hardware. The value of maximum consumption rate can only be calculated/measured once the processor is known.

Timers

Timers connected to a task influence the logical input rate of the task. The logical input rate is the sum of the input rates of each of the input ports, which now include the timer sources. The expression for the virtual input rate is:

$$r_{in} = \sum_{i=1}^{n} r_i$$

where r_i is the input rate on input port i connected to the task. The worst case period between incoming tokens of the task is then $1/r_{in}$. Each timer is connected to the task with its own input port.

Efficiency of Execution

The computational model described thus far assumes that all inter-task communication is via message communication across channels. In certain instances it may be more efficient and convenient to communicate by subroutine calls. This decision can be made for task boundaries inside a schedulable unit. Task boundaries of a schedulable unit represent possible points of non-determinism on which task is going to run next and, hence, cannot be implemented with a subroutine call. The notation in RDF to indicate a subroutine call is the same as that for accessing a data store.

Environmental Interfaces in Software

An environmental interface must be refined and implemented in a driver with a mechanism used for triggering information transfer. Possible mechanisms include polling, interrupts and direct memory access. As other hardware interfacing mechanisms become available, these can be added. A polling interface can only serve as a valid environmental input to a system if it is triggered periodically by a timer. Both the DMA and interrupt mechanisms require a set-up phase before activation. It is assumed that the set-up phase is executed without explicitly stating it in the diagram.

Interrupts

Pre-emptive tasks are triggered by asynchronous interrupts that do not adhere to the RTDF paradigm. Interrupts are used for handling unpredictable timed events. If one assumes that

1) there is a maximum rate at which these interrupts can occur, and that the interrupt arrival rate is uniformly distributed at this maximum rate, and 2) that each interrupt computational time is smaller than that of the non-pre-emptively scheduled tasks, then scheduling condition (1) is a necessary condition for the complete task set. In fact, if the interrupt sources are assigned priorities proportional to their rates, the Rate Monotonic Analyses (RMA) algorithm guarantees that all the tasks in a worst case task set will meet their deadlines for a processor load of up to 69% [71]. If all the interrupt sources are multiples of the same clock, the RMA algorithm can gaurantee that all tasks will meet their deadlines for a processor load up to 100%.

The task blocking test of scheduling condition (2) can be approximated to be valid only for the EDF scheduled tasks as it relates to testing whether a non-pre-emptive EDF scheduled task has slack which is greater or equal to the worst case delay it can experience. As the pre-emptively scheduled tasks have shorter periods (faster rates), not any one of the interrupt driven tasks can result in a non-pre-emptive task missing a deadline. However, the pre-emptive task load represents an average load on the processor that consumes capacity not available to execute the non-pre-emptive task load.

Improving the Execution Overhead of a Software Kernel

Using the architecture on the existing processors in the SUNSAT microsatellite requires using an operating system kernel (RTX). As the hardware resources are limited, the kernel must perform its function with minimum processing load on the CPU. Various efforts have been made to estimate the contribution of kernel overhead to processor load and to improve it [8, 109, 30]. It is expected that this will be an on-going endeavour. This dissertation does not consider it further, other than to recommend that the more efficient the implementation the more processor resources are available for the application.

4.2.3 Kernel Interface

The current real-time executive (RTX) interface implemented by [30] in the Modula-2 language is shown in Figure 4.1. The number of functions is small and the function signatures are simple. Thus, it is simple to use this kernel and, even more important, the kernel implementation is simple to comprehend aiding in validating kernel functionality. A number of items to note for this particular implementation of the kernel:

1. The interface only makes provision for disjunctive input port semantics.
2. The ports in the interface are in fact channels which carry tokens without contents.
3. The signals are the token types that are sent on 'port channels'.
4. Only single shot timers are available as a period timer is a single shot timer that is set by a task (process) on each invocation.
5. A thread is the entry point of a task (process).

A previous software implementation of the RDF interface is available in Appendix D.3. The kernel implementation in the appendix makes provision for the different port semantics as described in this dissertation.

```
TYPE
  Thread                = PROCEDURE();
  Process               = POINTER TO ProcessRec;
  Channel               = POINTER TO ChannelRec;
  Timer                 = POINTER TO TimerRec;
  Message               = SYSTEM.ADDRESS;

PROCEDURE CreateProcess(     process_code   : TECP_API.ObjectCode;
PROCEDURE CreateChannel(     channel_code   : TECP_API.ObjectCode;
PROCEDURE CreateInputPort(   port_code      : TECP_API.ObjectCode;
PROCEDURE CreatePort(        port_code      : TECP_API.ObjectCode;
PROCEDURE CreateMailbox(     mailbox_code   : TECP_API.ObjectCode;

PROCEDURE Send(          channel_handle     : Channel;

PROCEDURE Signal(        port_handle        : Channel) : CARDINAL;
PROCEDURE SignalOnce(    port_handle        : Channel);
PROCEDURE ClearSignals(  port_handle        : Channel);
PROCEDURE EnqueueSignal( port_handle        : Channel) : CARDINAL;

PROCEDURE Receive(       VAR message        : Message;

PROCEDURE SetTimer(      timer_code         : TECP_API.ObjectCode;
PROCEDURE StopTimer(     timer_code         : TECP_API.ObjectCode;

PROCEDURE RunKernel;
PROCEDURE Shutdown;
PROCEDURE Done;
```

Figure 4.1: RTX kernel application programmers' interface

4.2.4 Processor Overload

It can happen that the assumptions on the behaviour of the environment are not valid leading to a processor overload. For a mission critical application such as a microsatellite, it is important that timely corrective action be taken. The processor overload can be managed by considering the application task sets. Each task is used by one or more applications on the satellite. Thus, each task has a set of one or more applications with which it is associated. Each application is assigned a priority based on its contribution to the mission criticality of the satellite. This priority has nothing to do with scheduling priority.

Action must be taken when a task is to be scheduled for its k'th execution, and the scheduler calculates that the expected time of completion is larger than the deadline. The scheduler then marks all tasks that only relate to the application with the lowest priority as not-runable. Thus, the task load on the processor is reduced. The task that is going to overrun is then scheduled only if it was not flagged as not-runable by the scheduler.

The load represented by each set of tasks related to each application is calculated in advance as an aid to assign priorities to each of the applications. Disabling any of the applications implies that a certain pre-determined load is removed from the overloaded processor. To implement the algorithm, each task data structure must include a bit vector for indicating membership to an application and a bit indicating whether it is runable or not. The *processor application vector* contains the list of currently enabled applications. If a bit in the processor application vector is set, any of the tasks associated with the application can execute when triggered.

For example, there are four tasks that can contribute to the execution of five different applications.

```
application vector 00000
task 1 01111
task 2 10011
task 3 00010
task 4 10010

application priorities: 1 3 2 4 5

scenarios:

1. task 1 over runs => disable application 5
2. task 2 over runs => disable application 4
```

The question arises when to re-instate an application again. This is accomplished by a flag for each application which indicates manual re-instatement or automatic re-instatement. If automatic, the scheduler checks the running value of processor utilisation $\mu_t = c_t/p_t$, and if $100 - \mu_t > \mu_p$, where μ_t is the current utilisation of the processor and μ_p is its application utilisation, then the application is re-instated (all tasks that are not-runable in that application are marked as runable). The additional functions that must be executed by the task manager include:

1. On task completion, set each deadline timer of each task triggered by the outgoing communication from the task that just executed.
2. On task completion, read the c_i value of the just completed task and update the stored $task(c_i)$ value.
3. Read the deadline of the just completed task and check if an overrun occurred with $now > storedDeadline_i$.
4. If an overrun did occur, perform disable application operation.
5. Update the running processor load total $c_t/p_t : \mu_t = (\mu_t + c_i/p_i)/p_i$ with the contribution of the last task o_i that executed.
6. Repeat:
   (a) Select next task to execute (EDF selection algorithm).
   (b) Calculate expected deadline, if too late perform disable application operation.
7. Start c_i timer to measure execution time for the task to start now.

Task sets for which lower quality of service is acceptable, such as less resolution in a graphics stream, slower response time to react to a trigger, less available channels for communication (i.e. the SUNSAT communication services), can be modelled with different levels of performance for each different situation. Instead of disabling a task from running, it is marked to run in a degrade mode. The algorithm for imprecise computation is not explored here further. Work in this area was performed by [98].

4.3 Simple Processors

Most existing microprocessors are tuned for maximum performance with little regard to real-time guarantees. Further more, the architectures of modern microprocessors permeates the architecture of programming languages. A notable exception is the Transputer [47] where the architecture of the processor was matched with the Occam programming language from the outset of the design. Instructions to support real-time scheduling was proposed for the Transputer [4].

The problem is that tuning a processor for maximum performance often ignores the cost of developing software (design, implement, test and validate) for such a processor. The development cost is further escalated when the architecture of the application is not supported by the architecture of the processor. It is a problem, because it limits the size and scope of software implementations of dependable systems or, even worse, it leads to accidents in mission critical systems where software is used without the possibility of rigorous validation.

Recent processor architectures are optimised for performance with little regard to supporting high level application architectures. An exception is the addition of an instruction set which supports the direct execution of Java byte codes in the latest ARM processor core [2]. The support of a direct translation of an intermediate language is a step in the right direction. Previous work to establish a processor architecture for specific application level safety and real-time requirements include the Viper processor [55], a dedicated safety critical processor [54] and the Real-Time processor [84, 76]. However, none of the features of these processors was incorporated into the commercially available processors.

This section will introduce a very simple task processor (Sproc) which provides for predictability of time properties. Predictable execution time and a hardware supported scheduling algorithm with predictable real-time performance forms the basis of the processor. These simple processors further match the design architecture tasks one-to-one. The simple task processor will be defined in terms of its core instruction set, event processing, and translation of a graphical language into the processor assembler language. The actual internal architecture of the central processing unit, of data and control paths are not specified. The simple task processor does not have a direct connection to the environment in which it operates. To this end, the Simple Input Output Processor (SIOP) is introduced to manage data input and output between a Sproc processor network and the environment. The Simple Input Output Processors are optimised for data synchronisation and transfer. A particular function of them is to convert irregular rate data from the environment to regular rate data in the RDF processor network.

The key contributions described in this section are:

1. A clear separation of the input/output, task execution and task management functions.
2. A processor (Sproc) optimised for the execution of hard real-time task sets.
3. A task management processor based on the EDF scheduling principle.
4. An integrated architecture of SIOPs and Sprocs which offers tailor made performance and guarantees for the temporal properties of task sets.

4.4 Simple Task Processor

4.4.1 Processor Characteristics

The simple processor is an event driven processor which supports non-pre-emptive process execution, only. This has the following significant implications:

1. No stack is required due to non-pre-emptivity.
2. No explicit task list is required due to the event triggered activation of all tasks.

The task processor is a RISC processor with seven instructions of which one is the *END* instruction. These instructions are adequate to write the control flow of most programs. The processor's complexity is low as it is expected to execute as little as one task. The core instruction set can be extended by application specific instructions to achieve a particular performance level. The execution of tasks within a processor is initiated by trigger events. Once an event has occurred and the appropriate task was selected, it executes until completion. All trigger events which include arriving messages, elapsed timers and external events, are buffered one level deep. This buffering takes place on all trigger events in parallel, therefore the task processor is not interrupted to execute any preemptive tasks.

Tasks are allocated to a Sproc as a result of a schedulability test of a task set. This analysis ensures that the processors operate in the resource adequate mode only, i.e. the processors have enough resources available to execute an allocated task set.

The processor has built-in inter-processor communication functionality. Any number of input and output communication ports can be attached to a task processor. Reliable communication is assumed between processors. This is feasible with the processors in close proximity. For a distributed computing application a simple processor can be programmed for a high level communication function which operates on a channel with errors.

The simplicity of the processor makes it suitable for implementation on an FPGA with additional resources remaining on the FPGA. The additional resources available on an FPGA allow to implement application specific interfaces, special instructions or more simple task processors.

See Figure 4.2 for the basic structure of the processor that consists of the following elements:

1. task processor,
2. memory interfaces,
3. input and output communication ports,
4. timers, and
5. an EDF (Earliest Deadline First) event processor.

4.4.2 Task Processor

This section introduces a simple task processor to show how the RDF notation at a fine grain level can be efficiently translated into the instruction set of a typical processor. The task

processor is by no means optimised other than to be very simple and to be able to support the execution of one task in an RDF system. The task processor instruction set supports the final translation from RDF into a program.

The task processor is an 8 bit RISC processor executing only 7 instructions. All instructions are coded in a 16 bit word of which the 4 most significant bits are used for the instruction and the 12 least significant bits for an address. The entry point to each task on the task processor is mapped to a corresponding trigger event. Inter- and intra-processor communication is carried out by sending messages. The arrival of a message triggers the execution of the receiving task. The task processor consists of the following elements.

Accumulator to perform addition and subtraction on eight bits with a single carry bit. Instructions on the single accumulator include a memory reference for the other source operand.

Program Counter pointing to the current program code. The program counter is incremented or replaced by a new address based on a branching instruction. The program counter is 12 bits wide.

Instruction Register holding the instruction to be decoded and executed. The instructions are:

Load accumulator from memory

Store accumulator to memory

Load plus Carry accumulator from memory

Add memory to accumulator with carry

Subtract memory from accumulator minus carry

Jump to new location on no carry

End current task and return to event waiting status

Address Register holding an address pointing to data memory. The address register is 12 bits wide.

4.4.3 Memory Organisation

The program memory and the data memory are located in two separate areas. The program memory is either ROM or Flash based. Both these memory types provide memory over-write protection without the additional memory protection schemes found in conventional processors. The program memory is organised to support the event driven nature of the tasks. The first number of 16 bit words is reserved for the event mapping table. A JMP instruction with the address to the appropriate task entry point (function start) for each trigger event is stored here.

The program code of each task is stored in the following order. The first word contains the number of program bytes. The second set of words contains the maximum rate at which the task can accept trigger events on this processor. The inverse is the expected worst case execution time (WCET) of the task. The third entry is the maximum rate at which the task can generate output messages as was derived from system analysis. Thereafter follows the code. The tasks can be individually replaced for a Flash memory based code store.

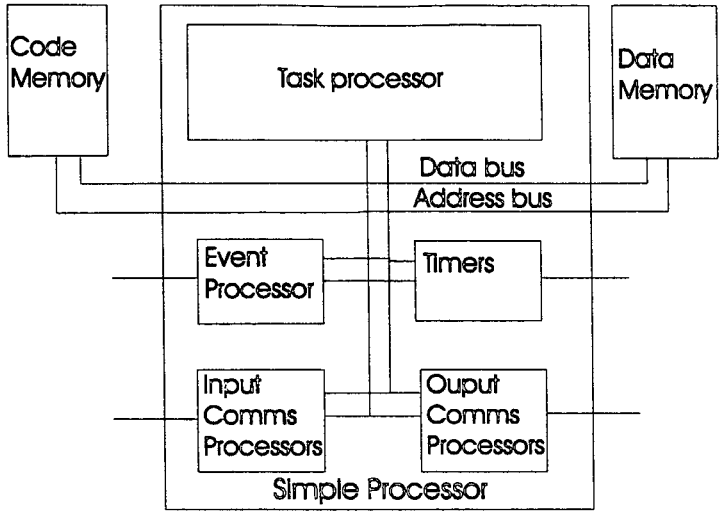


Figure 4.2: Basic concept of the architecture

The data memory is organised on a per task basis. The first number of data memory places is reserved for accessing the trigger events, the hardware communication ports, and the hardware timers. Thereafter, the data memory is organised on a per task basis. Each task data area consists of enough space for all the internal input and output ports associated with that task. It follows the area for all task variables.

4.4.4 Communication Ports

The behaviour of the communication ports between processors and internal to a processor are both predictable and deterministic. The data ports are unidirectional. A synchronised port or data return port provides for a return value to be sent.

External ports

The simplest communication interface between processors is a unidirectional, three signal bus for synchronous serial communication. The output port transmits a complete message by making the receive enable line true, gates the transmit clock onto the clock line, and sends the data serially on the data line. The number of data lines can be extended to provide an increase in bandwidth as required.

A port has storage space for one complete message. This allows the port to operate in parallel with the task processor. The port operates as follow. During the execution of a task, all the values which form part of a message to be transmitted are written into the output port memory. The last action the task undertakes is to enable the port communication engine to read these values and transmit them in a block to the input port of the receiving task. The receiving port accepts the whole message before setting the trigger event associated with receiving a message. The receiving task is now enabled for execution. The transmitting task can write to multiple ports and enable the output port engines in close succession for a parallel transfer of its messages. The transmitting processor waits until all messages have been transmitted before returning to the processor idle state awaiting an event. The time it takes for a message to transfer across a communication link must be included in the execution time for the transmitting task.

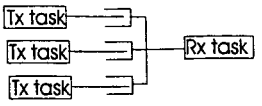


Figure 4.3: Connecting multiple output ports to one input port

Multiple output ports connected to a single input port correspond to a logical input port in RDF. However, in the hardware implementation, each output port writes into its own FIFO memory and the input port has a media scheduler which provides a transmit enable signal to read the contents of each FIFO memory in turn. As the output messages are buffered, all messages are retrieved one at a time until all are consumed and processed. The receiving task processor is executing at the sum of the transmitter rates. See Figure 4.3 for a representation of the multiple output port to one input port architecture.

Internal Ports

The tasks on the same processor communicate by writing into the input port message space of the receiving task. This is without any side effects due to the non-pre-emptive interleaving of the tasks on a single processor. When writing into the input port memory space, the input port event trigger bit is set. The event trigger bit signals the scheduler that the receiving task is runnable.

Multiple internal output ports connected to a common internal input port follow the same architecture as the external multiple output ports except that the message memory is associated with the input port. This is because of the non-pre-emptive task execution on a Sproc. Each task adds its message to the input port message space that it is connected to and sets the event trigger bit for that input port.

Synchronised Ports

A synchronised port or data return port is a bi-directional port to which a message is sent on the one unidirectional port and at which a reply message is received on the returning unidirectional port. The time for communication and remote function execution must be added to the task requesting the execution.

4.4.5 Timers

Each processor contains a number of single shot and a number of periodic timers. The number of timers depends on the task set on that processor. The elapse of any of the timers leads to the setting of the associated bit in the event trigger register. A periodic timer only needs to be loaded once with a period. Single shot timers are for measuring ad hoc timing periods and must be re-loaded for each activation. In addition to the set of timers, a global time register is kept up to date in each event processor. The time source for this register is a centrally distributed clock if the system is in a small geographic area. For a distributed system, GPS receivers are suggested to receive current time.

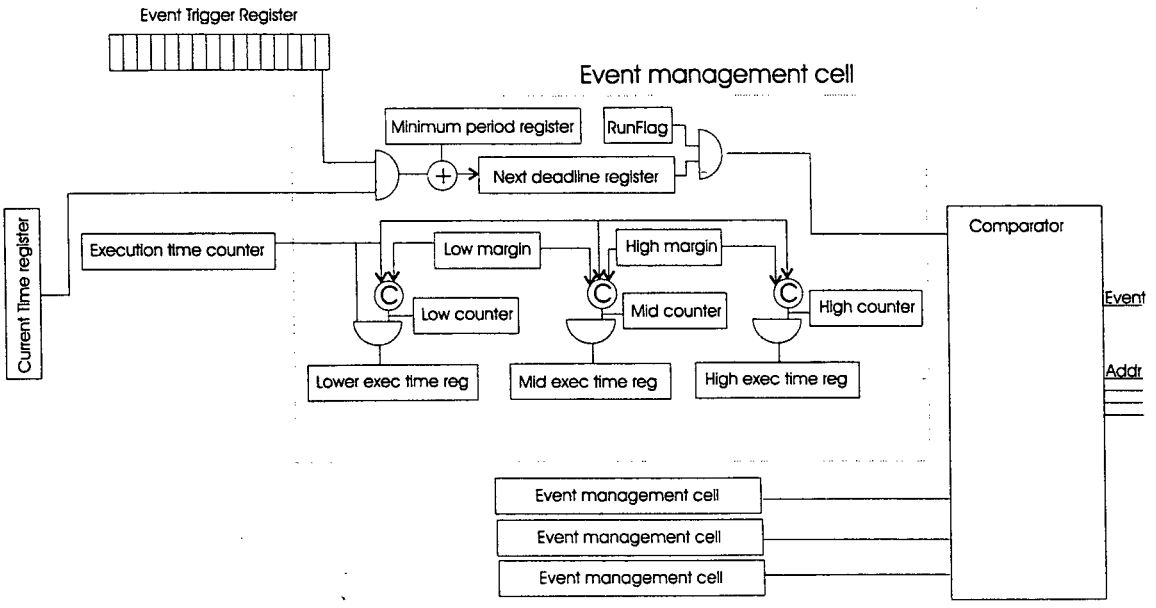


Figure 4.4: Event processor

4.4.6 Event Processor

A bit in the event trigger register represents each event source. The event processor has an event management cell with the following registers for each input port and timer in the system.

Next deadline register is loaded with the value of the next deadline when an event trigger for this input port is received. No deadline is indicated as a vector of binary ones.

Minimum period register contains the value p_i which is used to calculate the next deadline when a message is received on the associated input port.

Execution time registers Each of these 3 registers is associated with a counter to count the number of times the execution time fell within a particular range. Two margin registers determine the three ranges for which each of the computation time registers is used.

Each of the *next deadline registers* serves as an input for a comparator which determines the smallest value, i.e. the next closest task deadline. The number of the event (task) is output on the address bus with an event line to the task processor for prompting it to read the instruction in the code memory at the address on the address bus. Associated with the number of events is the same number of *JMPC* instructions stored in the first words of the code memory, called the event table. An *JMPC* instruction forces a jump to an appropriate task entry point. See Figure 4.4 for the architecture of the event processor.

On completion of the current task with an *END* instruction, the task processor replaces the program counter with zero and goes into an idle mode. The task processor signals the event processor that the current task is completed. At the next event, the task processor is signalled with an event signal with the number of the event on the address bus.

The event reaction latency time is guaranteed to be one instruction execution cycle. The event processor arbitrates between multiple events and only activates one task on the task

processor. While a task is executing, all incoming events are registered by the event processor and arbitrated between at completion of the current task.

With the dual processor system there is an opportunity for off-line scheduling decisions to be made in the event processor. However, as the next task is always selected on an Earliest Deadline First basis and the task could be a result of the current execution, it is sufficient to make a scheduling decision on completion of the current task.

4.4.7 Core Instruction Set

The core instruction set is introduced to show how fine grain RDF programming constructs can be translated to execute on the simple processor. It consists of seven instructions. There is only one addressing mode and a single accumulator. All instructions take three cycles to execute, and just one argument that is an address. The following is a description of each instruction.

Load The load instruction reads a data value from memory specified by the direct address and stores it into the accumulator. Constants are stored in the data memory at known locations by the compiler. The load instruction clears the carry flag.

Store The store instruction writes the accumulator to memory specified by the direct address.

Load plus Carry In order to operate on operands longer than the accumulator, this instruction allows the carry flag from a previous operation to be added to the next part of a longer operand.

Add The add instruction reads an operand from memory, adds the value to the accumulator with the carry flag, and stores the result in the accumulator.

Subtract The subtract instruction reads a value from the specified memory location, subtracts it from the accumulator, and stores the result in the accumulator.

Jump The jump instruction is used for conditional branching. The instruction tests the Carry flag and, if not set, jumps to the absolute address supplied as argument.

End The end instruction indicates the completion of the current task and returns the processor to an event waiting status.

Instruction	Mnemonic	Execution
LoaD Accumulator	LDA addr	A <- mem[AR]
STore Accumulator	STA addr	mem[AR] <- A
LoaD Accumulator + Carry	LDAC addr	A <- mem[AR] + C
ADD memory to accumulator + Carry	ADDC addr	A <- A + mem[AR]
SUB memory from accumulator - Carry	SUBC addr	A <- A - mem[AR]
Jump to new address		

```
on No Carry          JNC addr   PC <- AR (if C=0)

END the current      END addr   PC <- 0
task

Where PC = program counter
      AR = address register loaded with addr in each case
      A  = accumulator

Ports are accessed by using the following record:
PortName.Val[0..N-1] are the port values
PortName.e is the event set signal for intra-processor communication
      and the transmit port signal for inter-processor communication
```

4.4.8 Compilation into Core Instruction Set

It is acknowledged [34] that real-time languages should be reliable, foster predictable system behaviour, and be analysable for deadline matching during compile time. Such a visual data flow language RDF has been introduced in Chapter 3 and was first described in [79]. This section will show how the RDF language compiles directly to the presented architecture without leaving any semantic gaps.

Branching and Alternatives

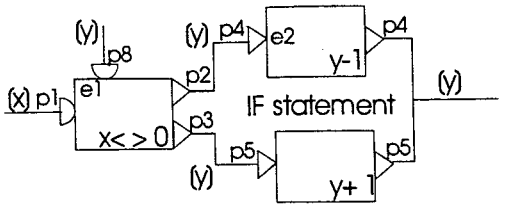
A conditional branch is performed by the conditional jump instruction. Branching based on a number of alternatives is performed by a set of conditional jump instructions. See Figure 4.5 for the RDF representation and the corresponding Sproc code. The tasks with entry points e1:, e2: and e3: could execute on the same processor or on three different processors.

The visual language follows the data flow paradigm. Note, however, that there are two types of ports. The triangular arrow head denotes disjunctive semantics, i.e. any one of the ports will create a trigger which will start the associated task. The round arrow head denotes conjunctive semantics, i.e. each of the ports must have one message received before the appropriate task is activated.

Procedure Calls

Procedure calls are not explicitly supported in the visual language. Rather all re-usable software components should be packaged as tasks that are activated in a data flow manner with trigger events. However, it is possible and sometimes preferable to use the subroutine paradigm for the actual implementation of software.

Procedures can be implemented as remote procedure calls with synchronised data channels. The data return port in RDF translates to a combined input and output port on both the sending task and the receiving task. The sending task sends a message requesting a specific function to be executed and, then, *waits* for the return message from the task it requested the function from. This same synchronised data channel is the mechanism used to access data stores. See Figure 4.6 for the graphical representation of the construct and the assembler code for the simple processors. Note that e1: and e3: are two different entry points for the task




```
e1:     LDA     p1.val[0] ;IF x < 0
        SUBC    zero      ; constant in code
        JMPNC   else:
        LDA     p8.val[0]
        STA     p2.val[0]
        STA     p2.e
        END
else:   LDA     p8.val[0]  ;ELSE
        STA     p3.val[0]
        STA     p3.e
        END

e2:     LDA     p4.val[0]
        SUBC    one
        STA     p6.val[0]
        STA     p6.e
        END

e3: ....
```

Figure 4.5: IF statement

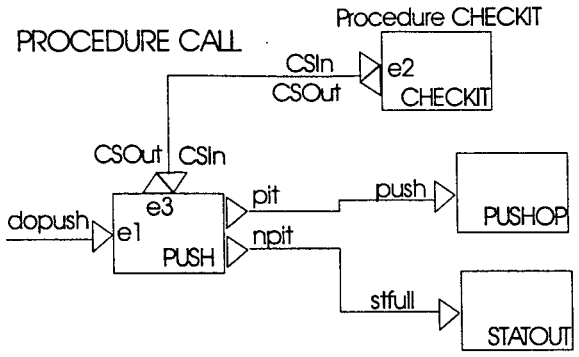
PUSH, while e2: is the entry point for the task *CHECKIT*.

Loops

Loops are implemented with conditional jump instructions. In order to force predictable timing behaviour, a *while* loop is always augmented by a single shot timer. The timer is set on loop entrance to the maximum execution time allowed for the loop. Once the timer elapses the loop terminates and transmits the result as calculated so far. See Figure 4.7 for a graphical representation and the listing below for example code. Note that e1: and e2: share some code. Furthermore, task entry point e3: is triggered by the single shot timer which timed out. *WhileTimeLimit* is a constant supplied by the programmer or the compiler. The tasks with entry points e1: to e3: would typically reside on one processor.

4.4.9 Simple Input Output Processor

Each Simple Input Output Processor (SIOP) in a network of simple processors is dedicated to a particular input or output task. These SIOP processors function as synchronising agents between the irregular/bursty data rate of the environment and the regular data rates in an Sproc network. See Figure 4.8 for an example of a processor network with co-operating SIOPs and Sprocs.



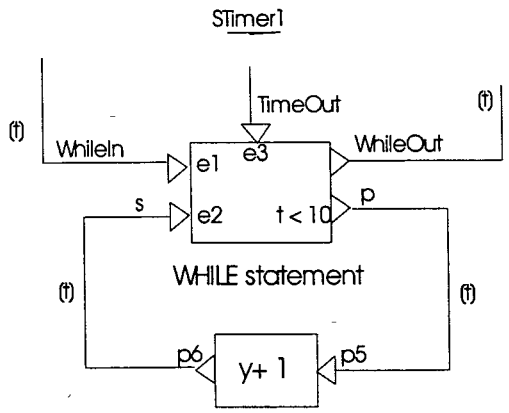

```
e1:     LDA     CheckStatusC
        STA     CSOut.val[0]
        STA     CSOut.e
        END

e2:     LDA     CSIn.val[0]
        SUBC    CheckStatusC
        JMPNC   ....
        ........
        STA     CSOut.val[0]
        STA     CSOut.e
        END

e3:     LDA     CSIn.val[0]
        SUBC    Full
        JMPNC   ....
        ......
        LDA     doit
        STA     pit.val[0]
        STA     pit.e
        END
```

Figure 4.6: Procedure call

The external data interfaces supported in conventional processors include

1. recognising events
   - by polling or
   - by interrupts, and
2. data transfer mechanisms:
   - memory mapped input/output,
   - DMA mechanisms,
   - memory bank switching, and
   - processor specific communication link interfaces.

Many of these mechanisms are viewed as inefficient in terms of processor utilisation (i.e. polling) on conventional processors. Other external data interfaces are considered as temporally unpredictable (i.e. cycle stealing DMA) although processor efficient. In a dedicated Simple IO




```
e1:     LDA     WhileTimeLimit
        STA     STimer1
        LDA     false
        STA     timeOutFlag
        LDA     WhileIn.val[0]
        STA     t
e2:     LDA     timeOutFlag
        SUBC    false
        JMPNC   endwh:
        LDA     t
        SUBC    ten
        JMPNC   endwh:
        LDA     t
        STA     p.val[0]
        STA     p.e
        END
endwh:  LDA     t
        STA     WhileOut.val[0]
        STA     WhileOut.e
        END

e3:     LDA     true
        STA     timeOutFlag
        END
```

Figure 4.7: Timed while statement

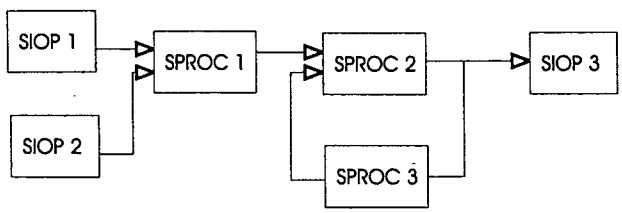


Figure 4.8: A network of simple input/output and task processors

Processor a different set of criteria is applicable as the processor is dedicated to IO functions. Here polling may not be inefficient, but appropriate. A dedicated SIOP has the following advantages over the input and output functionality provided in general purpose processors (GPP), because a GPP is optimised for general purpose applications while the SIOP is optimised for IO.

1. The silicon on a SIOP should be smaller than the equivalent silicon on a GPP, because less other functions have to be accounted for in an interface.
2. It is expected that an SIO should have a shorter start-up latency and be able to sustain data transfer at the maximum data bandwidth as the input/output operations are not interleaved with other operations on the processor.
3. The SIOP data bandwidth can be optimised for a particular application while the GPP data bandwidth is fixed and dimensioned for the code and data memory data paths.
4. A particular advantage is that a system can be configured with as many SIOPs as are required for a particular parallel processing input/output system. This ensures that bottlenecks in the input/output system can be avoided.

This section proceeds to describe a general input/output interface for the SIOP. The interfaces between the SIOP and the Sprocs in a network are the input and output communication processors found on the simple task processors.

Input/Output Transfer Interface

A SIOP environmental interface is based on a DMA engine of which the read (RD) and write (WR) functions run in parallel in different address spaces. A SIOP is configured at start-up with the characteristics of the communication over each interface. The SIOP executes this input/output cycle on data entering and leaving the processor network. In particular, each DMA engine function consists of

1. an address register,
2. an address function ALU which can increment, decrement or not alter the address for each cycle (The address function ALU could also be expanded to execute complex address patterns such as for a higher radix FFT operation.),
3. a data transfer register, and
4. a controller generating an RD and/or WR and other control signals depending on the direction of data transfer.

Each DMA engine has the appropriate number of address lines for its address space, and a number of data lines suited to the data bandwidth required. The dimensioning of the data bandwidth is described in the next section. The RD- and WR-DMA engines work in parallel on independent data streams. Figure 4.9 shows the internal architecture of an input processing SIOP with independent RD- and WR-DMA engines.

If one compares the above architecture with conventional processor IO, then the following analogies are seen. The particular mechanisms which are available (with the conventional name in brackets) include:

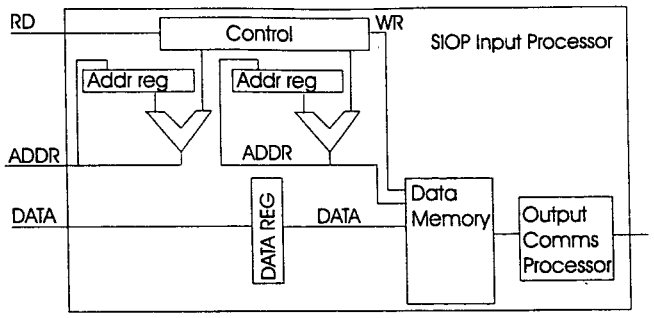


Figure 4.9: Simple input output processor architecture

1. one or more byte transfer through the data transfer register between two memory locations (memory mapped IO), and
2. moving a block of data between source and destination (block move instruction, DMA).

Data can be transferred after a signal is given that the data are ready to be moved, which can done in one of the following ways (with the conventional name in brackets):

1. regular interrogation of a ready signal (polling), or
2. event signalling from outside to indicate readiness (interrupts).

The performance of these mechanisms was evaluated for a conventional processor, the Transputer [47], in [80] and will not be further discussed here.

Dimensioning the Input/Output Interface

Two parameters determine the dimensions of the data bus and the memory of a SIOP:

1. the data rate which determines the data bus width, and
2. the memory buffer size which is determined by how much data is processed per unit of execution cycle and the data rate.

The bandwidth required and the size of the data block being processed determines the buffer size. The buffer size is given by the following expression:

$$MemoryBufferSize = (DataRate * BlockExecutionTime)$$

To choose the appropriate width of the IO data bus the following test can be performed:

$$F = BandwidthRequired/BandwidthAvailable$$

If the result is $F <= 1$, then the bandwidth is acceptable, if the result is $F > 1$, then the incoming bandwidth is larger than the data bus bandwidth and the width of the data bus has to be increased.

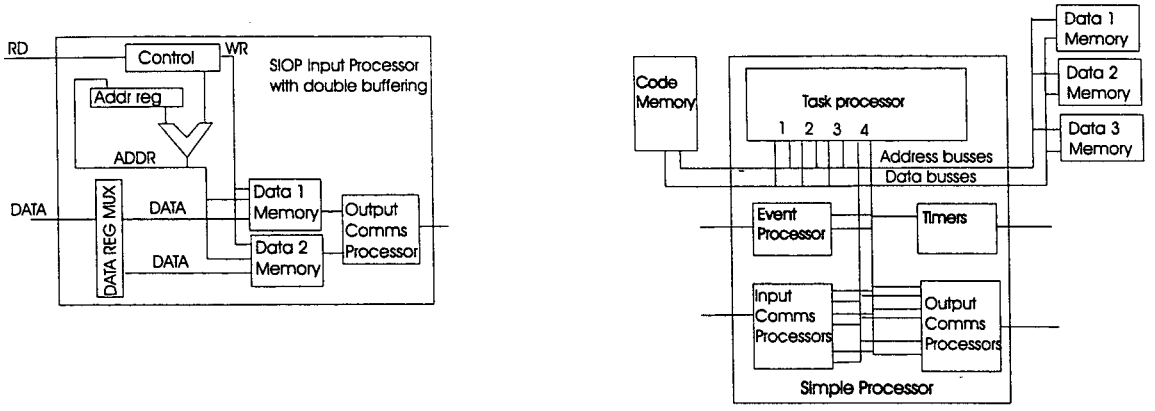


Figure 4.10: Increasing communication bandwidth for input and output

It is assumed that the SIOPs are integrated on the same silicon as the Sprocs and that there are no errors in the data streams due to the communication interfaces. If a system requires a fault tolerant communication interface, it implies a more complex IO building block which is not covered in this discussion.

Increasing the IO Bandwidth

The following mechanisms can be used in the SIO processor and in the Sproc to increase the bandwidth of data transfer. They are in order of complexity and transfer rate.

1. One memory block with shared address bus and data bus,
2. double buffering allowing transferring of data in one memory block and processing on the other memory block, and
3. triple buffering with input, processing and output executed in parallel on three independent memory banks.

The above mechanisms are applicable between an environment and a Sproc processor network and inside a Sproc processor network. See Figure 4.10 for two processor architecture options. The SIOP on the left contains a double buffer which means that the Output Communication Processor and the incoming data stream can operate in parallel. The Sproc on the right hand side of the figure has 3 memory banks which means that the Input Communication Processor, the Task Processor and the Output Communication Processor can operate in parallel on three different data sets.

4.4.10 Conclusion on Simple Processor

This section presented a simple processor architecture with predictable timing behaviour. The simple task processor has a simple event triggered task activation structure with a RISC core instruction set. The overall architecture was described in detail, while the implementation details were not shown. The simplicity of the processor leads to other advantages as well, which include packaging more than one processor on the same silicon, the ability to add custom

interface hardware, and the ability to add application specific instructions to the instruction set. The processor is simple, and requires no stack to support the nesting of procedures or the pre-emption of tasks. It has, however, been shown how to implement on the architecture the subroutines found in control flow languages. Furthermore, due to the design process tasks are allocated in such a manner that non-pre-emptive execution is the rule. The simple processor architecture is a suitable platform for data flow languages. Only the finest grain of data flow programming would not be supported efficiently on the architecture. The simple processor can be used as a single processor executing hard-real time application software. The simple processor architecture is comparable in complexity to an instruction set enhanceable processor for implementation on an FPGA [115]. It is, however, much simpler than the proposed processor in [76].

4.5 Implementation of RDF Architecture with Dual Processors

This section examines a case study, viz. the application of the RDF architecture in a microsatellite composed of hardware building blocks. A next generation microsatellite architecture is described [81] based on experience with the SUNSAT microsatellite. As background, the high level requirements of the software on the SUNSAT microsatellite can be found in Appendix B. The section begins with a description of a generic dual processor implementation of a task node in the RDF architecture. The next generation microsatellite architecture requires some background on mechanical structures and the localisation of control in current microsatellites. Both these aspects are influenced by the use of the RDF computational architecture.

4.5.1 Architecture of a Generic Dual Processor Component

A building block on which one or more RDF tasks can be implemented consists of two processors. One processor performs a set of standard functions including scheduling, receiving messages, and sending messages that are common to all building blocks. The second processor is optimised to perform a particular set of functions. The functions could be as diverse as a star imager or an AX.25 protocol engine. One could consider the building block as a combined (simple) processor and (simple) input/output processor. The input/output processor in this case has built-in processing. In some cases (imager data) the data rate being managed is significantly higher than the inter-node communication system bandwidth. Then, the input/output processor supports its own application specific ports.

Each building block has a standard interface that consists of a number of serial ports and any number of application specific ports, i.e. high speed audio or antenna ports. All building blocks support the same protocol over their serial links which can be extended according to the specific function that the block performs. Each building block as in Figure 4.11 is responsible for collecting its own telemetry and decode and actuate its own telecommands. Each building block has a least two identical serial ports over which commands are received and replies sent. Additional dual serial ports with a higher communication rate can be included to support higher bandwidth inter-node communication.

The generic building block consists of a processor which includes on the same package a boot-loader in PROM, FLASH memory, RAM memory, AtoD, DtoA, timers and 2 serial ports. Programs are stored in the FLASH memory, and the RAM memory is protected by a hardware EDAC circuitry.

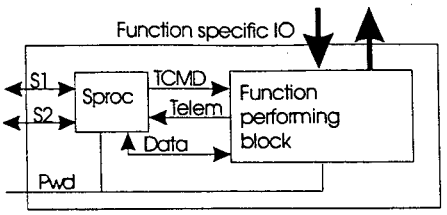


Figure 4.11: Generic building block architecture

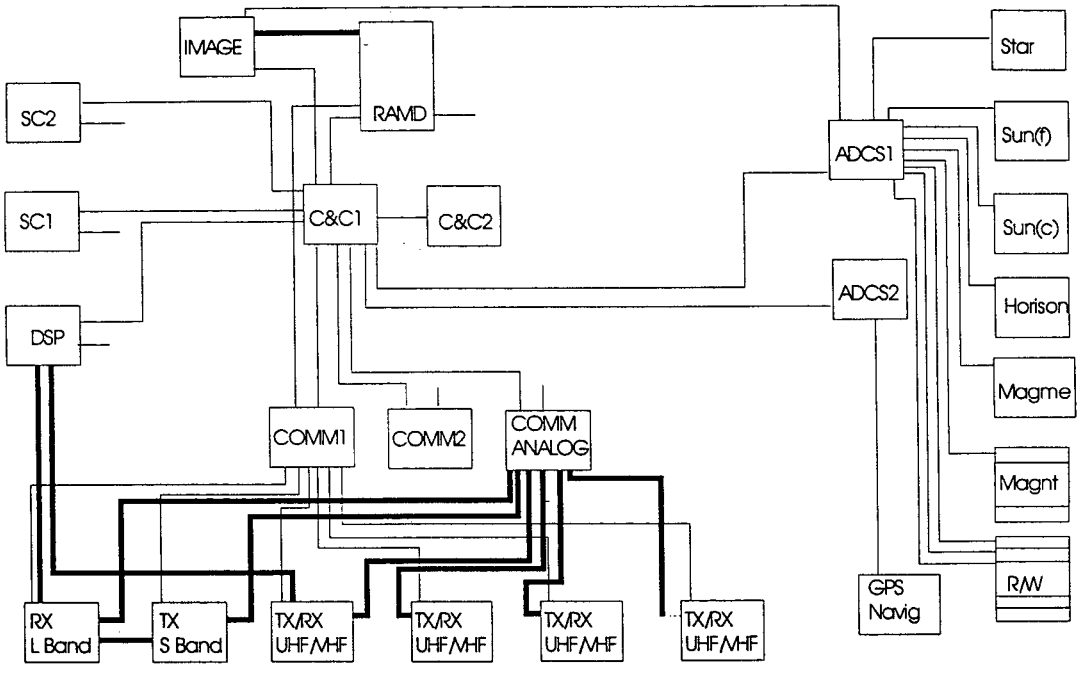


Figure 4.12: Complete next generation microsatellite architecture

4.5.2 Background on Microspacecraft

Microsatellites can be classified according to mechanical structure and localisation of control. The extent to which spacecraft components are optimised determines the amount of flexibility of a specific spacecraft component for application in multiple missions. It is recognised that spacecraft components should not be optimised to allow for late changes and flexible mission profiles. For example, the structure of a satellite should be able to support more than the minimum mass for the current mission.

Location of Control

The localisation of control can be central or distributed. Satellite architectures with central control are characterised by a single hardware redundant node of control [113] for the complete satellite, e.g. UOSats [25]. Distributed control is found where each system component supports its own low level functions completely, providing a high level interface, e.g. Tubsats[12]. Building blocks with distributed control in mechanically separate packaging lead to re-useable components which require the minimum of centralised control. This also has an additional ad-

vantage that it enhances the fault tolerance if these autonomous building blocks are replicated.

4.5.3 Next Generation Microsatellite Architecture

A next generation microsatellite architecture is proposed which builds upon the advantages of mechanically separate building blocks with as much autonomy of control as possible within each building block. In order to facilitate a comparison, the SUNSAT satellite will be used as a case study for the new system architecture. See Figure 4.12 for the complete satellite in the new architecture (note that not all communication links are shown for the second C&C processor).

The architecture consists of building blocks that are interconnected in star topology or bus topology networks. Each building block has its own processor and can perform its complete function, i.e. a reaction wheel. Each building block must adhere to the RTDF paradigm, viz. a task receiving messages/tokens must consume and process them at the same rate or faster than sent by the transmitting task. The physical layer for communication between the building blocks could be a point to point standard such as RS-232 or a bus standard such as RS-485. In addition, the protocol of the messages on the communication channels must be agreed upon between all building blocks. The particular content of the messages depends on the processing that must be performed by each building block.

Each of the subsystems will now be described in turn. See the high level software specifications in Appendix B for a specification of the functions required of the system.

4.5.4 Communication Payload

The communication payload is split across various building blocks according to the functions required. The primary receiver building block supports the lowest level protocol which is a frame receive operation for the AX.25 protocol. The lower part of the block diagram in Figure 4.13 contains one RF receiver, a demodulator, USART and a processor with extra RAM for buffering frames.

The primary transmitter building block supports the lowest level protocol which is a frame transmit operation for the AX.25 protocol. The upper part of the block diagram in Figure 4.13 contains one RF transmitter, a modulator, USART and a processor with extra RAM for buffering frames. Combining the receive and transmit functions into one building block leads to the design in Figure 4.13. In both cases the processor executes only the frame receive or frame transmit function in addition to its telemetry and telecommand functions. All packets received with satellite telecommand information are passed to the next higher level processor which is the command and control processor.

The high level protocol functions are performed by dedicated communication processors. These processors are duplicated for redundancy and multiple channel operation. Both processors (PACSAT comms and MUX comms) in Figure 4.14 have additional RAM for data storage or have access to the RAMDisk.

Analogue communication functions such as switching the audio and IF busses are supported with an analogue communication processor which has access to all the analogue data streams from the receivers and transmitters. Any source can be interconnected to any sink through the analogue communication processor (Analogue comms) in Figure 4.14. Analogue processing

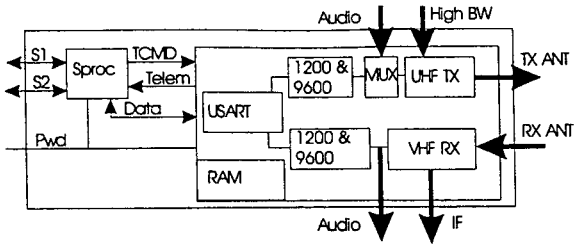


Figure 4.13: Communication building block

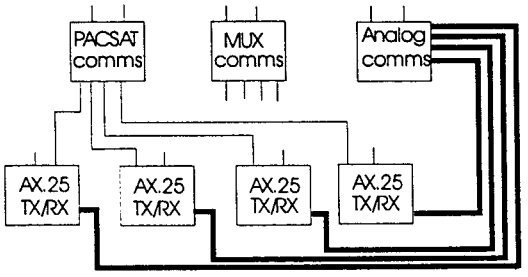


Figure 4.14: Communication system composition

such as high speed modulation, speech recording and play back (a parrot function) is performed by a digital signal processing (DSP) building block, that has direct access to the audio bus and intermediate frequency (IF) bus switched through the communication processors. An alternative for analogue processing is to include a DSP function into each communication block. Except for the UHF TX and VHF RX radio frequency front end blocks in Figure 4.13, the rest of the processing can be performed by the digital signal processor. This is in fact the recommended hardware configuration.

4.5.5 Imager Payload

The imager payload and the RAMDisk subsystem are completely separated from each other. The imager function block contains all required timing generation, analogue to digital conversion and serialising circuitry. See Figure 4.15 for the imager building block.

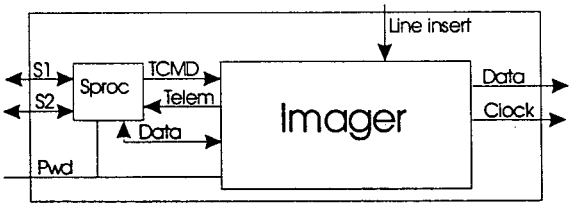


Figure 4.15: Imager building block and interface definition

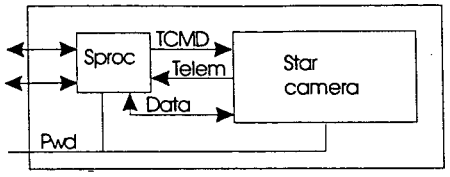


Figure 4.16: Generic sensor and actuator block

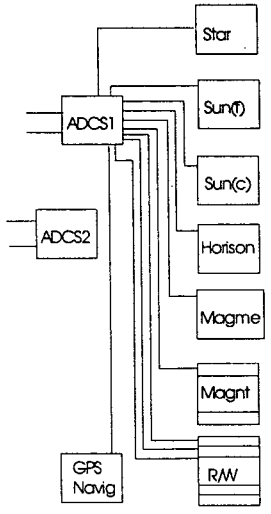


Figure 4.17: ADCS system architecture based on building blocks

4.5.6 Attitude Determination and Control System

The Attitude Determination and Control System (ADCS) is the single most complex system in the satellite in terms of the number of elements that operate together. The ADCS consists of a number of sensors which provide reference information, a number of actuators which with the satellite attitude can be changed, and a processor. The processor validates sensor information, and uses the validated sensor values to update a model of the satellite attitude. The processor further executes control algorithms which generate actuation values based on the current satellite attitude set point. All sensors and actuators lend themselves to operating as RDF building blocks. Each sensor and actuator block (Figure 4.16) executes the following functions: hardware input/output, data calibration, and modelling the relevant parameter, e.g. modelling the solar orbit for the sun sensor.

The number of sensors and actuators interconnected with the ADCS processors can be changed in a flexible manner. The ADCS with doubly redundant processors and a star interconnection topology is shown in Figure 4.17. The processing requirements for the estimation and control algorithms are such that it is feasible to provide multiple simple processors to execute these functions as in Figure 4.18. Existing efficient implementations for a single traditional processor makes it questionable whether to split the functions onto a number of simple processors.

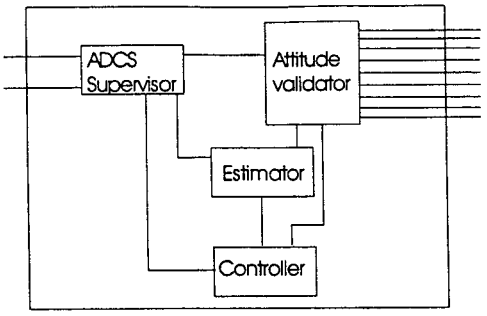


Figure 4.18: Internal composition of ADCS macro block

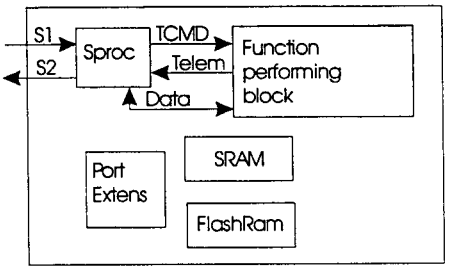


Figure 4.19: Extended generic building block

4.5.7 Command and Control Processor

The command and control processor is ultimately in control of the complete satellite. The relationship of the command and control processors (C&C1 and C&C2) with some of the other sub-systems is shown in Figure 4.20. The functions of a command and control processor are:

1. Collect telemetry, assemble telemetry frames, store telemetry in a file system for later retrieval, and transmit the collected telemetry to the communication controller processors.
2. Receive telecommand messages via the communication controller processors, validate commands against the satellite state, and send commands to appropriate subsystems.
3. Maintain the overall satellite schedule and send commands when specified by the diary.
4. Monitor satellite health and take appropriate action in case of an emergency.

The extensions to a generic building block which allow larger data memory and more ports required for interconnection are shown in Figure 4.19.

4.5.8 Influence on Groundstations

The same characteristics of low power consumption, small volume and reliable operation are required from the user terminal groundstations that interact with a satellite. Therefore, the same building blocks as in a satellite can be used for a portable groundstations. Satellite control groundstations can largely be constructed with the RDF building blocks identified for satellites.

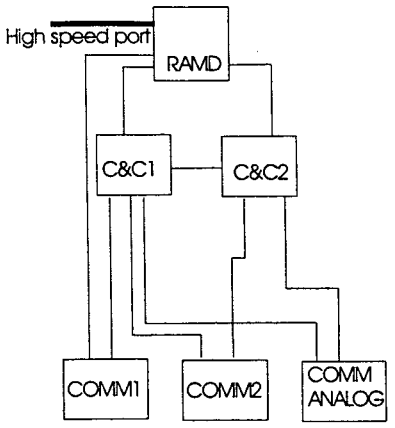


Figure 4.20: Relation of Command and Control block to architecture

4.5.9 Evaluation

Having all subsystems available in the RDF building block architecture will result in a fast response to construct satellites for new missions and to adapt an existing satellite to new requirements. Mission flexibility is important in order to respond quickly to changing launch opportunities which is characteristic for microsatellites.

The SUNSAT satellite is tray based, and for a building block version of SUNSAT one would expect a larger and heavier satellite but the following items are to be noted:

1. The telecommand and telemetry trays are eliminated.
2. The modems are piggybacked onto the RF building blocks.
3. The large and complex on-board computers [5] have been replaced by simple multi-port processors.
4. The harness is significantly simpler and smaller.

The disadvantages are that each of the autonomous subsystems has gained a little extra mass and additional power consumption. Modern processors and supporting components are, however, capable to go into a stand-by mode when not operating which should minimise additional power consumption. Another concern with the RDF building block architecture is the reliance on microprocessors to perform the Sproc function in Figure 4.11. Experience in Low Earth Orbit has indicated that microcontrollers such as the 8031 [48] and H-8 [42] are reliable. An advantage of this implementation is that, if a building block with a specific reliability is required, space qualified components can be used on that specific building block.

4.6 SUNSAT Implementation

The RDF architecture was used for the high level design of the flight software for SUNSAT [91]. The computational architecture was implemented as the RTX kernel [108] on an 80C188EC processor [50], an 80386EX processor [49] and a Transputer T800 processor [47]. The goal

was to achieve the same computational architecture on all three processors on SUNSAT. This is important as the software engineering effort to create the application software is significant. Currently the 80C188EC version of the kernel is in operation as well as a PC compatible version of it. The 80386EX and T800 implementations can, at best, be considered as prototypes and are not operational. The 80386EX has not been commissioned to run, and the Modula-2 compiler for the T800 has so many deficiencies that it has not enabled a reliable implementation of the kernel on the T800 to date. Currently the flight software on the 80C188EC processor and the groundstation software on a PC execute in the RDF architecture on the RTX kernel. The kernel provides the architecture abstraction layer supporting the RDF real-time computational architecture.

The traditional software engineering approach of implement and test was used to create the flight software. None of the realisability analysis, feasibility analysis or determination of scheduling rates supported by RDF was used in the SUNSAT software development. Furthermore, the tasks were implemented in Modula-2 after a high level task decomposition. The advantages of working within the computational architecture as implemented on the SUNSAT project are:

1. The kernel offers the same task computational architecture across the different hardware processors enabling a shorter development cycle.
2. The kernel provides a clear indication when a deadline was missed, assisting in fault finding.
3. An unambiguous communication language exists at the task level between design engineers.

In Chapter 5 the modelling and analysis supported in RDF are explored in the context of typical microsatellite flight software. It is shown that it is possible to determine at a very early design stage the processor load and feasible application task sets that can be scheduled together on a single processor. This will aid in future development of flight software and to increase services on board the SUNSAT microsatellite.

4.7 Conclusion

This chapter explored three ways of implementing the RDF architecture, viz. by multi-tasking on a single processor, as each task mapped to its own simple processor, and as high level building blocks on a dual processor platform. All three implementations ensure an architecture realisation that is amenable to rigorous analysis based on a one-to-one correspondence between the designed and implemented systems.

Implementing the RDF architecture with a software kernel on a traditional processor offers a way of implementing a system described in the RDF language on a single traditional processor today. The availability of testing for a feasible schedule enables hard real-time guarantees to be offered by such an implementation. An RDF task can be implemented as a simple processor in silicon with the option of adding additional instructions for application specific task processing. The advances in ASIC design and manufacturing and System on a Chip techniques provide for a one simple processor per task implementation on one device which can guarantee a resource adequate implementation in one package. A high level RDF task can be implemented by a processor pair. One processor executes inter-process communication such as telecommand and telemetry functions, and the other one performs application specific functions such as

audio stream processing. Each implementation maintains the one-to-one correspondence of the implemented components with the components identified in the RDF diagram at the design level.

The computational features required by RDF are supported directly in each one of the implementations, ensuring an efficient implementation and a simple validation process. The direct correspondence of an implemented component with a designed component forms the key to a simple validation step when an implemented system is validated against its designed system.

Chapter 5

Modelling and Evaluation

5.1 Introduction

This dissertation has introduced a computational architecture for the efficient engineering of real-time systems. The notation to describe this architecture was developed in Chapter 3, and three ways of implementing it were explored in Chapter 4. The question remains whether this architecture is rich enough to meet the requirements of real systems, and whether it aids development engineers in better understanding systems. Better understanding is expected to lead to shorter development times, lower costs and more dependable systems.

Other multi-view modelling and implementation environments [28, 106, 105, 75] based on data flow architectures offer language sets to describe intended systems. However, none of them offers a direct mapping to an implementation architecture that matches the architecture of design. The current focus in processor design is not matching design architectures with hardware, but rather optimising power consumption and power-aware operations.

This chapter explores the modelling and analysis possibilities with RDF in a microsatellite case study. The goal is to test the expressiveness of the RDF computational architecture on a sizable problem, and to determine if it is rich enough to express and argue about the temporal properties. In the course of this, an insight into microsatellite software is developed which should lead to shorter development and upgrade cycles for future missions. The microsatellite case study is developed in stages to explore different aspects of RDF. The different satellite models and the reasons for their development are the following.

1. The simple core satellite with telecommand model explores the use of the RDF architecture in an engineering process. The general process is described in Appendix A.6.2.
2. The core satellite model is then extended with telemetry, showing how the state graph description is used to represent information that is difficult to express in RDF.
3. The satellite model with instrument managers is scaled up from the core satellite model with telemetry after recognising an architectural pattern. This explores the ability of patterns to be recognised in an RDF graph, and the extrapolation of the realisability and feasiblity analysis that goes with the scaled model.
4. The satellite model is improved to include a reliable communication service. The resulting model is explored to determine trade-offs in packet size and the number of active channels,

demonstrating that an RDF model does lead to a better understanding of a system.

5. The architectural pattern of a communication service is scaled to a number of communication services in the next model of the satellite. The combination of different characteristics of each of the services can now be explored in the realisability and feasibility analysis of the system. The full satellite model can be assembled from core satellite, telemetry, instrumentation and communication services. With the addition of the interrupt load discussed in Appendix D.1, an estimate of the processing power available for the Attitude Determination and Control System (ADCS) function can be given.

6. The final case study is of a hardware in the loop test bench for the ADCS system. This case study shows a number of design refinement steps with RDF in a multi-processor environment.

The achievements described in this chapter include a scalable model of the software on a microsatellite comparable in complexity to the complexity of the SUNSAT microsatellite. The outcome of this chapter is an insight into the real-time behaviour of the task sets that could execute on a satellite such as the SUNSAT microsatellite. This is useful information for future mission analysis and planning.

The purpose of this chapter is to show how the RDF analysis techniques can usefully be applied for the modelling and analysis of task sets. The other views required to fully specify task sets, i.e. implementation structure, project development status, and OOP/component based architecture, are introduced as part of the case study when they complement the RDF description with information that cannot be represented in RDF.

This chapter starts with a very simple satellite to model and gradually expand it with telemetry, instrumentation servers, connected mode communication, and communication services. Each of the examples is modelled, checked whether it is realisable, and then the feasibility of execution on the processor of the SUNSAT On Board Computer 1 (OBC1) is checked. The final case study is for a hardware in the loop test set-up around an ADCS.

5.2 Core Satellite Model

The first example in this chapter is chosen for its simplicity to focus on the modelling and analysis ability available with RDF. The examples will progressively go into less and less detail of how to work with RDF. The focus will move towards the microsatellite system and a better understanding of its application software.

5.2.1 Problem Statement

The most basic service that executes on the satellite is the telecommand service, which receives messages from the groundstation, decodes them, sets the telecommand registers, and then sends acknowledge messages to the groundstation that the commandd have been executed. The assumptions under which the system operates are:

1. the uplink and downlink communication takes place in unconnected mode, viz. a perfect communication channel is assumed which always delivers messages correctly, and

2. every packet received by the system is a telecommand packet; since this is the only service available, this assumption is valid.

5.2.2 System Architecture

The simplest core of the software system required on the satellite is seen in Figure 5.1. The trigger source for the SCC driver o_{SCCDIN} is the environmental input node o_{Rx}. The SCC drivers o_{SCCDIN} and $o_{SCCDOUT}$ are modelled as two separate tasks. The incoming bytes are assembled into frames and, if no errors occurs, then the AX.25 packet is extracted and passed on to the AX.25 server task $o_{AX.25}$. The AX.25 server task decodes the packet and passes the command contents to the command and control task node $o_{C\&C}$. The command and control task decodes the command, writes the command to the telecommand interface o_{TCMD}, and sends an acknowledgement message to the $o_{AX.25}$ task to be sent back to the groundstation.

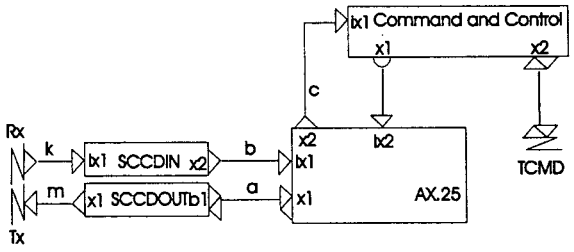


Figure 5.1: Core satellite software

Modelling Features

The following use of the input and output port relationships should be noted. Task node $o_{AX.25}$ executes when a message on either input port ix_1 *or* input port ix_2 is received. Task node $o_{C\&C}$ outputs a message on output port x_1 on completion of the command decode operation. Task node o_{SCCDIN} outputs a message on output port x_2 on receipt of a message on input port ix_1. Task node $o_{SCCDOUT}$ outputs a message on output port x_1 during the processing required when a message is input from bi-directional port b_1. During the execution of task node $o_{C\&C}$, task node o_{TCMD} is written to, and the control returns to $o_{C\&C}$. Otherwise, task node o_{TCMD} does not execute.

5.2.3 Realisability

The system is realisable if all output port rates are determined in terms of the rates from the trigger sources, and if the environmental output rate constraints are observed. See Section 3.6.3 for the conditions of realisability. The only environmental input source driving the system is the environment input node o_{Rx}. The semantics of the environment output node o_{Tx} are such that it does not drive the system, but dictates a maximum rate at which it can output tokens to the environment. That is the maximum rate at which the environment can consume tokens.

The output rate on output port $o_{Rx}.x_1 1$ is r_{Rx}, which is the rate at which the environment drives the environmental input node o_{Rx}. This is also the rate at which the task node o_{SCCDIN} is triggered. The task node o_{SCCDIN} receives bytes which it assembles into packets to be passed on

to the task node $o_{AX.25}$. The worst case behaviour results when the shortest packets are received. A previous analysis of the system [8] revealed 22 bytes as the length of the shortest message. Thus, the rate of message production on output port $o_{SCCDIN}.x_2$ is $ix_1.rate/22 = r_{Rx}/22$. The task node $o_{AX.25}$ receives tokens/messages on input ports ix_1 and ix_2. The task node $o_{AX.25}$ receives messages on input port ix_1, unpacks them and sends the contents to the task node $o_{C\&C}$. The output rate on output port $o_{AX.25}.x2$ is $ix_1.rate = r_{Rx}/22$. The task node $o_{C\&C}$ receives commands on input port ix_1, decodes the commands, sends the contents to the environmental output node o_{TCMD}, and then sends acknowledgements to task node $o_{AX.25}$. The output rate on output ports $o_{C\&C}.x_1$ and $o_{C\&C}.x_2$ is $ix_1.rate = r_{Rx}/22$. The acknowledge messages that task $o_{AX.25}$ receives on input port ix_2 are packetised and sent to the task node $o_{SCCDOUT}$ for transmission to the groundstation. The output rate on output port $o_{AX.25}.x1$ is $ix_2.rate = r_{Rx}/22$. The task node $o_{SCCDOUT}$ receives the packets on input port b_1, and sends them to the environmental output node o_{Tx}. The output rate on output port $o_{SCCDOUT}.x_1$ is $o_{SCCDOUT}.x_1.rate = b_1.rate = r_{Rx}/22$. The environmental output node o_{Tx} then transmits the packets byte for byte triggered by a pre-emptive *EmptyTxBuffer* interrupt from the serial communication controller (SCC). The rate at which o_{Tx} consumes messages is, thus, dictated by the environment. Assume that from a system analysis the acknowledge packet size transmitted by o_{Tx} is also 22 bytes.

The output port rate functions and the input rates are shown in the table in Figure 5.2. Note that task nodes o_{Tx} and o_{Rx} are triggered by a environmentally determined scheduling rate which is shown in the table as r_{Tx} and r_{Rx} on ports $o_{Tx}.e_1$ and $o_{Rx}.e_1$, respectively.

Task node	Port	Input rate	Output rate
o_{Rx}	ie_1	r_{Rx}	
o_{Rx}	x_1		r_{Rx}
o_{SCCDIN}	ix_1	r_{Rx}	
o_{SCCDIN}	x_1		$r_{Rx}/22$
$o_{AX.25}$	ix_1	$r_{Rx}/22$	
$o_{AX.25}$	ix_2	$r_{Rx}/22$	
$o_{AX.25}$	x_1		$r_{Rx}/22$
$o_{AX.25}$	x_2		$r_{Rx}/22$
$o_{C\&C}$	ix_1	$r_{Rx}/22$	
o_{TCMD}	b_1	$r_{Rx}/22$	
$o_{SCCDOUT}$	b_1	$r_{Rx}/22$	
$o_{SCCDOUT}$	x_1		$r_{Rx}/22$
o_{Tx}	ix_1	$r_{Rx}/22$	
o_{Tx}	ie_1	r_{Tx}	
o_{Tx}	x_1		r_{Tx}

Figure 5.2: Input and output rates for the core satellite software

The rate at which task node $o_{AX.25}$ must consume tokens and, hence, be scheduled is:

$$o_{AX.25}.r_{in} = ix_1.rate + ix_2.rate = r_{Rx}/22 + r_{Rx}/22 = r_{Rx}/11$$

and the rate at which task node o_{Tx} must consume tokens and, hence, be scheduled is

$$o_{Tx}.r_{in} = ix_1 + ix_{e1} = r_{Rx}/22 + r_{Rx} = 23/22 r_{Rx}$$

The output rate constraint imposed on the system by environmental output o_{Tx} is that its output may not be faster than r_{Tx}. Thus, the transfer rate function for o_{Tx} is $f_{out}(o_{Tx}) =$

$22/23 r_{in} = 22/23 * 23/22 r_{Rx} = r_{Rx}$. This satisfies the condition, as the environment transmission rate is the same as the environment reception rate, viz. $r_{Tx} = r_{Rx}$.

Given that the output port rate equations for the task nodes are solved, the system is realisable. Given that enough processing power is allocated to each of the task nodes, the implemented system can guarantee its time deadlines. Testing whether enough processing power is available is the subject of the feasibility analysis in the next section.

5.2.4 Feasibility

To calculate the feasibility of a task set, it is necessary to choose a target platform to execute on. This is required as the computational cost incurred when a token/message is received and processed must be known.

For the purpose of the feasibility analysis in this chapter (unless stated otherwise), the task set will execute on an on-board processor such as used on the SUNSAT microsatellite. It is an 80188EC processor running at 13 MHz. This corresponds to a scenario where the processor has been selected not only for its processing capability, but also because of limited power consumption and history for use in space.

Task node	Port	Input rate	Computation time
o_{Rx}	ie_1	r_{Rx}	interrupt input byte
o_{SCCDIN}	ix_1	r_{Rx}	build frame
$o_{AX.25}$	ix_1	$r_{Rx}/22$	get message contents
$o_{AX.25}$	ix_2	$r_{Rx}/22$	build message + $o_{SCCDOUT} \cdot c_i$
$o_{C\&C}$	ix_1	$r_{Rx}/22$	decode command + $o_{TCMD} \cdot c_i$
o_{Tx}	ix_1	$r_{Rx}/22$	start transmission
o_{Tx}	ie_1	r_{Tx}	interrupt output byte

Figure 5.3: Input rate and computational time contributions for core satellite software

The table in Figure 5.3 shows the functions that are performed when a message is received on each of the ports and the resulting computational times. For a system which can receive and transmit at 1200 Baud, the actual periods and estimated execution times are shown in Figure 5.4. The computational costs are based on measurements documented by Boot [8] on an Intel 80C188EC processor running at 13 MHz. Where a measured computation cost value was not available, a value for the computational cost was estimated based on the similarity of computational complexity between a task with a measured value and a task with an unknown value. The processor utilisation is shown for the pre-emptive task set, the non-pre-emptive task set and the total for the processor. The total load on the processor is seen to be just more than 3.8%. There is, thus, 96% of the processor capacity available to accommodate more tasks.

Non-pre-emptive Task Blocking Analysis

For the non-pre-emptive task set a further test is required to see if any of the tasks with a longer period blocks any of the tasks with a shorter period. For this analysis, the computational cost of the non-pre-emptive tasks is increased by to the lower processor availability due to the processor load consumed by the pre-emptive task set. The computational cost c_i' of each non-pre-emptive task then becomes $c_i' = c_i/(1 - \mu_p/\mu)$, where μ_p is the processor utilisation consumed by the

Core satellite software: Kernel, SCC drivers, AX.25, C&C and TCMD output

Adjustable parameters	Rate	units
Baud rate	1200	baud
Bits per byte	8	bits
Bytes per packet (cmd packets)	22	bytes
No of simultaneous channels (SCC channels)	1	channels

Non preemptive task activation

Input no	Port name	Description	Rate /sec	Execution duration(ci) micro sec	Period(pi) micro sec	ci/pi
1	O(SCCDIN).ix1	Input rate for SCCIn Driver	150	35	6667	0,0052
2	O(AX.25).ix1	Input rate from SCCIn	7	927	142857	0,0065
3	O(C&C).ix1	Input rate for C&C from AX.25	7	1400	142857	0,0098
4	O(AX.25).ix2	Input rate from C&C	7	567	142857	0,0040
5	O(Tx).ix1	Input rate from SCCDout	7	722	142857	0,0051
		Non-preemptive task load on processor				0,0306

Preemptive task activation

7	*O(Rx).e1*	*Receiving rate from modem*	*150*	*15*	*6667*	*0,0022*
6	*O(Tx).e1*	*Transmit rate to modem*	*150*	*35*	*6667*	*0,0052*
		Preemptive task load on processor				0,0075
		Total load on processor (> 1 not feasible)				*0,0381*

Figure 5.4: Processor load for core satellite software

pre-emptive task set and μ is the total processor utilisation. For this example, $\mu_p = 0.0075$ and, thus, $c_i' = c_i/(1 - (0.0075/0.0381))$. The modified computational loads for the non-pre-emptive task set are listed in Figure 5.5.

It is assumed that the pre-emptive task set executes with a uniform processor request pattern and contributes equally to the running time of each non-pre-emptive task. This assumption may be violated during the actual execution of the system, if a burst of interrupts occurs which causes the pre-emptive tasks to execute. Given that a hard real-time system is required, a correct solution would be to move these task sets to their own processors on which real-time guarantees can be given.

5.2.5 Scheduling

Pre-emptive Task Set

The interrupt activated task set must be scheduled according to the rate monotonic analysis method. This is because the hardware for current microprocessors makes provisions for priority assignment to interrupts, only.

Adapted non-preemptive task computational load due to preemptive processor load

Port name	*Description*	*Rate* /sec	*Execution* duration(ci) *micro sec*	*Period(pi) micro sec*	*ci/pi*
O(SCCDIN).ix1	*Input rate for SCCIn Driver*	*150*	*44*	*6667*	*0,0065*
O(AX.25).ix1	*Input rate from SCCIn*	*7*	*1154*	*142857*	*0,0081*
O(C&C).ix1	*Input rate for C&C from AX.25*	*7*	*1744*	*142857*	*0,0122*
O(AX.25).ix2	*Input rate from C&C*	*7*	*706*	*142857*	*0,0049*
O(Tx)	*Input rate from SCCDout*	*7*	*899*	*142857*	*0,0063*
	Total load on processor (> 1 not feasible)				0,0381

Figure 5.5: Processor load for non-pre-emptive tasks after taking the processor load of the pre-emptive task set into account for core satellite software

Pre-emptive tasks		
Task node.port	Input rate	Scheduling priority
$o_{Rx}.ie_1$	$r_{Rx} = 150$	1
$o_{Tx}.ie_1$	$r_{Tx} = 150$	1
Non-pre-emptive tasks		
TaskNode.port	Input rate	Scheduling period (micro sec)
$o_{SCCDIN}.ix_1$	r_{Rx}	6667
$o_{AX.25}.ix_1$	$r_{Rx}/22$	146,674
$o_{AX.25}.ix_2$	$r_{Rx}/22$	146,674
$o_{C\&C}.ix_1$	$r_{Rx}/22$	146,674
$o_{Tx}.ix_1$	$r_{Rx}/22$	146,674

Figure 5.6: Priority and period assignment for pre-emptive and non-pre-emptive task sets

Non-pre-emptive Task Set

The non-pre-emptive task set is scheduled with an earliest deadline first scheduler. The deadline for the each of the input ports of a task is the inverse of the input port rate plus the time of message transmission. The messages received on the port must be consumed and processed before the next one arrives. These periods are the inverse of the rates at which the tasks must consume messages/tokens to enable the system to adhere to its real-time requirements. The table in Figure 5.6 shows the priority and period assignment for the task set.

5.2.6 Refinement

The environmental output o_{Tx} must provide the environment with data at a rate of $22 * r_{in}$. This can be modelled in one of two ways as shown in Figure 5.7. Subfigure (i) shows the environmental output o_{Tx}, Subfigure (ii) illustrates a refinement with a periodic timer solution, where the periodic timer triggers the task node at a fixed rate at which it outputs its tokens, and Subfigure (iii) finally shows a refinement with an interrupt source where the SCC device provides a *TxBufferEmpty* trigger to execute the task node. The latter is the one implemented in SUNSAT.

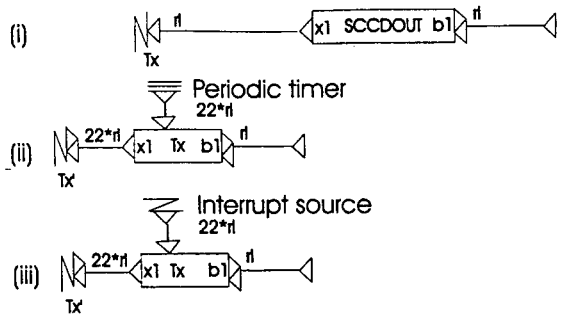


Figure 5.7: Refining an environmental output with synchronous data flow behaviour

5.3 Core Software with Telemetry Manager

5.3.1 Problem Statement

The core satellite cannot give an account of its status or any values measured on the satellite. For this purpose, the status of the satellite must be sampled and transmitted to the ground-station. This is called telemetry (remote measurement). Furthermore, the satellite is in sight of the controlling station for approximately 10 minutes three to four times per day, only. The measurements on the satellite must, thus, be collected, stored on-board, and sent to the control station when requested. For a general indication of activity, the current telemetry frame should be transmitted once every 30 (say) frames sampled. The assumptions under which the system operates are:

1. the telemetry data are sent to the groundstation as a broadcast, and
2. the same downlink is used as for telecommand response.

5.3.2 System Architecture

The telemetry manager is added to the RDF description of the system in Figure 5.1 which results in the RDF graph in Figure 5.8. The telemetry manager consists of four operational task nodes, o_{TLMS} the server, o_{TLMD} the driver, o_{TLMT} a timer, and a data store o_{FILE}. Execution of the telemetry server task node o_{TLMS} is triggered either by an interrupt driven driver o_{TLMD}, the a periodic timer o_{TLMT}, or a command update from $o_{C\&C}$. The telemetry driver is interrupted every time a sample for the telemetry frame is ready. The driver then reads the sample and assembles a telemetry frame of 255 samples which is sent to the telemetry server o_{TLMS}. The telemetry server can either send a telemetry frame to the $o_{AX.25}$ task node for transmission to the groundstation and/or store it in the o_{FILE} data store for later retrieval and transmission to the groundstation. A frame counter and two control station adjustable parameters, viz. *realTimeRepeat* and *storeRepeat*, determine how many frames must be skipped in between transmissions.

The telemetry server can be configured by commands via the command and control task. The specific commands are:

1. setup(int realTimeRepeat, int storeRepeat) — Set up the o_{TLMS} with parameters *real-*

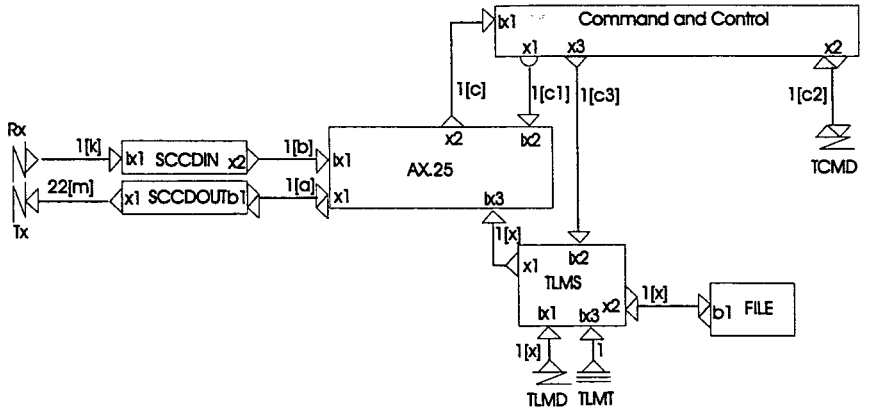


Figure 5.8: Adding telemetry manager to core satellite software

TimeRepeat and *storeRepeat*.

2. start() — Start the o_{TLMS} server.

3. getTelemetryFile() — This command triggers the telemetry server o_{TLMS}, to retrieve the stored telemetry frames one by one and send them to the $o_{AX.25}$ task node for transmission to the groundstation. The first frame is sent in response to the command, subsequent frames are sent on trigger of the single shot timer o_{TLMT} which is set each time by o_{TLMS} if there are more frames to send.

4. clearTelemetryFile() — Clears the current stored telemetry frames in the data store o_{FILE}.

Modelling Features

The behaviour of the telemetry server is best described by a stategraph representation. This will allow unambiguous description of the output port rates, as the internal operation of the task node is explicit. For completeness, the stategraph representation of the behaviour of the telemetry driver is also shown in Figure 5.9.

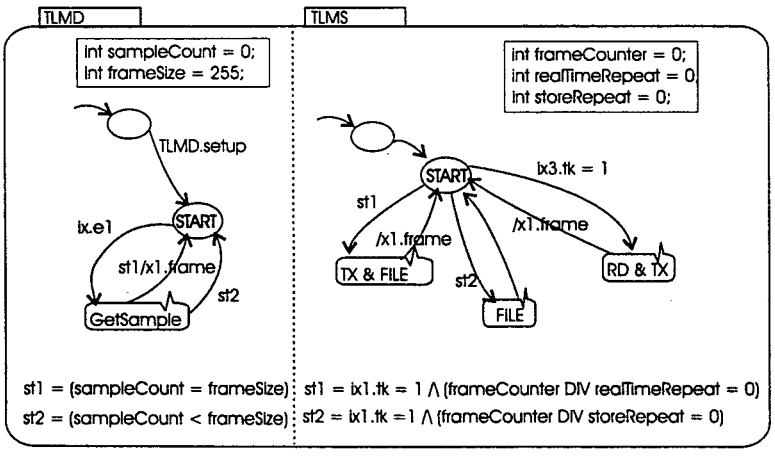


Figure 5.9: Stategraph for telemetry server and telemetry driver

In the stategraph, three local persistent variables are required in the telemetry server task node, viz. *frameCounter*, *realTimeRepeat* and *storeRepeat*. These could be local variables in the corresponding stategraph, or made explicit as data stores that can be accessed external to o_{TLMS}. As these variables are not accessed by any other task, it is recommended that they are local to the TLMS stategraph. In the same way, the stategraph for TLMD requires a *sampleCount* local persistent variable. Each task node has a function that can be called to configure it. The function is called *setup()* and is used in Figure 5.9 to indicate that the telemetry driver environmental input node cannot collect samples until it is configured.

5.3.3 Realisability

The system is realisable, when the output port rate equations are expressed in terms of the environmental input rates, and if the rate constraints on the environmental output ports are met. There are four nodes added to the system with two environmental input nodes that can contribute to triggering the system. The data store o_{FILE} does not contribute to the complexity, as its execution is controlled by the o_{TLMS} node.

The new environmental input sources driving the system are the telemetry driver o_{TLMD} and the telemetry frame transmission timer o_{TLMT}. The input rate for implicit input port $o_{TLMD}.e_1$ is the rate r_{TLMD} at which the telemetry hardware system samples the satellite status either at 150 samples per second (corresponding to a 1200 Baud output) or 1200 samples per second (corresponding to a 9600 Baud output). The telemetry frame length is 255 samples which means that the output rate on output port $o_{TLMD}.x1$ is $r_{TLMD}/255$. This is the rate at which the task node o_{TLMS} is triggered on port ix_1. The task node o_{TLMS} receives telemetry frames which are transmitted and stored or just stored in the data store o_{FILE} depending on the counters *realTimeRepeat* and *storeRepeat*.

The telemetry transmission timer o_{TLMT} is set to trigger the telemetry server if a *getTelemetryFile()* command was received from the command and control task node. The period with which the timer is set is the same as the maximum rate at which telemetry frames can be transmitted, which is in first approximation the same as the rate at which the node o_{TLMD} triggers the telemetry server.

The task node o_{TLMS} receives tokens/messages on input ports ix_1, ix_2 and ix_3. The task node o_{TLMS} receives telemetry frames or notices to transmit stored telemetry frames on input ports ix_1 and ix_3. Task o_{TLMS} then sends the frames to the task node $o_{AX.25}$. The worst case output rate on output port $o_{TLMS}.x1$ is $ix_1.rate + ix_3.rate$. Messages received on input port ix_2 do not result in an output being generated.

The messages that task node o_{TLMS} receives on input port ix_2 require further investigation. In the worst case, three telemetry server command messages can be sent in succession, each being 22 bytes long. This results in messages being received by o_{TLMS} at the rate $r_{Rx}/22$. However, after this burst of messages, it might take another orbit before the same sequence is sent again. The rate at which the telemetry server must be able to consume the messages is $r_{Rx}/22$, but the effective rate over a 100 minute period is $r_{Rx}/22/(100\ minutes)$ which is negligible. It is also expected that no commands for the telemetry server are sent while the file is being output. Thus, for the worst case analysis, this input rate contribution is ignored.

The output port rate functions and the input rates are shown in the table in Figure 5.10. Compared to the table in Figure 5.2, the task node $o_{AX.25}$ has another input port on which it can receive messages. From the table in Figure 5.10 it can be seen for node o_{Tx} that for the

Task node	Port	Input rate	Output rate
o_{Rx}	x_1		r_{Rx}
o_{Rx}	ie_1	r_{Rx}	
o_{TLMD}	ie_1	r_{TLMD}	
o_{TLMD}	x_1		$r_{TLMD}/255$
o_{TLMT}	ie_1	$r_{TLMD}/255$	
o_{TLMT}	x_1		$r_{TLMD}/255$
o_{SCCDIN}	ix_1	r_{Rx}	
o_{SCCDIN}	x_1		$r_{Rx}/22$
$o_{AX.25}$	ix_1	$r_{Rx}/22$	
$o_{AX.25}$	ix_2	$r_{Rx}/22$	
$o_{AX.25}$	ix_3	$r_{TLMD} + r_{TLMT}$	
$o_{AX.25}$	x_1		$r_{Rx}/22 + 2r_{TLMD}$
$o_{AX.25}$	x_2		$r_{Rx}/22$
$o_{C\&C}$	ix_1	$r_{Rx}/22$	
o_{TLMS}	ix_1	r_{TLMD}	
o_{TLMS}	ix_2	$r_{Rx}/22$(ignore)	
o_{TLMS}	ix_3	r_{TLMT}	
o_{TLMS}	x_1		$2r_{TLMD}$
$o_{SCCDOUT}$	b_1	$r_{Rx}/22 + 2r_{TLMD}$	
$o_{SCCDOUT}$	x_1		$r_{Rx}/22 + 2r_{TLMD}$
o_{Tx}	ix_1	$22 * r_{Rx}/22 + 2r_{TLMD}$	
o_{Tx}	ie_1	r_{Tx}	
o_{Tx}	x_1		r_{Tx}

Figure 5.10: Input rate and output rate for core satellite software plus telemetry manager

system to be able not to overrun the following inequality must hold for the rate at which the environment (modem) can accept tokens:

$$r_{Tx} \geq (r_{Rx}/22 * 22) + 2r_{TLMD}$$

With the same modem rate being used for r_{Rx}, r_{TLMD} and r_{Tx}, it is clear that there are three times more data available than the environmental output node o_{Tx} can transmit. To ensure that the system is realisable, the following measures can be taken.

1. The worst case scenario described can be managed by putting a mailbox (FIFO queue) between $o_{SCCDout}$ and o_{Tx} with the maximum number of telemetry frames that can be transmitted from the data store as its size. This would buffer the additional data until the task o_{Tx} consumes them for transmission to the groundstation.

2. System level restrictions on the data sources can be enforced such as:

   (a) Schedule the telemetry collection and telemetry transmission phases. For instance, this can be achieved by not transmitting any telemetry in real time while the stored telemetry is being transmitted.

   (b) Decrease the rate at which telemetry is retrieved from the data store and transmitted by the amount required to accept and acknowledge telecommand messages under normal circumstances.

   (c) Determine a realistic rate at which telecommand packets can be received and executed by the system. For instance, every tenth message could be a telecommand

message. This delay could simply be incorporated in the groundstation command software, to be guaranteed on the system level.

(d) Refine the model to take into account the *realTimeRepeat* parameter which leads to a smaller contribution to the output rate of frames received from the telemetry driver.

Given the above measures, the input rate equations for node o_{TLMS} become:
$o_{TLMS}.r_{in} = ix_1.r_{TLMD} + ix2.r_{Rx}/22/10\ OR$
$o_{TLMS}.r_{in} = ix2.r_{Rx}/22/10 + ix3.r_{TLMT}$
The output rate equation for node o_{TLMS} becomes:
$o_{TLMS}.x_1.rate = ix_1.r_{TLMD}/realTimeRepeat\ \ OR$
$o_{TLMS}.x_1.rate = ix_3.r_{TLMT}$ (1)
where $r_{TLMT} = 9/10 r_{TLMD}$, which allows some bandwidth for telecommand acknowledgement.

The system is then realisable if the following condition holds:

$$r_{Tx} \geq (r_{Rx}/22 * 22)/10 + o_{TLMS}.x_1.rate \quad (2)$$

where the load due to telecommand acknowledgement is 1/10 of what it was due to the system level assumption that at most every tenth message can be a telecommand message.

The rate at which task node $o_{AX.25}$ must consume tokens and, hence, be scheduled is:
$o_{AX.25}.r_{in} = ix_1.rate + ix_2.rate + ix_3.rate$
$= r_{Rx}/22/10 + r_{Rx}/22/10 + o_{TLMS}.x_1.rate$
$= r_{Rx}/110 + ix_1.r_{TLMD}/realTimeRepeat\ \ OR$
$= r_{Rx}/110 + ix_3.r_{TLMT}$
There are two operating combinations for $o_{TLMS}.x_1.rate$ that need to be analysed to determine if the system will operate correctly under all circumstances. In particular, the two system parameters *realTimeRepeat* and r_{TLMT} must be pre-determined at the system level to provide a real-time guarantee. To determine the system level parameters *realTimeRepeat* and r_{TLMT}, they are determined in terms of the rate at which the environment can accept tokens/messages (in this case bytes). From (2) it is clear that if $o_{TLMS}.x_1.rate \leq 9/10 r_{Rx}$, then the system is realisable. Solving for *realTimeRepeat* in (1):
$9/10 r_{Rx} \geq r_{TLMD}/realTimeRepeat$
$with\ r_{Tx} = r_{Rx} = r_{TLMD}$
$realTimeRepeat \geq 10/9$
$realTimeRepeat \geq 2$
Solving for the rate at which timer o_{TLMT} must activate transmission it is found that

$$o_{TLMT}.x_1.rate = r_{TLMT} \leq 9/10\ r_{Tx} * 1/255$$

Given that the output port rate equations are all solved, for both operating conditions the system is realisable. Given that enough processing power is allocated to each of the task nodes, the implemented system can guarantee its deadlines.

It is important to note that the above analysis and system refinement was carried out without any regard to the actual computational cost of each of the task nodes, and without regard to the target platform on which the system will be implemented. The input and output communication rates are independent of processor utilisation, and only once the computational cost c_i and the processor load c_i/p_i is visible it will be clear whether the environmental communication rates or the processing power will dominate the system constraints.

5.3.4 Feasibility

For the task set to be feasible on a particular target, the utilisation for each processor resource must be less than 100% and no task may block another task from execution such that it is delayed past its deadline. The feasibility of the task set is checked for the 80188EC processor target running at 13 MHz. This resource constraint processor is used in the SUNSAT microsatellite. The processor load table is shown in Figure 5.11. From this table a number of aspects should be pointed out:

1. There is more than one valid task set based on the rate constraints of the environmental output task node o_{Tx}. This manifests in the table as *Input no's* 4(a) and 4(b), respectively. The second last and third last processor load columns c_i/p_i reflect the processor load for the two cases.
2. The processor load c_i/p_i for three task sets is shown in the table. The first task set is for the maximum rate at which telecommand packets can be received. The second task set is for normal rate telecommand packets (every tenth message is a telecommand) and a real-time transmission of every *realTimeRepeat* telemetry frame and the collection of every (*storeRepeat* = *1*) frame. The third task set is for the retrieval of the stored telemetry frames from the data store o_{FILE} and the transmission of each of these to the groundstation.

Worst case constraint analysis is not appropriate in this scenario, as the rate at which certain external events happen, such as the rate at which telemetry is collected, is fixed. The focus is to build up a better understanding of typical operational scenarios of task sets executing on the processor.

For a system which can receive and transmit at 1200 Baud, and the telemetry collection runs at $1200/8 samples/second$ for 255 sample frames, the actual periods and execution times are shown in Figure 5.11. The processor utilisation is shown for the pre-emptive task set, the non-pre-emptive task set and the total load of the processor. The total load on the processor for the **TCMD intense** task set is seen to be just more than 7.8%. With this task set the complete downlink bandwidth is consumed with telecommand acknowledgements. If the telecommand load is reduced to one tenth of its full intensity, then the column labelled **TLM RT** (real-time telemetry transmission) and the column labelled **TLM Stor** show a marginally lower processor load. The telemetry and telecommand load on the processor consumes less than 10% of its capability, which leaves more than 90% of its capacity to handle other tasks. It is also clear that the first improvement required for our simple satellite is an increase in the downlink bandwidth from 1200 to 9600 Baud. If this is not done, no more data sources such as scientific instruments can by added to the satellite.

Improving the downlink capability to 9600 Baud with the telemetry and telecommand task set leads to an increase in downlink bandwidth available and to a marginal increase in processor utilisation. See Figure 5.12 for the periods and execution times. It is important to note that the new improved downlink is not restraining the real-time telemetry rate or the telemetry transmission timer rate. The full telemetry rate can, thus, be output to the ground station. The scheduling introduced earlier of either real-time or stored telemetry is retained to enable easier comparison, but also because of the expected requirement for communication bandwidth for other instruments on the satellite.

The total load on the processor from the column labelled **TCMD intense** task set is seen to be just more than 11.6% given that with this task set the telecommand uplink is restricted to

Core satellite software: Kernel, SCC drivers, AX.25, C&C and TCMD output plus Telemetry Server

Adjustable parameters	Rate	units
Input baud rate	1200	baud
Output baud rate	1200	baud
Bits per byte	8	bits
Bytes per packet(cmd packets)	22	bytes
No of simultaneous input channels (SCC channels)	1	channels
Telemetry sample rate	150	samples/sec
Telemetry transmit timer rate	0,529	frames/sec
realTimeRepeat	2	frames

Non preemptive task activation

Input no	Port name	Description	Rate /sec	Execution duration (ci)micro s	Period (pi) micro sec	ci/pi TCMD intense	ci/pi TLM RT	ci/pi TLM store
1	O(SCCDIN).ix1	Input rate for SCCIn Driver	150	35	6667	0,0052	0,0052	0,0052
2	O(AX.25).ix1	Input rate from SCCIn	6,818	927	146667	0,0063	0,0063	0,0063
3	O(AX.25).ix2	Input rate from C&C	6,818	567	146667	0,0039	0,0039	0,0039
4a	O(AX.25).ix3	Input rate from TLMS (realtime)	0,294	1200	3400000	0,0004	0,0004	
4b	O(AX.25).ix3	Input rate from TLMS (storage)	0,529	1200	1888889			0,0006
5	O(C&C).ix1	Input rate for C&C from AX.25	6,818	1400	146667	0,0095	0,0095	0,0095
6	O(TLMS).ix1	Input rate from TLMD	0,588	28000	1700000	0,0165	0,0165	0,0165
7	O(TLMS).ix2	Input rate from C&C	6,818	900	146667	0,0061	0,0061	0,0061
8	O(TLMS).ix3	Input rate from TLMT	0,529	28000	1888889	0,0148	0,0148	0,0148
9a	O(Tx).ix1	Input from SCCDout (only TCMD ack)	6,818	567	146667	0,0039		
9b	O(Tx).ix1	Input from SCCDout (TCMD ack 1/10)	0,682	722	1466667		0,0005	0,0005
9b	O(Tx).ix1	Input rate from SCCDout (TLMS)	0,529	2700	1888889		0,0014	0,0014
		Non–preemptive task load on processor				0,0666	0,0647	0,0650

Preemptive task activation

10	O(Rx).e1	Receiving rate from modem	150	15	6667	0,0022	0,0022	0,0022
11	O(Tx).e1	Transmit rate to modem	150	35	6667	0,0052	0,0052	0,0052
12	O(TLMD).e1	Sample rate for telemetry	150	25	6667	0,0037	0,0037	0,0037
		Preemptive task load on processor				0,0112	0,0112	0,0112
		Total load on processor (> 1 not feasible)				0,0779	0,0759	0,0762

Figure 5.11: Processor load for a telemetry and telecommand satellite with a 1200 Baud bandwidth downlink

the previous 1200 Baud. If the telecommand load is reduced to one tenth of its full intensity, then the column labelled **TLM RT** (real-time telemetry transmission) and the column labelled **TLM Stor** show a marginally lower processor load.

5.4 Instrumentation Managers

System Architecture

The computational architecture of the telemetry manager forms an architectural pattern for handling other instrumentation on the satellite. In particular, the software managers for the GPS receiver, the school experiments, the magnetometer and the star imager can be modelled in exactly the same way. There are two generalisations to the instrumentation manager architecture to be made:

1. None of the other instruments produces a truely continuous stream such as the telemetry of the satellite, so the rate at which data are output to the transmitter is either the rate sampled from the instrument in real time (unlikely), or the rate at which the data are retrieved from the data store for transmission (more likely).
2. The communication from the instruments to the on-board processor can share the same communication bus. Thus, adding another instrument manager does not necessarily increase the rate at which the data must be handled by the processor. The same operations are performed as before, but for another instrument. This further restricts the scheduling combinations of instruments.

The RDF graph for a task set for a satellite with additional instrumentation (GPS receiver, magnetometer, school experiments) is shown in Figure 5.13. Note that the instrument servers all use the same data store for storing and retrieving their data frames. The GPS server has its own communication channel, while the school experiments, magnetometer and star imager share the same communication bus, which limits the maximum rate at which data from these instruments can reach the on-board data handling unit.

Realisability

It is expected that, if the output port rate equations can be solved for the telemetry manager and the instrumentation managers are architectural clones of the telemetry manager, the output port rate equations can be solved for the extended satellite software. Typical values for the data production rates of the instruments are a continuous stream of data at 19200 Baud in 32 byte packets. The input and output rates for the above assumptions are shown in the table in Figure 5.14.

5.4.1 Feasibility

For a task set to be feasible on a particular target, the utilisation for each processor resource must be less than 100%, and no task may block another task from execution such that it is delayed past its deadline.

Core satellite software with telemetry manager

Adjustable parameters	Rate	units
Input baud rate	1200	baud
Output baud rate	9600	baud
Bits per byte	8	bits
Bytes per packet (cmd packets)	22	bytes
No of simultaneous input channels (SCC channels)	1	channels
Telemetry sample rate	150	samples/sec
Telemetry transmit timer rate	0,5882	frames/sec
realTimeRepeat	2	frames

Non preemptive task activation

Input no	Port name	Description	Rate /sec	Execution duration (ci)micro s	Period(pi) *micro sec*	ci/pi TCMD intense	ci/pi TLM RT	ci/pi TLM stor
1	*O(SCCDIN).ix1*	*Input rate for SCCIn Driver*	*150*	*35*	*6667*	*0,0052*	*0,0052*	*0,0052*
2	*O(AX.25).ix1*	*Input rate from SCCIn*	*6,8182*	*927*	*146667*	*0,0063*	*0,0063*	*0,0063*
3	*O(AX.25).ix2*	*Input rate from C&C*	*6,8182*	*567*	*146667*	*0,0039*	*0,0039*	*0,0039*
4a	*O(AX.25).ix3*	*Input rate from TLMS (realtime)*	*0,2941*	*1200*	*3400000*	*0,0004*	*0,0004*	
4b	*O(AX.25).ix3*	*Input rate from TLMS (storage)*	*0,5882*	*1200*	*1700000*			*0,0007*
5	*O(C&C).ix1*	*Input rate for C&C from AX.25*	*6,8182*	*1400*	*146667*	*0,0095*	*0,0095*	*0,0095*
6	*O(TLMS).ix1*	*Input rate from TLMD*	*0,5882*	*28000*	*1700000*	*0,0165*	*0,0165*	*0,0165*
7	*O(TLMS).ix2*	*Input rate from C&C*	*6,8182*	*900*	*146667*	*0,0061*	*0,0061*	*0,0061*
8	*O(TLMS).ix3*	*Input rate from TLMT*	*0,5882*	*28000*	*1700000*	*0,0165*	*0,0165*	*0,0165*
9a	*O(Tx).ix1*	*Input from SCCDout (only TCMD ack)*	*6,8182*	*567*	*146667*	*0,0039*		
9b	*O(Tx).ix1*	*Input from SCCDout (TCMD ack 1/10)*	*0,6818*	*722*	*1466667*		*0,0005*	*0,0005*
9b	*O(Tx).ix1*	*Input rate from SCCDout (TLMS)*	*0,5882*	*2700*	*1700000*		*0,0016*	*0,0016*
		Non-preemptive task load on processor				0,0683	0,0665	0,0668

Preemptive task activation

10	*O(Rx).e1*	*Receiving rate from modem*	*150*	*15*	*6667*	*0,0022*	*0,0022*	*0,0022*
11	*O(Tx).e1*	*Transmit rate to modem*	*1200*	*35*	*833*	*0,0420*	*0,0420*	*0,0420*
12	*O(TLMD).e1*	*Sample rate for telemetry*	*150*	*25*	*6667*	*0,0037*	*0,0037*	*0,0037*
		Preemptive task load on processor				*0,0480*	*0,0480*	*0,0480*
		Total load on processor (> 1 not feasible)				*0,1163*	*0,1145*	*0,1149*

Figure 5.12: Processor load for a telemetry and telecommand satellite with a 9600 Baud downlink

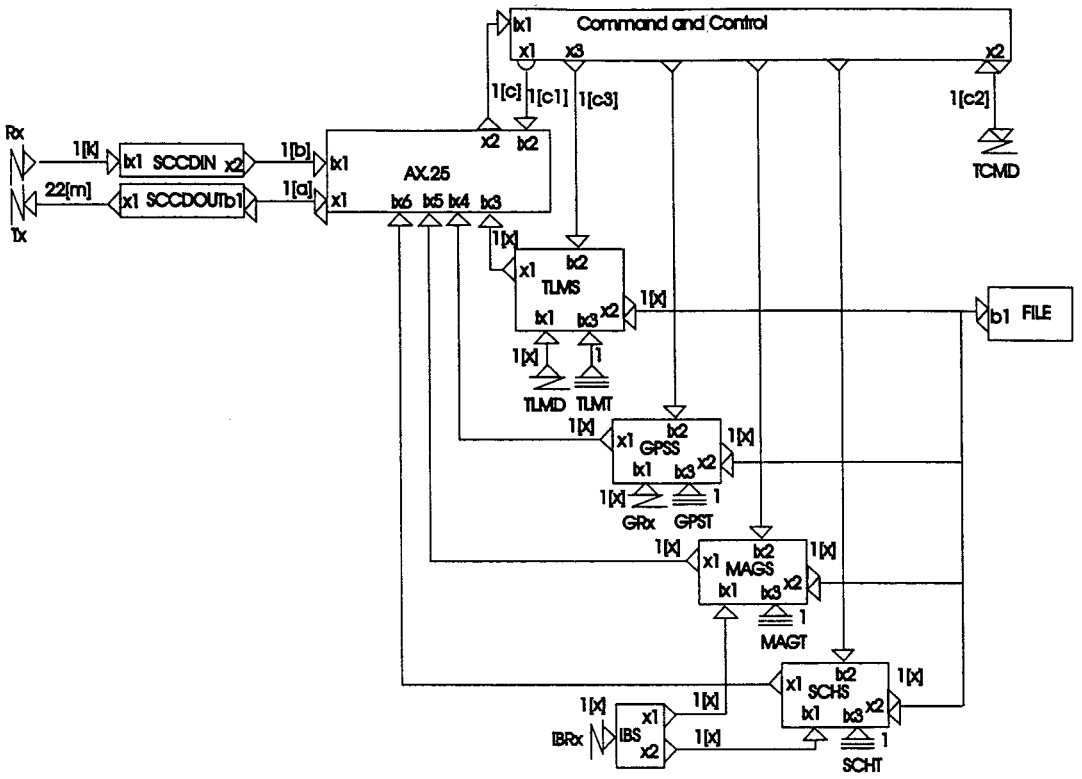


Figure 5.13: Adding instrumentation managers to core satellite software

The processor load table is shown in Figure 5.15 for four task set scenarios. Telemetry and telecommand are present as before. In addition, the school experiments are always being monitored in the task set with label **TCMD+SCHS**. The second scenario in the column labelled **TLM RT + GPSS** has the GPS receiver experiment collecting data while transmitting the stored GPS data frames to the groundstation. The third scenario (in the column labelled TLM Stor + 3 instrum) adds the magnetometer instrument manager to the task set.

The system parameters should be pre-determined to ensure reliable real-time behaviour. In particular, the rate at which the collected instrument data are to be transmitted to the groundstation can be programmed to allow for communication bandwidth for the already established telecommand and telemetry task loads. The rate at which any of the instrument data can be transmitted is $r_{GPST} = (9600 - 1200)/8 * (1/255)$. This allows 25% of the capacity of the bandwidth for telecommand and telemetry transmission. This implies that the information for only one instrument can be retrieved from the satellite at any time. From table in Figure 5.15 the following can be deduced:

1. The reception of the instrumentation frames imposes a significant processor load c_i/p_i, as compared to the other processor load contributions. See the row with label Input number 9 in Table 5.15 where the GPS experiment manager uses 15% of the total processor capacity.

2. From a processor utilisation point of view, all three instrument managers can run at the same time as the core satellite software, while only one instrument's data are transmitted to the groundstation.

Task node	Port	Input rate	Output rate
o_{Rx}	x_1		r_{Rx}
o_{Rx}	ie_1	r_{Rx}	
o_{TLMD}	ie_1	r_{TLMD}	
o_{TLMD}	x_1		$r_{TLMD}/255$
o_{TLMT}	ie_1	$r_{TLMD}/255$	
o_{TLMT}	x_1		$r_{TLMD}/255$
o_{SCCDIN}	ix_1	r_{Rx}	
o_{SCCDIN}	x_1		$r_{Rx}/22$
$o_{AX.25}$	ix_1	$r_{Rx}/22$	
$o_{AX.25}$	ix_2	$r_{Rx}/22$	
$o_{AX.25}$	ix_3	$r_{TLMD} + r_{TLMT}$	
$o_{AX.25}$	x_1		$r_{Rx}/22 + 2r_{TLMD}$
$o_{AX.25}$	x_2		$r_{Rx}/22$
$o_{C\&C}$	ix_1	$r_{Rx}/22$	
o_{TLMS}	ix_1	r_{TLMD}	
o_{TLMS}	ix_2	$r_{Rx}/22$(ignore)	
o_{TLMS}	ix_3	r_{TLMT}	
o_{TLMS}	x_1		$r_{Rx}/22$(ignore)+$2r_{TLMD}$
o_{GPSS}	ix_1	r_{GRx}	
o_{GPSS}	ix_2	$r_{Rx}/22/10$(ignore)	
o_{GPSS}	ix_3	r_{GPST}	
o_{GPSS}	x_1		$r_{Rx}/22/10$(ignore)+r_{GRx} OR $r_{Rx}/22/10$(ignore)+r_{GPST}
o_{MAGS}	ix_1	r_{IBRx}	
o_{MAGS}	ix_2	$r_{Rx}/22/10$(ignore)	
o_{MAGS}	ix_3	r_{MAGT}	
o_{MAGS}	x_1		$r_{Rx}/22/10$(ignore)+r_{IBRx} OR $r_{Rx}/22/10$(ignore)+r_{MAGT}
o_{SCHS}	ix_1	r_{IBRx}	
o_{SCHS}	ix_2	$r_{Rx}/22/10$(ignore)	
o_{SCHS}	ix_3	r_{SCHT}	
o_{SCHS}	x_1		$r_{Rx}/22/10$(ignore)+r_{IBRx} OR $r_{Rx}/22/10$(ignore)+r_{SCHT}
o_{IBRx}	ie_1	r_{IBRx}	
o_{IBRx}	x_1		$r_{IBRx}/32$
o_{IBS}	ix_1	$r_{IBRx}/32$	
o_{IBS}	x_1		$r_{IBRx}/32 * 9/10$
o_{IBS}	x_2		$r_{IBRx}/32 * 1/10$
$o_{SCCDOUT}$	b_1	$r_{Rx}/22 + 2r_{TLMD}$	
$o_{SCCDOUT}$	x_1		$r_{Rx}/22 + 2r_{TLMD}$
o_{Tx}	ix_1	$22 * r_{Rx}/22 + 2r_{TLMD}$	
o_{Tx}	ie_1	r_{Tx}	
o_{Tx}	x_1		r_{Tx}

Figure 5.14: Input and output rates for core satellite plus instrumentation managers

Core satellite software with instrument managers

Adjustable parameters	Rate	units	
Input baud rate	1200	baud	
Output baud rate	9600	baud	
Bits per byte	8	bits	
Instrument baud rate	19200	baud	
Bytes per packet (cmd packets)	22	bytes	
No of simultaneous input channels (SCC channels)	1	channels	
Telemetry sample rate	150	samples/sec	1200/8
Telemetry transmit timer rate	0,5882	frames/sec	1200/8 * 1/255 bytes per frame
realTimeRepeat	2	frames	
GPS transmit timer rate	4,1176	frames/sec	(9600-1200)/8 * 1/255 bytes per frame

Non preemptive task activation

Input no	Port name	Description	Rate /sec	Execution duration (ci)micro s	Period (pi) micro sec	ci/pi TCMD +SCHS	ci/pi TLM RT +GPSS	ci/pi TLM stor +3 instrum	ci/pi TLM stor +3 ins, no gnd station
1	O(SCCDIN).ix1	Input rate for SCCIn Driver	150	35	6667	0,0052	0,0052	0,0052	0,0052
2	O(AX.25).ix1	Input rate from SCCIn	6,8182	927	146667	0,0063	0,0063	0,0063	0,0063
3	O(AX.25).ix2	Input rate from C&C	6,8182	567	146667	0,0039	0,0039	0,0039	0,0039
4a	O(AX.25).ix3	Input rate from TLMS (realtime)	0,2941	1200	3400000	0,0004	0,0004		
4b	O(AX.25).ix3	Input rate from TLMS (storage)	0,5882	1200	1700000			0,0007	0,0007
5	O(C&C).ix1	Input rate for C&C from AX.25	6,8182	1400	146667	0,0095	0,0095	0,0095	0,0095
6	O(TLMS).ix1	Input rate from TLMD	0,5882	28000	1700000	0,0165	0,0165	0,0165	0,0165
7	O(TLMS).ix2	Input rate from C&C	6,8182	900	146667	0,0061	0,0061	0,0061	0,0061
8	O(TLMS).ix3	Input rate from TLMT	0,5882	28000	1700000	0,0165	0,0165	0,0165	0,0165
9	O(GPSS).ix1	Input rate from GRxD - assemble store frame	75	2000	13333		0,1500	0,1500	0,1500
10	O(GPSS).ix2	Input rate from C&C - get command	6,8182	900	146667		0,0061	0,0061	0,0061
11	O(GPSS).ix3	Input rate from GPST - tx frame to ground	4,1176	28000	242857		0,1153	0,1153	
12	O(MAGS).ix1	Input rate from IBRxD	75	2000	13333			0,1500	0,1500
13	O(MAGS).ix2	Input rate from C&C	6,8182	900	146667			0,0061	0,0061
14	O(MAGS).ix3	Input rate from MAGT	4,1176	28000	242857				
15	O(SCHS).ix1	Input rate from IBRxD	75	2000	13333	0,1500	0,1500	0,1500	0,1500
16	O(SCHS).ix2	Input rate from C&C	6,8182	900	146667	0,0061	0,0061	0,0061	0,0061
17	O(SCHS).ix3	Input rate from SCHT	4,1176	28000	242857	0,1153			
18a	O(Tx).ix1	Input from SCCDout (only TCMD ack)	6,8182	567	146667	0,0039			
18b	O(Tx).ix1	Input from SCCDout (TCMD ack 1/10)	0,6818	722	1466667		0,0005	0,0005	0,0005
18b	O(Tx).ix1	Input rate from SCCDout (TLMS)	0,5882	2700	1700000		0,0016	0,0016	0,0016
18b	O(Tx).ix1	Input rate from any Instrument Server	4,1176	2700	242857		0,0111	0,0111	
		Non-preemptive task load on processor				0,3397	0,5052	0,6617	0,5353

Preemptive task activation

10	O(Rx).e1	Receiving rate from modem	150	15	6667	0,0022	0,0022	0,0022	0,0022
11	O(Tx).e1	Transmit rate to modem	1200	35	833	0,0420	0,0420	0,0420	0,0420
12	O(TLMD).e1	Sample rate for telemetry	150	35	6667	0,0052	0,0052	0,0052	0,0052
13	O(GRxD).e1	Sample rate for GPS data	2400	35	417		0,0839	0,0839	0,0839
14	O(IBRxD).e1	Sample rate for instrumentation bus data	2400	35	417	0,0839	0,0839	0,0839	0,0839
		Preemptive task load on processor				*0,1334*	*0,2174*	*0,2174*	*0,2174*
		Total load on processor (> 1 not feasible)				0,4732	0,7226	0,8791	0,7526

Figure 5.15: Processor load for instrumentation managers added to the core satellite software

The processor load scenario in the column labelled **TLM stor + 3 instrum** represents the worst case where the satellite is passing over the groundstation, is collecting data from all the instruments, while at the same time transmitting one of the stored sets of data to the groundstation. This is a representative scenario, but it is also important to consider the load on the satellite when it is not over the groundstation, which is shown in the last c_i/p_i column in Figure 5.15.

5.5 Connected Mode Communication

The communication used in the example satellite so far assumes a communication link which delivers all packets without errors and in correct order. The actual communication link does not exhibit these properties, and a communication protocol is required to detect errors, retransmit missing or damaged packets, and to maintain the correct order of packets. Such a communication protocol is the AX.25 protocol [27].

5.5.1 Problem Statement

The connected mode communication system on SUNSAT consists of various synchronous Serial Communication Controller (SCC) channels, which are used by AX.25 to create a reliable link layer protocol. Various application level protocols including Packet Bulletin Board System (PBBS), Packet Satellite communication (PACSAT), SSTP (Timer server), Satellite File Transfer Protocol (SatFTP), a log server and other house keeping tasks use the AX.25 server to send and receive messages.

AX.25 consists of the link layer (LAPB) and the packet level protocols of the CCITT 1988 recommendation X.25 as amended by the AX.25 protocol definition published in [27]. The unconnected mode communication used in the previous satellite software examples does not make use of any of the connected mode features of AX.25. LAPB can receive a burst of frames up to its maximum window size. The smallest information frames that can occur in a burst would be 3 bytes long. The time to transfer one byte at 9600 Baud is 830 microseconds.

For the purposes of the analysis the complete chain of communication protocols is considered, but it is limited to one application level protocol, viz. Public Bulletin Board System. This is done as it is expected that an architectural pattern will emerge for adding other communication services, which decreases the modelling and analysis effort. If a realisable and feasible solution can be found for one communication service, it is expected that the results can be extrapolated for additional communication services.

5.5.2 System Architecture

A subset of the communication system with one application level communication service is shown in Figure 5.16. This subset includes interrupt handlers for the reception and transmission of bytes, Serial Communication Controller drivers, the AX.25 server $o_{AX.25}$, the Packet Bulletin Board System server o_{PBBS}, the file system (SFS) o_{SFS}, and the command and control process $o_{C\&C}$.

5.5.3 Realisability

A design graph is realisable if the set of output port rate equations can be solved. This is independent of the processor on which the task set is executed. If a design graph is realisable, then the task set is guaranteed to execute within its real-time constraints given that a fast enough processor is provided for each of the task nodes.

The maximum transmission rate of frames from the satellite can be programmed to a specific value. This rate is one of the system parameters to be determined from the real-time modelling and analysis of the SUNSAT software. The minimum transfer time for a frame can be determined by considering the shortest packet. Given an opening flag, the fourteen address bytes, a command byte, then 3 information bytes, two CRC bytes and a closing flag, the frame length is 22 bytes long. The minimum transfer time for a frame is therefore $22 * 830\ microseconds = 18.26\ milliseconds$.

Output Port Rate Functions

The rate at which messages flow in the system is determined by the activity of field stations, groundstation transmissions, and the on-board programmed functions. The worst case communication scenario occurs when a sustained burst of messages is received from the ground (uplink) while the on-board processor is sending a sustained burst of messages to the ground (downlink). Assuming further, that the uplink and downlink communication originators are not related to each other, then two simultaneous full duplex communication channels are active at the same time. This would be the limit imposed by the satellite as no more than two RF communication transmitters can operate simultaneously. The communication model is expanded from its use in unconnected mode as the RF environment has a distinct impact on the efficiency of the protocol. The processing load for individual bytes is handled by the o_{SCCDIN} task node.

The uplink flow of messages begins with the SCC driver being interrupted for every byte received, and by sporadic false alarms due to the white noise input to the SCCs. The SCC driver checks completely assembled packets for integrity, and then sends them to the SCC Server which passes all packets on to the AX.25 Server, which in turn passes the packets on to the correct destination task. The PBBS server o_{PBBS} receives packets which could be directory requests, link management messages or messages to be stored in the file system. For a message that is received to be stored, the worst case real-time behaviour occurs when the message must be stored on the file system o_{SFS}. A downlink activity is triggered with timer L3 that triggers the PBBS server to retrieve a packet from the file system with a synchronous communication call on port $o_{PBBS}.x_2$. This message is then sent to $o_{AX.25}$ with a synchronous call on output port $o_{PBBS}.x_1$. The $o_{AX.25}$ node takes this packet and adds it to its internal queue which is passed on to the SCC driver at a rate determined by o_{L2}.

A design graph is realisable if the set of output port rate function equations can be solved. Starting with the environmental input nodes and the timers in the task graph, the set of equations in terms of port rates is solved. The pre-emptive task o_{Rx} determines the rate at which the on-board processor is interrupted and the rate at which the SCC driver o_{SCCDIN} is signalled for processing. The following assumptions are made on the number of messages for control purposes and channel rate functions for a worst case load:

- $o_{SCCDIN}.ix_1.rate$ = reception Baud rate/(number of bits per character).

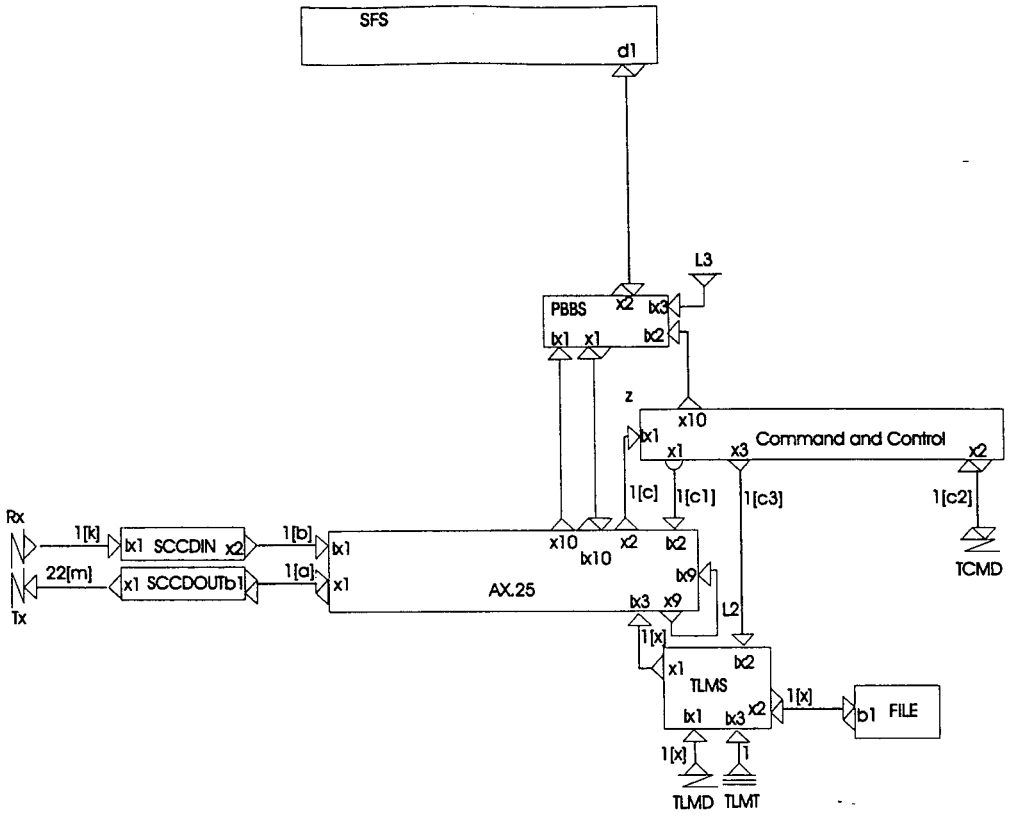


Figure 5.16: RDF diagram of the PBBS communication service

- $o_{AX.25}.ix_1.rate$ = character rate/(packet length) where the minimum packet length is 22 bytes which provides worst case load on processor.
- $o_{C\&C}.ix_1.rate = o_{AX.25}.ix_1.rate/20$ Every 20th packet is for the $o_{C\&C}$ task node.
- $o_{PBBS}.ix_2.rate = o_{C\&C}.ix_1.rate/10$ Every 10th command and control packet is for the o_{PBBS} task node.
- $o_{PBBS}.ix_1.rate = 17/20 * o_{AX.25}.ix_1.rate$ All packets not received by other task nodes are PBBS packets. This would represent a worst case scenario for the processor load. The communication bandwidth might, however, be the dominant impact.
- $o_{L3}.rate$ = rate of responses from the PBBS server. This is a system parameter that can be set to program the processor load.
- $o_{Tx}.ix_1.rate = 1/22 o_{Tx}.ie_1.rate$ The environmental trigger rate determines the allowable rate at which packets can be accepted at node o_{Tx} for transmission. The shortest packet length is 22 bytes what represents the worst case scenario.
- $o_{L2}.rate$ = rate of triggers to the AX.25 server to interrogate the flow control queue and timeouts for possible response generation. This is a system parameter that can be set to program the processor load.

The design graph of Figure 5.16 is realisable, because the set of output port rate equations is solvable in terms of environmental input and time control node rates. The environmental output port rates must be respected by the pre-determined timer periods for o_{L3} and o_{L2}. The resulting rates in terms of input and output ports are shown in the table in Figure 5.17.

Task node	Port	Input rate	Output rate
o_{Rx}	x_1		r_{Rx}
o_{Rx}	ie_1	r_{Rx}	
o_{TLMD}	ie_1	r_{TLMD}	
o_{TLMD}	x_1		$r_{TLMD}/255$
o_{TLMT}	ie_1	$r_{TLMD}/255$	
o_{TLMT}	x_1		$r_{TLMD}/255$
o_{SCCDIN}	ix_1	r_{Rx}	
o_{SCCDIN}	x_2		$r_{Rx}/22$
$o_{AX.25}$	ix_1	$r_{Rx}/22$	
$o_{AX.25}$	ix_2	$r_{Rx}/22$	
$o_{AX.25}$	ix_3	$r_{TLMD} + r_{TLMT}$	
$o_{AX.25}$	ix_9	r_{L2}	
$o_{AX.25}$	x_1		$r_{Rx}/22 + 2r_{TLMD} + r_{L2}$
$o_{AX.25}$	x_2		$r_{Rx}/22/20$
$o_{AX.25}$	x_10		$r_{Rx}/22$
$o_{C\&C}$	ix_1	$r_{Rx}/22$	
$o_{C\&C}$	x_1		$r_{Rx}/22$
$o_{C\&C}$	x_2		$r_{Rx}/22$
$o_{C\&C}$	x_3		$r_{Rx}/22$
$o_{C\&C}$	x_10		$1/200 * r_{Rx}/22$
o_{PBBS}	ix_1	$17/20 * r_{Rx}/22$	
o_{PBBS}	ix_2	$1/200 * r_{Rx}/22$	
o_{PBBS}	ix_3	r_{L3}	
o_{PBBS}	x_1		$17/20 * r_{Rx}/22 + r_{L3}$
o_{TLMS}	ix_1	r_{TLMD}	
o_{TLMS}	ix_2	$r_{Rx}/22$(ignore)	
o_{TLMS}	ix_3	r_{TLMT}	
o_{TLMS}	x_1		$2r_{TLMD}$
o_{TLMS}	x_2		$2r_{TLMD}$
$o_{SCCDOUT}$	b_1	$r_{Rx}/22 + 2r_{TLMD}$	
$o_{SCCDOUT}$	x_1		$r_{Rx}/22 + 2r_{TLMD}$
o_{Tx}	ix_1	$22 * r_{Rx}/22 + 2r_{TLMD}$	
o_{Tx}	ie_1	r_{Tx}	
o_{Tx}	x_1		r_{Tx}

Figure 5.17: Input and output rates for core satellite software plus telemetry manager and communication services

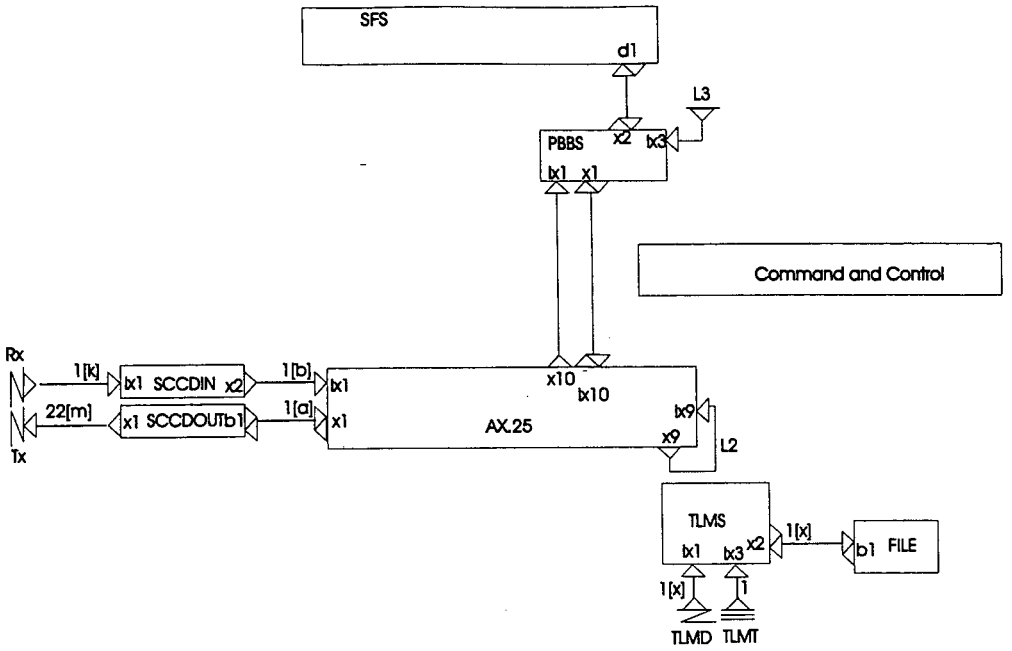


Figure 5.18: RDF diagram of the dominant data paths for the PBBS communication service

5.5.4 Feasibility Analyses

To determine feasibility, the worst case (longest) execution time for each task needs to be known. The table in Figure 5.19 shows the computational cost c_i and the shortest inter-invocation period for each input port. The computational costs used were measured by Boot [8] on an Intel 80C188EC processor running at 13 MHz. Improvements in the execution time are expected as the software is being improved. For the purpose of this analysis, the order of magnitude is more than adequate to demonstrate the usefulness of the RDF analysis.

A particular worst case scenario is assumed to determine the processor load with one communication channel. It is assumed that all packets received from the groundstation are sent to the PBBS service. Thus, all set-up operations of the satellite have been completed, the telemetry manager collects data for later transmission, and no telecommand packets are received. These assumptions identify the dominant contributors to processor load. The resulting RDF graph with the dominant data paths is shown in Figure 5.18.

The feasibility analysis table is shown in Figure 5.19. This is for one active SCC channel and 22 bytes messages, which represent the worst case in terms of processor load. For a single channel the processor utilisation is seen to be 10%. This is with no other interrupt load on the processor than servicing the SCC channel for full duplex communication.

5.5.5 Number of Channels and Packet Size

The sensitivity to changes in processor utilisation with increasing processing loads is important for mission planning and scheduling. In particular, the quality of service of the communication

Core satellite software: Kernel, SCC drivers, AX.25, TLMS and PBBS

Adjustable parameters	Rate	units	Scene1	Scene2	Scene3	Scene4
Input baud rate	1200	baud	1200	1200	1200	1200
Output baud rate	1200	baud	1200	1200	1200	1200
Bits per byte	8	bits	8	8	8	8
Bytes per packet (cmd packets)	22	bytes	22	22	120	220
No of simultaneous TT&C channels (SCC)	1	channels	1	1	1	1
No of simultaneous comms channels (SCC)	1	channels	1	12	12	12
Telemetry sample rate	150	samples/sec	150	150	150	150
Telemetry transmit timer rate	0,588	frames/sec	0,58824	0,58824	0,58824	0,58824
realTimeRepeat	2	frames	2	2	2	2

Non preemptive task activation

Input no	Port name	Description	Rate /sec	Execution duration (ci)micro s	Period(pi) micro sec	ci/pi	ci/pi	ci/pi	ci/pi
1	*O(SCCDIN).ix1*	*Input rate for SCCIn Driver*	*150*	*35*	*6667*	*0,0053*	*0,0630*	*0,0630*	*0,0630*
2	*O(AX.25).ix1*	*Input rate from SCCIn*	*6,818*	*927*	*146667*	*0,0063*	*0,0758*	*0,0139*	*0,0076*
3	*O(AX.25).ix9*	*Input rate from timer L2*	*6,818*	*683*	*146667*	*0,0047*	*0,0559*	*0,0102*	*0,0056*
4	*O(PBBS).ix1*	*Input rate from AX.25*	*6,818*	*4000*	*146667*	*0,0273*	*0,3273*	*0,0600*	*0,0327*
5	*O(PBBS).ix3*	*Input rate from timer L3*	*6,818*	*2000*	*146667*	*0,0136*	*0,1636*	*0,0300*	*0,0164*
6	*O(TLMS).ix1*	*Input rate from TLMD*	*0,588*	*28000*	*1700000*	*0,0165*	*0,0165*	*0,0165*	*0,0165*
7	*O(TLMS).ix3*	*Input rate from TLMT*	*0,588*	*28000*	*1700000*	*0,0165*	*0,0165*	*0,0165*	*0,0165*
8	*O(Tx).ix1*	*Input from SCCDout (only PBBS)*	*6,818*	*567*	*146667*	*0,0039*	*0,0464*	*0,0085*	*0,0046*
		Non-preemptive task load on processor				0,0939	0,7650	0,2186	0,1628

Preemptive task activation

9	O(Rx).e1	Receiving rate from modem	150	15	6667	0,0023	0,0270	0,0270	0,0270
10	O(Tx).e1	Transmit rate to modem	150	35	6667	0,0053	0,0053	0,0053	0,0053
11	O(TLMD).e1	Sample rate for telemetry	150	25	6667	0,0038	0,0038	0,0038	0,0038
		Preemptive task load on processor				*0,0113*	*0,0360*	*0,0360*	*0,0360*
		Total load on processor (> 1 not feasible)				*0,1052*	*0,8010*	*0,2546*	*0,1988*

Figure 5.19: PBBS server processor utilisation for 1 channel, 12 channels, 120 bytes messages and 220 bytes messages

services is programmable and, hence, the load on the processor is programmable. The other processor loads are either on or off depending on a particular mission objective.

To investigate the increase in processor load (i.e. utilisation) due to increased communication services, the number of simultaneous active channels is varied. Another parameter which has a significant influence on the processor load is packet size. This parameter cannot be directly dictated by the SUNSAT operators for public communication services. However, when measured for actual traffic, it will provide information as to the processor utilisation which will be achieved for a particular number of channels. Communication applications under control of the satellite operator could operate more tasks simultaneously on the satellite when using larger packets rather than smaller packets.

The total capacity of the communication link is a function of the access protocol, the number of groundstations which require simultaneous access, and the packet size. This optimisation is not the subject of this dissertation, and a study of the access protocols was given in [32]. This section is aimed at optimising processor utilisation and guaranteeing real-time response. The effect of packet size and number of simultaneous channels can be seen in the last three columns of the table in Figure 5.19. Increasing the number of channels leads to a significant processor load of 80% in the column labelled Scene2, while increasing the size of the packets for the same number of channels leads to a significant decrease in processor load in the columns labelled Scene3 and Scene4 in Figure 5.19.

5.5.6 Increasing the Number of Communication Services

System Architecture

See Figure 5.20 for the RDF representation of the system architecture with additional communication services.

Realisability and Feasibility

The RDF graph shows an architectural pattern for the communication services added. A solution was found for the output port rates for one service in Section 5.5. Given that no other changes are made to the design graph, the output port rates for the RDF graph in Figure 5.20 can be determined and, hence, the RDF graph is realisable.

The feasibility of execution on a single processor can also be extrapolated from the single communication service case. The processing required on the input and output SCC channels is bound by the bandwidth on the physical communication channels. Furthermore, assuming that all digital protocols consume more or less the same amount of processor resources, as they all implement more or less the same algorithms, then increasing the number of channels is the same as increasing the number of communication services. Each service is provided on one or more physical channel. Thus, the expected task set feasibility can be deduced from the analysis of a single communication service.

It is, however, expected that each particular communication service has its own average packet size. The satellite FTP service would normally use as large a packet as possible. While the Automatic Position and Reporting System (APRS) is optimised for a maximum number of accesses from field stations and uses, hence, very small packets. It is expected that the PACSAT

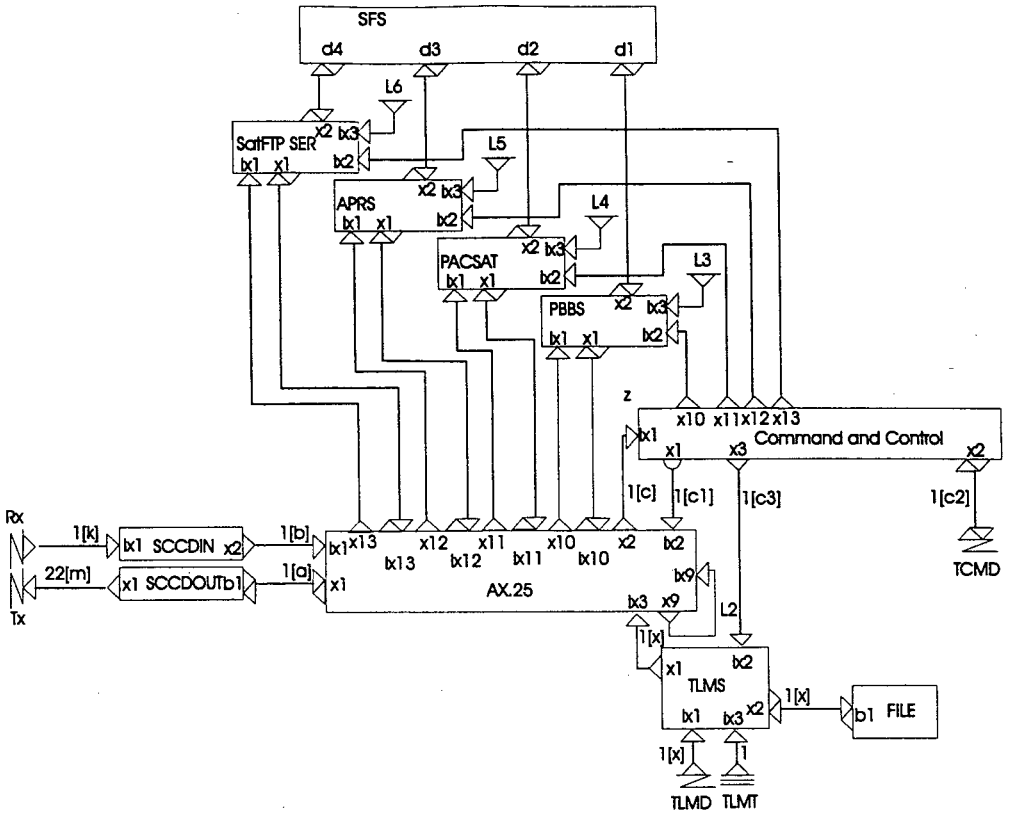


Figure 5.20: RDF diagram of satellite software with four communication services

protocol is mostly used for public file transfer that will lead to large packets, while the Public Bulletin Board System (PBBS) is typically used by people typing on terminal software where each line is sent when concluded with a carriage return. Thus, the packets will be of intermediate size, typically from 40 to 80 characters.

For the four communication services PBBS, PACSAT, SatFTP and APRS to operate together, an optimal resource allocation can be established which would provide the best balance between processor utilisation, real-time guarantees, and access to communication bandwidth. One operating scenario could be one SatFTP channel, two PACSAT channels, and the balance of the processor capacity equally split between the PBBS and APRS services.

5.6 Case Study: HIL Implementation of an ADCS

The following example demonstrates how RDF is used as a modelling language for a multiprocessor system including the ground support segment and the space segment of a test set-up of the Attitude Determination and Control System (ADCS) on the SUNSAT microsatellite. This example was elaborated in parallel also on a conventional computational architecture, with PC software written in Pascal and Transputer software in Modula-2. The goal of this example is to demonstrate design refinement steps within the RDF graph. The example further demonstrates use of a structuring language in conjunction with the RDF graph. No feasibility testing is performed in this example. This section begins with a review of the design steps that are followed, describes the system to be modelled, and continues with a refinement of the

system model.

To determine the time behaviour of a design with an RDF graph, the following design steps are taken:

1. Determine the behaviour of the system at a particular decomposition level.
2. Derive the objects, their operations and interconnection at the level of decomposition that will perform the desired functions.
3. Derive the transmission rate functions for each of the channels.
4. Derive the transmission rates based on the environment and timer rates which drive the system.

The following example demonstrates the application of these design steps.

5.6.1 Problem Statement

The system to be designed consists of a part of the attitude determination and control system of SUNSAT [101] based on magnetometer measurements and torque coil actuators. Attitude is measured with a 3-axis magnetometer and attitude control is actuated with torque coils in three orthogonal dimensions.

A Hardware In the Loop system is to be constructed, which means that some of the sensors, actuators and processors are the real SUNSAT hardware, and some are simulated to complete the system. In this example, the hardware is the Attitude Control Processor. This processor models the magnetosphere, compares the model with the actual measurements in an extended Kalman filter, estimates the co-ordinates and angular velocity of the satellite body, and outputs these values to the torque coil actuator controller on the simulated satellite. Figure 5.21 shows the structure of the system with the various components which will interact.

5.6.2 Functional Description

The space environment, satellite dynamics and kinematics, the magnetometer measurements and torque coils are simulated. The attitude estimator and control algorithm executes on the actual satellite hardware. All current values of the measured, estimated and control parameters are displayed for reference and debugging purposes. The logical task diagram is shown in Figure 5.22.

The function of each of the tasks is given below:

Space environment Calculates the orbit, orbit perturbations and the magnetic field.

Rest of satellite Integrates numerically to solve the differential equations describing the dynamics (angular momentum around the centre of gravity (cg)) and the kinematics (orientation around cg) of the satellite.

Torque coils controller Calculates the forces to exert on the satellite body to keep the nadir face earth pointing and the satellite spinning at 10 minutes per revolution.

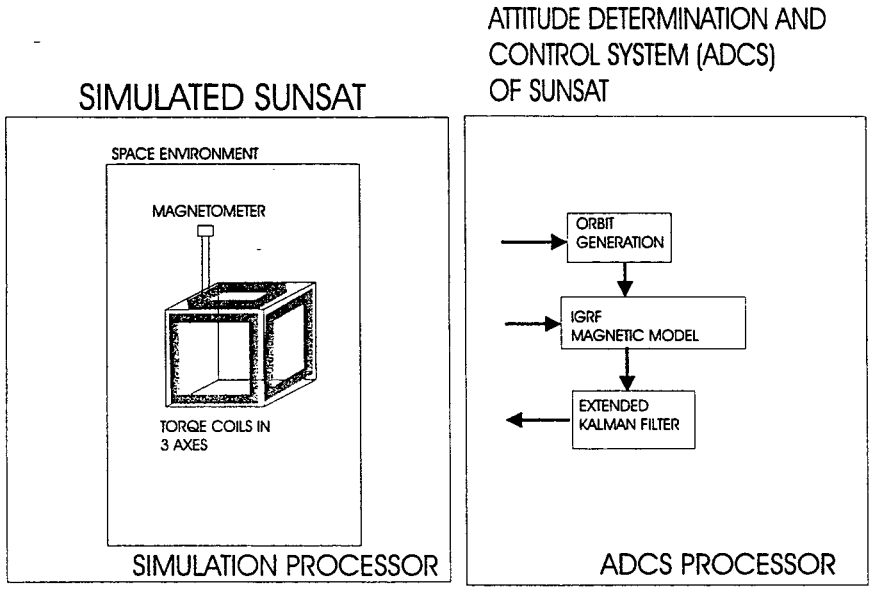


Figure 5.21: Attitude Determination and Control (ADCS) demonstration system

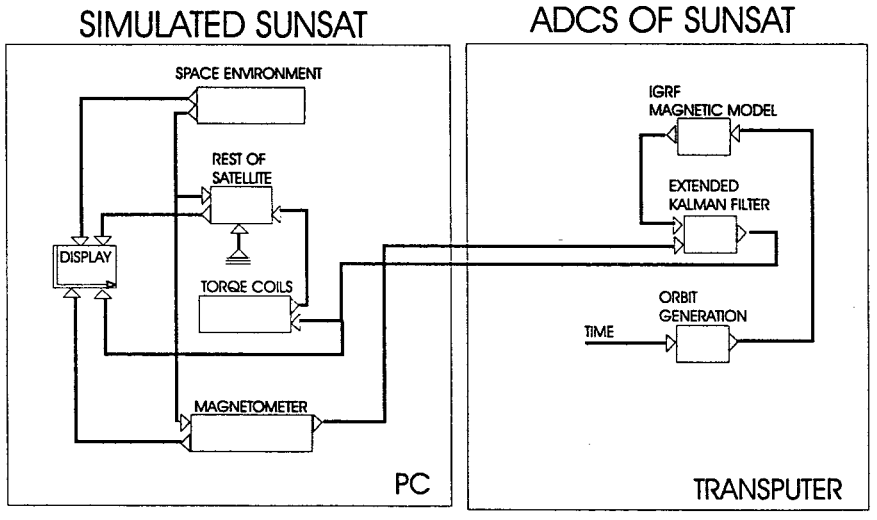


Figure 5.22: Logical task diagram for ADCS demonstration system

Magnetometer Uses the orbit information from the space environment and an IGRF model of the magnetosphere to output an estimated measured value of the three components of the magnetic field strength.

Display Displays all parameters of interest in graphical format for easy interpretation.

IGRF Magnetic model Calculates the expected value of the magnetic field at a specific orbit point.

Orbit generation Given a time after start of launch, it calculates the current point of the satellite in the orbit plane.

Extended Kalman filter Given the measured magnetic field components from the magnetometer and the IGRF magnetic field components from the model, it estimates the quaternion co-ordinates of the satellite body in addition to the estimated angular velocity of the satellite body.

5.6.3 Proposed Design

The designed system is shown in Figure 5.23. It consists of one periodic timer o_{T1} which drives the timing of the system, one data store $o_{SUNSAT\ STATE}$ which holds the orbit components and orientation matrix of the satellite, seven application/server tasks, two communication driver tasks $o_{ASYNC\ MUX1}$ and $o_{ASYNC\ MUX2}$ which interconnect the satellite with the simulation across an asynchronous serial link, and a composite display task. In absence of any other environmental interface which could trigger system functions, the periodic timer is the trigger of the whole system.

The high level functions of each of the tasks and the messages sent and received are as follow:

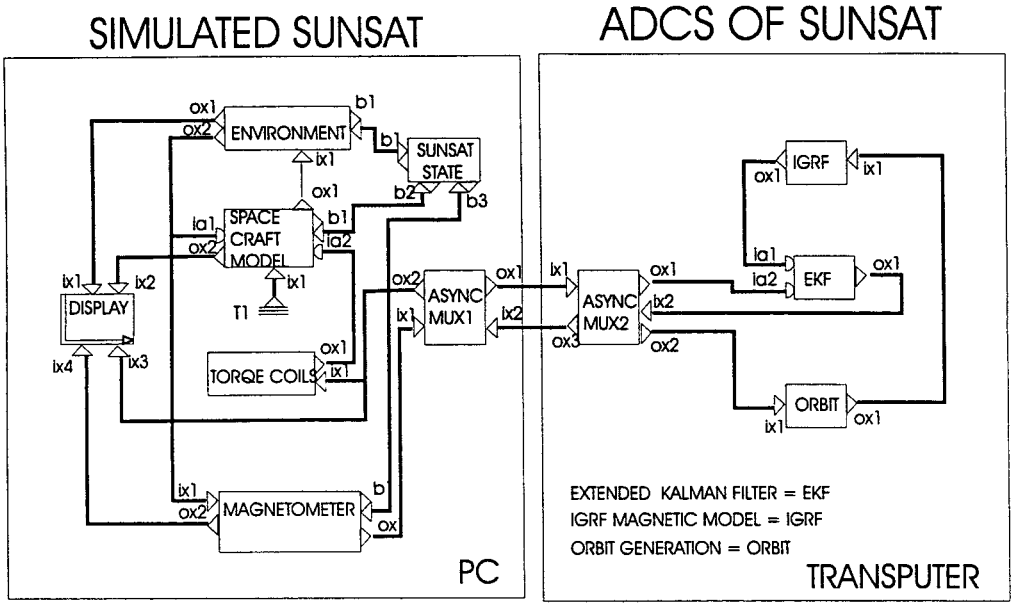
Periodic timer task This timer triggers the integration period of the satellite's kinematics and dynamics. Given that the space craft model is continuous, it must be integrated 10 times faster than the environment.

Space craft model task Integrates numerically to solve the differential equations describing the dynamics and kinematics of the satellite. Triggers the calculation of the space environment every 10 integration steps. Sends the current satellite yaw, pitch and roll angles to the display.

Space environment task Receives an execution trigger from the space craft model to calculate the orbit, orbit perturbations and magnetic field. Passes the orbit and magnetic field information on to the magnetometer simulation, and the orbit and orbit perturbations to the space craft model.

Torque coils controller task Receives the estimated orientation and angular velocities from the extended Kalman filter and calculates the forces to exert on the satellite body, to keep the nadir face earth pointing and the satellite spinning at 10 minutes per revolution around the axis through the nadir face to the earth centre. Sends the resulting forces to the space craft model task.

Magnetometer Uses the orbit and orientation information retrieved from the SUNSAT state data store and an IGRF model of the magnetosphere to calculate and output an estimated measured value of the three magnetic field strength components.




```
-------------------------
Task: IGRF Magnetic Model
Description: Inputs the lattitude and longitude and outputs the magnetic
             field strength components.

Port definitions:
IGRF.ix1.LatiLong : REAL32, REAL32;
IGRF.ox1.BxByBz : [1..3]REAL32;

--------------------------------
Task: EKF Extended Kalman Filter
Description: Estimates the co-ordinates of the centre of gravity (cg)
             of the satellite and the angular momentum around the cg.

Port definitions:
EKF.ia1.BxByBz : [1..3]REAL32;
EKF.ia2.BxmBymBzm : [1..3]REAL32;
EKF.ox1.EQQ1234EwxEwyEwz : [1..4]REAL32, [1..3]REAL32;

----------------------
Task: Orbit generation
Description: Calculates the position in an orbit after a certain amount
             of time has elapsed.

Port definitions:
ORBIT.ix1.Time : CARDINAL64;
ORBIT.ox1.LatiLong : REAL32, REAL32;
```

Figure 5.23: Design of attitude determination and control demonstration partially specifying messages and tasks

Display task Receives all the parameters of interest and displays them in graphical format for easy interpretation.

IGRF Magnetic model task Receives the lattitude and longitude from the orbit generation task, calculates the expected value of the magnetic field at a specific orbit point, and sends it to the extended Kalman filter.

Orbit generation task Receives time after start of launch and calculates the current point of the satellite in the orbit plane and sends the information to the IGRF model.

Extended Kalman filter task Receives the measured magnetic field components from the magnetometer and the IGRF magnetic field components from the model, and estimates the quaternion co-ordinates and the estimated angular velocity of the satellite body. The calculated values are sent to the torque coil task.

Async MUX1 task Receives all information meant for the satellite and transmits it over the asynchronous serial link. Receives all information across the serial link from the satellite and distributes it to the correct destinations according to the address field.

Async MUX2 task Receives all information meant for the satellite and distributes it to the correct destinations according to the address field. Receives all information meant for the simulated SUNSAT and transmits it over the asynchronous serial link.

5.6.4 Derivation of Message Transmission Rates

The transmission rates are labelled to the output ports in Figure 5.24. The derivation of these functions follows from the dynamics of the simulation and control algorithms. For instance, the space craft's dynamics model uses a trapezium integration rule which requires two interactions per integration point. Furthermore, the space craft environment is adequately sampled at a rate 10 times lower than the space craft model is simulated. Thus, the output transmission rate from the space craft model to the environment task is $1/20th$ the integration time step.

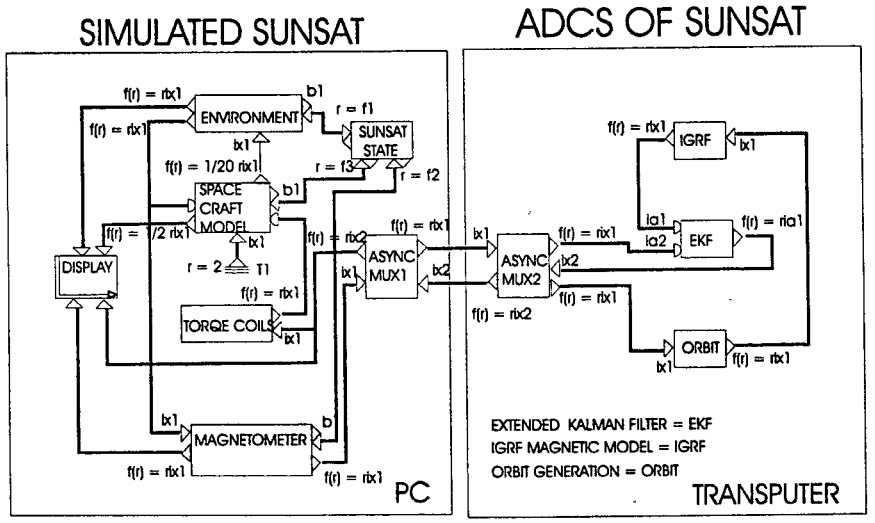


Figure 5.24: Transmission rate functions for the ADCS design

The dynamics of the space craft and environment indicate that a 0.1 Hz sampling update of the environment and sensors is adequate. With the space craft model required to be continuous, it

requires an update rate of 1 Hz. The trapezium rule further dictates two iterations per time step which implies a 2 Hz trigger rate for the space craft model as indicated in Figure 5.24. From this all the transmission rates can be determined.

5.7 Conclusion

Given that there are three implementation media available for realising an RDF architecture and a rigorous notation, the remaining question was how well the RDF architecture lends itself to modelling and analysis of a typical real-time system. This chapter showed that a complex system such as a microsatellite can be modelled in RDF, that the initial analysis is easy enough to be carried out by hand, and that the analysis leads to better understanding of the microsatellite used as a case study. In particular, it was clearly shown that the temporal properties of interest can be described and argued about.

The microsatellite case study was developed in stages to explore different aspects of RDF. The case studies demonstrated the following aspects of RDF:

1. The use of the RDF architecture fits in well with existing engineering processes, both for the design refinement in the ADCS example and the development process described in Appendix A.6.2.
2. The state graph description is complementary to the RDF description representing information difficult to express in RDF.
3. Architectural patterns can be recognised through visual inspection and scaled with visual interpretation. The underlying RDF model scales with the architectural pattern scaling leading to the re-use of analysis results and better understanding of more complex systems.
4. An RDF model leads to better understanding of a system, supporting the exploration of trade-offs in system parameters such as packet size and the number of active channels.

The hypothesis of this dissertation is, thus, supported through the exploration of the case studies in this chapter.

Chapter 6

Conclusions

The problem of modelling, design, analysis and implementation of real-time systems is the subject of this dissertation. Its main focus is on the real-time aspect of systems, and the one-to-one correspondence of a component in the design of a system with a component in its final implemented. Emphasis was placed on a computational architecture which lends itself to modelling and design of telecommunication and control real-time systems at the application level. Equal emphasis was placed on the realisation of this computational architecture in software, hardware and a combination of software and hardware. The problem of realising a correct system as designed is crucial for complex real-time embedded systems, and is encountered in software design with significant impacts on hardware design.

6.1 Motivation

In order to create larger classes of dependable real-time systems, a wider community of people is required to understand and accept the correctness of the functionality implemented in software. Without a computational architecture with simple timing semantics, designers will continue to go through unnecessary iteration cycles of design and testing. With the rising complexity of systems today, there is a demand for an automatable approach that deals with expressing, implementing and validating temporal properties. The first-time correct engineering of real-time systems is essential for a larger class of mission critical systems to reach the market in time and to avoid costly iterations.

6.2 Contributions

Computational Architecture A computational architecture was derived from the real-time data flow paradigm which requires receiving processes to complete processing of previous messages/tokens and to be always ready to accept the next messages/tokens. The single token data flow paradigm was introduced by Jeffay in [52]. It was extended and through a rigorous semantics it was defined as the RDF computational architecture. Some of the extensions are:

1. The single token real-time data flow is extended with synchronous data flow. This allows tasks producing (a priori known) multiple tokens to be interconnected and scheduled.

2. The triggering semantics are extended from OR-triggered to AND- and OR-triggered semantics on the same task. The original real-time data flow paradigm made provisions for OR-triggered semantics, only, while traditional data flow systems just make provisions for AND-triggered semantics.

3. In the process of introducing the triggering semantics, channels and ports were explicitly separated. Tasks no longer know the channels with which they are interconnected, but only the ports on each task. This significantly improves the re-usability of task components as all knowledge on the time properties is now local to the ports and, hence, to the task.

A basis for hardware/software co-design was established. Its key is that within the engineering process from design to implementation a common computational architecture is available which facilitates:

1. a one-to-one translation from design to implementation without going through a translation process from one computational architecture to another, leading to
2. smaller human cognitive effort to understand and comprehend a system at any level of refinement, leading to
3. less effort for validation by diverse back translation, leading to
4. a larger class of systems that can be implemented in software and certified for use in mission and safety critical systems.

It was demonstrated that the RDF computational architecture is expressive enough to model communication, instrumentation, command and control, and control systems. Further, over a period of 8 years, this architecture was used as the basis for the flight software of the mission critical SUNSAT microsatellite. The architecture was used at the high level design and at the task implementation level, only, without taking advantage of its modelling capabilities or the one-to-one transformation capabilities offered in the design refinement process.

Implementation Media A novel real-time processor implementation was proposed, of which preliminary versions were published in [84, 81, 79], consisting of:

1. A simple processor optimised to execute one (or more) task(s) with built-in communication subsystems supporting the RDF computational architecture. Each simple processor has a real-time event processor that manages multiple tasks on the simple processor according to the earliest deadline first scheduling algorithm.
2. A simple input/output processor which handles all communication to and from the outside world to the network of simple processors executing a given task set.

A number of these processor types can be integrated onto one silicon device for a resource adequate implementation of a real-time system to guarantee hard-real time performance under the environmental conditions assumed.

Two other implementation media were described to enable use of the RDF computational architecture on conventional processors:

1. The traditional way of implementing a new computational architecture is through a software layer that provides the necessary abstraction. This mechanism makes it possible for multiple tasks to execute on a single processor. The key feature for a hard real-time application are the timeliness guarantees offered by the scheduling algorithm. The earliest deadline first algorithm was chosen instead of the more common fixed priority algorithms found in modern kernels as the expression of deadlines in EDF is a function of the inverse of the token production rate. This satisfies the goal of achieving a one-to-one correspondence between a designed and an implemented system.

   A processor overload management algorithm was proposed which makes provisions for distinguishing between important applications and other one by deciding which processor loads to drop in the case of processor overload.

2. A unique dual processor implementation was introduced which is optimised for situations where there is a clear distinction between command and telemetry message flow and an application specific data flow such as a high speed data link. Both processors together form a processing node which can satisfy the RTDF paradigm on all its ports and, hence, be used as a building block in an RDF system.

6.3 Future Work

Rate Based Distributed Communication The RDF real-time computational model should find application in the modelling of quality of service (QoS) as required for multimedia data streams on networks such as the Internet. The application of the principles of earliest deadline first scheduling on a limited communication bandwidth medium should lead to guaranteed timely delivery of services as long as the restricted communication resource is not overloaded. The analogy of adding more communication bandwidth for additional services required ties in well with the addition of more simple processors to handle an increase in computational load. Further work can be done to understand the rate based characteristics of communication between distributed resources on different network types such as LAN, processor cluster, Intranet and Internet (with known topology). This will provide an understanding of the characteristics of these communication resources, which is crucial to underpin the temporal properties of distributed applications executing on it.

Tool Support Since the work on RDF started, various software tools have become available to support either structured design, object oriented design [26] or state machine based design [44]. Each of the tools assumes an underlying model (computational architecture). It is expected that RDF will be mapable to subsets of what these tools can offer. This has the advantage not to restrict the use of the RDF computational architecture by the availability of tools. In such a tool, the translation of a design to an implementation architecture will, however, not be based on RDF. It is proposed for future work to evaluate the rigour offered by the existing tools for modelling the RDF architecture, and to extend the most appropriate tool(s) to generate implementations in the RDF computational architecture. An intermediate goal in achieving an implementation medium supporting the RDF architecture is to extend a platform independent, embedded system language such as Java to support the RDF computational architecture in its concurrency multi-thread model first as a library, and then as support in the Java Virtual Machine or Java Real Machine levels. This would enable the direct mapping, at the task level, of a design to a Java language program.

Parameterised Hardware Architectures The engineering process requires that there is some knowledge of the execution duration of tasks on a processor to determine the processor load due to all tasks on the processor for feasibility analysis. Currently this can only be achieved by measurements or simulation of machine code. It would be an improvement to have a parameterised set of standard processor architectures so that the processor load for a particular task is known *before* actually executing it. The prior knowledge will allow an a priori grouping and allocation of tasks to processors. It is also applicable for allocating one task to a simple processor of which the final architecture performance is not known. Hardware synthesis efforts might provide insight into this area of future work.

Component Re-use The next key step in utilising the advantages of a commonly agreed upon computational architecture is establishing the set of components that can be used to create applications. For example, in the VDI/VDE 3696 [111] guideline a building block library consisting of 67 components is defined with which all applications in the chemical process industry can be constructed [33]. A challenge would be to identify a building block library with which typical microsatellite applications can be constructed. This is deemed as a feasible project given that the software is known for such satellites in the form of SUNSAT, and that the software was developed within the computational architecture described in this dissertation.

6.4 A Last Word on Architecture Development

The most generally used computational architecture has been the Von Neumann architecture. Implementations of the Von Neumann architecture determined what is possible and feasible in contemporary microprocessors. The next most generally agreed computational architecture is found in high level language compilers such as Algol, Pascal, Modula-2 and C++. One of the primary goals of the computational architecture supported by such compilers is efficient translation into the Von Neumann architecture. A further level of computational architecture are operating systems, which are limited to task level computation and high level generic services providing access to hardware resources.

There is, however, another departure point for computational architectures such as the block diagram languages provided by Simulink [75] and IEC 61131-3 [45]. It is interesting to note that both offer a data flow model and focus on the application domain rather than the computational model supported in existing processors. Still, there is a translation step required from the computational architecture offered by these products to the computational architecture offered by hardware.

RDF contributes to a better understanding of real-time systems by offering a generalised architecture for data flow systems providing for the expressibility of time from the highest design level to the final implementation level. RDF thus offers a computational architecture suitable for application domain modelling. This dissertation goes further to show how this computational architecture can be implemented in hardware and software or a combination of hardware and software, to correspond one-to-one with the design in the same computational architecture. This opens the opportunity for a wider group of people to cognitively understand the systems designed and implemented with RDF. Ultimately, this should result in a wider class of systems amenable to validation techniques such as diverse back translation to certify software for safety critical systems.

Bibliography

[1] M. Ackerman. A kernel for temporally correct reactive systems. Master's thesis, University of Stellenbosch, Dec 1993.

[2] ARM. *ARM Jazelle Technology.* ARM, http://www.arm.com/sitearchitek/support.ns4/html/jazelle?OpenDocument, 22 edition, Nov 2000.

[3] Arvind and K. Gostelow. The U-Interpreter. *IEEE Computer*, 15(2), Feb 1982.

[4] A. Bakkers, H. Roebbers, J. Sunnter, and K. Wijbrans. Design Analysis of a Priority Driven Scheduler for Transputers. In P. Welch, editor, *Transputing '91*, pages 725–736. IOS, Van Diemenstraat 94, 1013 CN Amsterdam, 1991. Proceedings of the International Symposium on Transputers, 22-26 Apr 1991, Sunnyvale, CA, USA.

[5] P. Bakkes and S. Mostert. Fault Tolerance in SUNSAT Satellite. In *Proceedings of the AIAA Computing in Aerospace 9 Conference*, pages 212–215, San Diego, USA, 1993. AIAA.

[6] G. Bilsen, M. Engels, R. Lauwereins, and J. Peperstraete. Cyclo-Static Dataflow. *IEEE Transactions on Signal Processing*, 44(2):3397–408, Feb 1996.

[7] G. Booch. *Object-Oriented Analysis and Design with Applications.* Benjamin Cummings, 2nd edition, 1993.

[8] J. Boot. Implementation and Integration of Ground and Space Segment Communication Software for the SUNSAT Microsatellite. Master's thesis, University of Stellenbosch, Stellenbosch, South Africa, Mar 1997.

[9] Borland. *Delphi Development Environment.* http://www.borland.com/delphi/, 2000.

[10] A. Burns and A. Wellings. Priority Inheritance and Message Passing Communication: A Formal Treatment. *Real-Time Systems*, 3(1):19–44, Mar 1991.

[11] G. C. Buttazzo. *Hard Real-Time Computing Systems.* Kluwer Academic Publishers, first edition, 1987.

[12] P. Butz and U. Renner. TUBSAT-C: a microsat-bus for Earth observation payloads. In *Small Satellites Systems and Services*, pages Session 6 – 96/6/3. CNES, Jun 1996.

[13] K. Chen. A Study on the Timeliness Property in Real-Time Systems. *Real-Time Systems*, 3(3):247–274, Sep 1991.

[14] H. Chetto, M. Silly, and T. Bouchentouf. Dynamic Schedulig of Real-Time Tasks under Precedence Constraints. *Real-Time Systems*, 2(3):181–194, Sep 1990.

[15] N. Cossement, R. Lauwereins, and F. Catthoor. DF*: An Extension of Synchronous Dataflow with Data Dependency and Non-determinism. In *Proceedings of Forum on Design Languages 2000*, Tubingen, Germany, , Sep 2000.

[16] P. J. Courtois and D. L. Parnas. Documentation for Safety Critical Software. In Basili, editor, *Proceedings of 15th International Conference on Software Engineering*, pages 315–323, IEEE Computer Society Press, P.O. Box 3014, Los Alamitos, CA 90720-1264, May 1993. IEEE, IEEE.

[17] P. J. Denning, D. E. Comer, D. Gries, M. C. Mulder, A. B. Tucker, A. J. Turner, and P. R. Youg. Computing as a Discipline. *IEEE Computer*, 22(2):63–70, 1989.

[18] D. Dikel, D. Kane, S. Ornburn, W. Loftus, and J. Wilson. Applying Software Product-Line Architecture. *IEEE Computer*, 30(8):49–55, 1997.

[19] DIN44300. *DIN 44300: Informationsverarbeitung.* Beuth Verlag, Berlin-Cologne, 1985.

[20] B. P. Douglas. *Real-Time UML: Developing Efficient Objects for Embedded Systems.* ISBN-0-201-32579-9. Addision Wesley, USA, 1st edition, 1998.

[21] B. Eckel. *Thinking in Java.* Prentice Hall Computer Books, Jun 2000.

[22] H.-E. Eriksson and M. Penker. *UML Toolkit.* ISBN: 0-471-19161-2. John Wiley and Sons, first edition, 1998.

[23] G. Estrin, R. S. Fenchel, R. R. Razouk, and M. K. Vernon. SARA (System ARchitects Apprentice): Modeling, Analysis, and Simulation Support for Design of Concurrent Systems. *IEEE Transactions on Software Engineering*, 12(2):293–311, 1986.

[24] A. A. Faustini and E. B. Lewis. Toward a Real-Time Dataflow Language. *IEEE Software*, 3(1):29–35, Jan 1986.

[25] M. Fouquet, M. Sweeting, and M. Allery. Remote sensing using small satellites: in orbit capabilities and development programme. In *Small Satellites Systems and Services*, pages Session 3 – 96/3/2. CNES, Jun 1996.

[26] M. Fowler and K. Scott. *UML Distilled: Applying the standard object modelling language.* ISBN 0-201-32563-2. Addison-Wesley, 10th print edition, 1997.

[27] T. L. Fox. *AX.25 Amateur Packet-Radio Link-Layer Protocol.* American Radio Relay League, Inc., 1984.

[28] A. Girault, B. Lee, and E. Lee. Hierarchical Finite State Machines with Multiple Concurrency Models. *IEEE Transactions On Computer-aided Design Of Integrated Circuits And Systems*, 18(6), Jun 1999.

[29] S. Graves, L. Ciscon, and J. Wise. A Modular Software System for Distributed Telerobotics. In *Proceedings of the 1992 IEEE Robotics and Automation Conference*, 1992.

[30] H. Grobler. RTX Programmers Reference Manaul. Technical Report ESL 99, University of Stellenbosch, 1999.

[31] H. Grobler, S. Mostert, et al. Overview of the Communication Interfaces of the Second On-Board Computer of SUNSAT 1. In B. Peterson, editor, *Teletraffic Systems Engineering Seminar*, pages 449–454, Durban, South Africa, Sep 1996. Telkom.

[32] R. Grobler and S. Mostert. SRMA:Dynamic Protocol Management for LEO Store-and-Forward Systems. *The Transactions of the SAIEE*, 90(1):43–53, 1999.

[33] W. A. Halang and A. H. Frigeri. Methods and Languages for Safety Related Real Time Programming. In *SAFECOMP98*, 1998.

[34] W. A. Halang and A. D. Stoyenko. *Constructing Predictable Real Time Systems.* Kluwer Academic Publishers, Dordrecht, The Netherlands, 1991.

[35] D. Harel. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming 8*, pages 231–274, 1987.

[36] D. Harel, H. Lachover, A. Naamad, and A. Pnueli. STATEMATE: A Working Environment for the Development of Complex Reactive Systems. *IEEE Transactions on Software Engineering*, 16(4):403–413, Apr 1990.

[37] D. Harel and A. Naamad. The STATEMATE Semantics of Statecharts. *ACM Transactions on Software Engineering and Methodology*, 5(4):293–333, 1996.

[38] B. Hayes-Roth. A blackboard architecture for control. *Artificial Intelligence*, 26:251–321, 1985.

[39] B. Hayes-Roth. Architectural Foundations for Real-Time Performance in Intelligent Agents. *Real-Time Systems*, 2(1-2):99–126, May 1990.

[40] R. K. Henn. Feasible Processor Allocation in a Hard-Real-Time Environment. *Real-Time Systems*, 1(1):77–93, Jun 1989.

[41] G. Hilderink, A. Bakkers, and J. Broenink. A Distributed Real-Time Java System Based on CSP. In *The Third IEEE International Symposium on Object-Oriented Real-Time Distributed Computing, ISORC 2000*, pages 400–407, Newport Beach, California, Mar 15-17 2000. IEEE.

[42] Hitachi. *H-8 Users Manual.* Hitachi, first edition, 1988.

[43] C. Hoare. *Communicating Sequential Processes.* Prentice Hall, 1986.

[44] *i Logix. Rhapsody in J. i − Logix, http : //www.ilogix.com/fs$_p$rod.htm*, 22 edition, Nov 2000.

[45] IEC. *Draft IEC 1131-1 Programmable controllers - Part 3: Programming lnguages.* International Electrotechnical Commission, 1992.

[46] INMOS. *OCCAM 2 Reference Manual.* Prentice Hall, 1988.

[47] INMOS. *The Transputer Reference Manual.* Prentice Hall, 1988.

[48] Intel Corporation. *8031 Programmer's Reference Manual*, 1986.

[49] Intel Corporation. *80386 Programmer's Reference Manual*, 1986.

[50] Intel Corporation. *Using the 80186/188/C186/C188*, 1988.

[51] Y. Ishikawa, H. Tokuda, and C. W. Mercer. An Object-Oriented Real-Time Programming Language. *IEEE COMPUTER*, 66(10):66–73, Oct 1992.

[52] K. Jeffay. *The Real-Time Producer/Consumer Paradigm: Towards Verifiable Real-Time Computations.* PhD thesis, University of Washington, Sep 1989.

[53] K. Jeffay, D. F. Stanat, and C. U. Martel. On Non-Preemptive Scheduling of Periodic and Sporadic Tasks. In *Proceedings of the Twelfth IEEE Real-Time Systems Symposium*, pages 129–139, San Antonio, Texas, Dec 1991. IEEE Computer Society Press.

[54] S.-K. Jung. *A Computer Architecture for the Execution of Safety Critical Licensable Software Programmed in a Graphical High Level Language.* PhD thesis, Groningen, University of Groningen, The Netherlands, 1992.

[55] J. Kershaw. The VIPER Microprocessor. Technical Report Technical Report 87014, Royal Signals and Radar Establishment, Malvern, Worcs. London: HMSO, 1987.

[56] P. Koopman and J. DeVale. The Exception Handling Effectiveness of POSIX Operating Systems. *IEEE Transactions on Software Engineering*, 26(9):837–849, Sep 2000.

[57] R. Kurki-Suonio. Real Time: Further Misconceptions (or Half-Truths). *IEEE Computer*, 27(6):71–76, 1994.

[58] D. Lanneer. *Design Models and Data-Path Mapping for Signal Processing Architectures.* PhD thesis, IMEC, Leuven, Belgium, Mar 1993.

[59] P. A. Laplante. *Real-time Systems Design and Analysis.* IEEE Press, 1993.

[60] R. Lauwereins, M. Engels, M. Ade, and J. Peperstraete. Grape-II: A System-Level Prototyping Environment for DSP Applications. *IEEE Computer*, 18:35–43, Feb 1995.

[61] H. W. Lawson. Engineering Predictable Real-Time Systems. In W. A. Halang and A. D. Stoyenko, editors, *Real Time Computing*, pages 31–46. Springer, Berlin, Heidelberg, 1994.

[62] E. A. Lee. Multidimensional Streams Rooted in Dataflow. In *Proceedings of the IFIP Working Conference on Architectures and Compilation Techniques for Fine and Medium Grain Parallelism*, New York, Jan 1993. IFIP, North-Holland.

[63] E. A. Lee. Dataflow Process Networks. Technical Report UCB/ERL M94/53, University of California Berkeley, Department of Electrical Engineering and Computer Science, Jul 1994.

[64] E. A. Lee. Embedded Software - An Agenda for Research. Technical Report ERL Technical Report UCB/ERL No. M99/63, Dept. EECS, University of California, Berkeley, CA 94720, Dec 15 1999.

[65] J. P. Lehoczky and S. Ramos-Thuel. An optimal algorithm for scheduling soft- aperiodic tasks in fixed priority preemptive systems. In *Proc. Real-Time Systems Symposium*, pages 110–123. IEEE, Dec 1992.

[66] J. P. Lehoczky, L. Sha, and J. Strosnider. Enhance aperiodic responsiveness in hard real-time environments. In *Proc. Real-Time Systems Symposium*, pages 261–270. IEEE, Dec 1987.

[67] B. Lent and H. Kurmann. The OR Dataflow Architecture for a Machine Embedded Control System. *Real-Time Systems*, 1(2):107–132, Sep 1989.

[68] K. Leppälä. Interpretative execution of program code increases software robustness in embedded computer systems. *Microprocessing and Microprogramming*, 18(1-5):63–68, Sep 1986.

[69] N. Leveson, M. Heimdahl, J. Reese, and R. Ortega. Experiences using Statecharts for a System Requirements Specification. In *Proceedings: Sixth International Workshop on Software Specification and Design*, pages 31–41. IEEE Computer Society Press, Como, Italy, Oct 25-26 1991.

[70] S. T. Levi and A. K. Agrawala. *Real Time System Design.* McGraw-Hill, 1990.

[71] C. Liu and J. Layland. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *Journal of the ACM*, 20:46–61, 1973.

[72] J. W. S. Liu, W.-K. Shih, K.-J. Lin, R. Bettati, and J.-Y. Chung. Imprecise Computations. *Proceedings of the IEEE*, 82(1):83–94, Jan 1994.

[73] V. Lun. *A Framework for Distributed Real-Time Intelligence.* PhD thesis, University of Witwatersrand, Johannesburg, Dec 1990.

[74] I. M. MacLeod and V. Lun. Progressive Reasoning for Real-Time Intelligent Computing. *IEEE Control Systems*, pages 79–83, Apr 1992.

[75] MathWorks. *Matlab User Manual.* MathWorks, MathWorks, United States, Aug 1999.

[76] H.-P. Meske. Conceptual Design of an Architecture for Hard Real Time Computing. In W. Halang, editor, *Proceedings of 19th IFAC/IFIP Workshop on Real Time Programming.* IFIP, 1994.

[77] G. Milne. The SUNSAT Microsatellite Programme. Concepts, Progress, and Opportunities. In *Africon92.* IEEE, Sep 1992.

[78] S. Mostert. Towards Hard Real-Time System Engineering. In A. Stoyenko, editor, *IEEE Workshop on Real-Time Applications*, pages 207–210, Los Alamitos, California, May 1993. IEEE Computer Society Press.

[79] S. Mostert. An architecture supporting dependable real-time systems. Technical Report SUDEE-1/94, University of Stellenbosch, Stellenbosch, South Africa, Jan 1994.

[80] S. Mostert. Real World Interfacing with Transputers. *Transactions of the SAIEE*, 85(2), Jun 1994.

[81] S. Mostert. SUNSAT 2R: Mission Analysis, Resources and Architecture. In *Small Satellites and Control Systems Symposium 94*, Stellenbosch, South Africa, Oct 1994. IEEE.

[82] S. Mostert. A Visual Method for Real-Time Software Engineering. *The Transactions of the SAIEE*, 87(3):128–139, Sep 1996.

[83] S. Mostert and P. Bakkes. Hardware and Software Diversity for a Fault Tolerant Space Data Management System. Technical Report SUDEE-2/93, University of Stellenbosch, Stellenbosch, South Africa, Dec 1993.

[84] S. Mostert, H. Meske, and W. Halang. An Architecture Supporting Hard Real-Time Computing. *Control Engineering Practice*, 3(6):863–870, 1995.

[85] B. Nuseibeh, J. Kramer, and A. Finkelstein. Expressing the Relationships Between Multiple Views in Requirements Specification. In Basili, editor, *Proceedings of 15th International Conference on Software Engineering*, pages 187–196, Los Alamitos, USA, May 1993. IEEE, IEEE Computer Society Press.

[86] A. L. Opdahl. Structure Analysis, Structured Design, Visual Programming. In *1993 IEEE Symposium on Visual Languages*, pages 292–297. IEEE, Aug 1993.

[87] J. Ostroff and W. Wonham. Modelling, Specifying and Verifying Real-time Embedded Computer Systems. In *Proceedings IEEE Real-Time System Symposium*, pages 124–132. IEEE, 1987.

[88] K. Ramamritham and J. Stankovic. Scheduling strategies adopted in Spring: An overview. Technical Report COINS Technical Report 91-45, University of Massachusetts, Department of Computer and Information Science, 1991.

[89] Rowley. *Rowley Parallel Modula-2.* Rowley Associates, 32 Rowley, Cam, Dursley, Gloucestershire GL11 5NT, third edition, Aug 1993.

[90] D. C. Sastry and M. Demirci. The QNX Operating System. *IEEE Computer*, 28(11):75–77, Nov 1995.

[91] A. Schoonwinkel, G. Milne, S. Mostert, W. Steyn, and K. van der Westhuizen. Pre-flight performance of the communication payloads on SUNSAT, South Africa's first remote sensing and packet communications microsatellite. . In *10th AIAA/USU Conference on Small Satellites*, pages Session V, paper 4: 1–19, Logan, Utah, USA, Aug 1996. UTAH University.

[92] K. Schwan and H. Zhou. Dynamic Scheduling of Hard Real-Time Tasks and Real-Time Threads. *IEEE Transactions on Software Engineering*, 18(8):736–747, Aug 1992.

[93] B. Selic, A. Moore, M. Bjorkander, M. Gerhardt, B. Watson, et al. Response to the OMG RFP for Schedulability, Performance and Time. RFP Response OMG document number ad/2000-08-04, OMG, Aug 2000.

[94] L. Sha, J. P. Lehoczky, and R. Rajkumar. Solutions for some practical problems in prioritized preemptive scheduling. *IEEE*, pages 181–191, 1986.

[95] L. Sha, R. Rajkumar, and S. S. Sathaye. Generalized Rate-Monotonic Scheduling Theory: A Framework for Developing Real-Time Systems. *Proceedings of the IEEE*, 82(1):68–82, Jan 1994.

[96] J. A. Sharp. *Data Flow Computing.* Ellis Horwood Series in Computers and their Applications. Halsted Press, John Wiley and Sons, 22 Worcester Rd, Rexdale, Ontario, Canada, first edition, 1985.

[97] A. C. Shaw. Communicating Real-Time State Machines. *IEEE Transactions on Software Engineering*, 18(9):805–816, Sep 1992.

[98] W.-K. Shih and J. W. S. Liu. Algorithms for Scheduling Imprecise Computations with Timing Constraints to Minimize Maximum Error. *IEEE Transactions on Computers*, 44(3):466–471, Mar 1995.

[99] D. Skillicorn and J. Glasgow. Real-Time Specifications Using Lucid. *IEEE Transactions on Software Engineering*, 15(2):221–229, Feb 1989.

[100] B. Sprunt, L. Sha, and J. Lehoczky. Aperiodic Task Scheduling for Hard-Real-Time Systems. *Real-Time Systems*, 1(1):27–60, Jun 1989.

[101] W. Steyn. *A Multi-Mode Attitude Determination and Control System for Small Satellites.* PhD thesis, University of Stellenbosch, December 1995.

[102] A. D. Stoyenko, T. J. Marlowe, and M. F. Younis. A Language for Complex Real-Time Systems. *The Computer Journal*, 38(4):319–338, 1995.

[103] M. Takesue. Dataflow Computer Extension Towards Real-Time Processing. *Real-Time Systems*, 1(4):333–350, Apr 1990.

[104] L. Thiele and K. Strehl. Funstate - An Internal Design Representation for Codesign. In *Proc ICCAD99*, pages –, San Jose, CA, Nov 7-11 1999.

[105] L. Thiele, J. Teich, M. Naedele, K. Strehl, and D. Ziegenbein. SCF - State Machine Controlled Flow Diagrams. Technical report TIK-Report Nr 33, ETH, ETH Zurich, Computer Engineering and Networks Laboratory, CH-8092 Zurich, Jan 1998.

[106] F. Thoen and F. Catthoor. *Modeling, Verification and Exploration of Task-Level Concurrency in Real-Time Embedded Systems.* Kluwer Academic Publishers, Netherlands, first edition, 2000.

[107] M. Timmerman. RTOS Issue II. *Real Time Magazine*, 93(4):6–104, Dec 1993.

[108] C. Tribelhorn. The Provision of a Flexible Environment for Software Development on Embedded Systems. Master's thesis, University of Stellenbosch, Stellenbosch, South Africa, Dec 1997.

[109] C. Tribelhorn and S. Mostert. Optimising Kernel Overhead for the Development of Communication Systems. In T. Broadhurst, editor, *Teletraffics Conference*, Durban, South Africa, Sep 1996. Telkom.

[110] R. van Ommering, F. van der Linde, J. Kramer, and J. Magee. The Koala Component Model for Consumer Electronics Software. *IEEE Computer*, 33(3):78–85, Mar 2000.

[111] VDI3696. *VDI/VDE 3696: Vendor Independent Configuration of Distributed Process Control Systems.* Beuth Verlag, Berlin-Cologne, 1995.

[112] W. W. Wadge and E. A. Ashcroft. *Lucid, the Dataflow Programming Language*, volume 22 of *APIC Studies in Data Processing.* Academic Press, 24-28 Oval Road, LONDON NW1 7DX, first edition, 1985.

[113] A. K. Ward, A. Barrington-Brown, and S. Gardner. Cluster 5M: An Alternative Approach To The CLUSTER Project. In *Small Satellites Systems and Services*, pages Session 1 – 96/1/2. CNES, Jun 1996.

[114] G. Willumsen, O. I. Lindland, J. A. Gulla, and A. H. Seltweit. An Integrated Environment for Validating Conceptual Models. In H.-Y. Lee, editor, *Proceedings Sixth International Workshop on Computer Aided Software Engineering*, pages 353–363. IEEE, IEEE Computer Society Press, Jul 1993.

[115] M. J. Wirhthlin, K. L. Gilson, and B. L. Hutchings. The Nano Processor: a Low Resource Reconfigurable Processor. In *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, pages 1–9. IEEE, 1994.

[116] N. Wirth. A Plea for Lean Software. *IEEE Computer*, 28(2):64–68, 1995.

Appendix A

Development Environment

A.1 Introduction

The RDF computational architecture for real-time system engineering provides the same computational architecture for design and implementation across application domains and implementation options. It thus narrows the semantic gap between design and implementation, leading to better understanding of application and implementation, shorter development cycles, lower cost and, ultimately, the possibility for larger programs to be trusted in real-time systems.

1. An appropriate architecture in isolation is not going to revolutionise real-time system development. There are many other elements in the development environment that have to be in place.
2. It is important to understand the influence of the RDF architecture on the development environment, and where it fits into the development process.

This appendix examines the development environment around the SUNSAT project to better understand the impact of using the RDF architecture. In other words, it is investigated in how many of the elements of a development environment the use of RDF can make a positive contribution. The construction of a real-time system on heterogeneous hardware platforms forces one to scrutinise the application requirements and to make choices which programming language, design language, operating system, application programmers interface and development process to use. A clear understanding of the system is important for the verification and validation of the complete system. Furthermore, the nature of the project environment causes a 50% manpower turnaround every year [1].

This appendix details the choices made and the experience gained in the development environment around the SUNSAT microsatellite. The emphasis will be on how RDF developed and where it can be made full use of in the future. The appendix contains an update on the work reported in [78] just two years after the start of the project. The project is running for 8 years and the spacecraft is already in space for 20 months. The appendix begins by describing the hardware and software required for the application domain. Each of the areas in which a decision had to be made is discussed in turn with all the options available, the final choice made, and the reasons for doing so. The appendix will close by reporting the experience obtained with the approach taken to construct a heterogeneous real-time system.

[1] Graduate students are taken in every year and graduate after two years

Software component	Hardware support	Reliability expected
Boot loader	Fusible link PROM	No errors
Default application	EPROM	'No errors'
Household and integrity tasks	SRAM with EDAC	One error per 30 days
Operating system kernel	SRAM with EDAC	One error per 30 days
Device drivers	SRAM with EDAC	One error per 30 days
Fine ADCS application	SRAM	One error per 30 minutes
Bulletin board application	SRAM with EDAC	One error per 30 days
Experimental software	SRAM with EDAC	One error per orbit

Figure A.1: The software functions and their reliability required

A.2 Application Requirements

The application software is for the space and ground segments of a microsatellite constellation. The microsatellite is a 45 cm cube box weighing 64 kg. The strict mass and power budgets placed constraints on the flight control hardware that had a profound effect on the computing resources available. The real-time system of interest runs on the space segment and the supporting ground segment.

A.2.1 Hardware Platforms

The application (a microsatellite) requirements state that the system must be dependable in a remote and harsh environment such as space. This requirement in addition to the heterogeneous platforms found in the space and the ground segments of the microsatellite complicates the choice of a development and a target executable environment. Traditional dependable hardware for use in space is Triple Modular Redundant (TMR). Each similar module is of high cost due to the reliability encased in it. A less expensive alternative is to have heterogeneous hardware modules to increase fault tolerance and resistance to failure [83]. In SUNSAT the processors are an Intel 80C188EC and 80386SL, and an INMOS T800. In any space-based system, the ground support adds another dimension to heterogeneity, because being accessible for repair and with unlimited power budget, it does not need to have the same architecture as the space segment. The use of traditional processors for the final implementation is inevitable as processors with a space legacy are required for use in space and inexpensive hardware is required for the groundstation. The diversity of architectures of the various processors employed supports the requirement for the same computational architecture to be provided for all processors.

A.2.2 Software Reliability

The levels of reliability required from software in the application domain ranges from ultra-reliability to acceptance of an error once per 30 days. The table in Figure A.1 summarises the software functions and the estimated reliability required.

A.3 Languages

There have been numerous reviews and comparisons of languages suitable for real-time applications [34, 59]. In all cases it was motivated that mainstream languages such as C, Pascal, Ada, Modula-2 etc. are not suitable for the construction of real-time systems. From the commercially available real-time operating systems [107] one can deduce that C and C++ are being used extensively for real-time applications. According to our experience, C is very good high level assembly language, and extreme discipline is required to create dependable software. In favour of C is that it is by far the most widely implemented language on different hardware platforms.

The language chosen for the space segment software is Modula-2 with assembly language inserts for hardware interfacing and processor utilisation improvements. The following requirements dictated this choice.

1. Most of the software engineers in the SUNSAT project are electronic engineers with limited training in software engineering.
2. The project staff joins the project for at most two years at a time.
3. Most of the flight software has to be very reliable.
4. Most of the flight software is to be maintained across a five year time span.

Modula-2 offers the following advantages.

1. It is more readable for people with less software training. This aids in maintenance as most of our staff has limited training.
2. It is stricter in its syntax than C and C++, making it more difficult to induce unintended errors. The stricter syntax also aids in the verification and validation process, making it easier to create reliable software.
3. Compilers are available for all our microprocessors.
4. The new software engineers joining the project were trained in Modula-2 before.

A.3.1 Design Languages

At the start of the project, the number of design languages was restricted to enable faster inter-engineer communication. The design languages can be split in two groups, i.e. structural and behavioral. The table in Figure A.2 lists the options that were available under each of the groups at the start of the project. The structural design languages are aimed at arriving at re-usable software and specify the way in which software is packaged. The behavioral design languages specify how software is going to behave in terms of state changes, execution flow and timing characteristics.

As the project progressed, it became clear that it was more practical to have one set of design languages, only, as it shortened the learning period for new people joining the project. Multiple language sets also created a communication barrier during a review meeting where the semantics of interfacing were not always clear. It was from this realisation that the RDF notation evolved

Structural design languages	Behavioral design languages
Modular decomposition	Flowcharts or pseudo code
HOOD (High Order Object Oriented Design)	Statecharts
Software topology	RDF data flow diagrams
Hardware topology	

Figure A.2: Structural and behavioral design languages

as the only way in which the software was designed and described, other than the actual Modula-2 source code that would be compiled to execute on the on-board processor.

A.4 Operating System

The requirements placed on the operating system are different for the space and ground segments. The space segment hardware resources are costly and constrained by power and mass budgets. The ground segment hardware resources are adequate for all practical purposes. The requirements for the space-based operating system are support for process dispatching, interprocess communication, synchronisation, loading and unloading of process sets, hardware management, and time functions. The implementation of these requirements on different hardware processors leads to the decision where to draw the dividing line in the software between software diversity and homogeneity. The cost effective use of software diversity is explored in [83]. It amounts to using software diversity on the kernel level, where it must be implemented only once, and no software diversity on the application level. It is expected that the software at the application level will be upgraded and maintained over a period of time.

The space segment processors require an efficient operating system kernel in order to make best use of the resources. There are many such kernels on the market [107, 90], which all support priority based scheduling. Deadline driven scheduling is optimal when compared with rate monotonic scheduling [71], and deadline driven scheduling specifies end to end deadlines more succinctly. The RDF notation and the associated architecture are supported at an implementation level with deadline driven scheduling [40]. The RTX implementation by [1] was used, later upgraded by [109], and recently upgraded and maintained by [30].

At the ground segment, processors are in such abundance that it was decided to support the resource adequate paradigm [61]. Each application on the ground would run on its own processor (80x86 PC). The cost associated with this approach is a communication mechanism that must be maintained between processors. The RDF architecture as embodied in the RTX kernel proved to be well suited for the groundstation software, and the current version of the groundstation software is running on RTX in the RDF architecture. Amongst multiple communication links, it supports multiple consoles as part of the groundstation user interface.

A.5 Application Programmers Interface

In order to provide access to a unified computational architecture across space and ground segments an Application Programmers Interface (API) is required. This API has turned out to be the RDF architecture as embodied in the RTX kernel interface, and is the simplest subset of functions required for parallel execution o processes in a hard real-time environment. The same

	Development phase				
Type of software output	Requirements	Concept design	Detailed design	Coding	Installation
Hardware debugging	*			*	*
Subsystem testbed	*			*	*
Hardware demonstration software	*			*	*
Flight software	*	*	*	*	*
Ground station software	*	*	*	*	*
Porting existing software	*	*	*	*	*

Figure A.3: Application of the waterfall development life cycle to various types of software

API is also supported on the ground segment, which enables application software engineers to concentrate on their tasks and not on idiosyncrasies of the different kernels on the different processors. Prototype implementations of the kernel for the 386EX and T800 processors have fallen into disuse. The Rowley Modula-2 compiler [89] for the T800 introduced unexplained errors which have slowed down development. The 386EX on-board computer was declared as back-up, and all programming resources were mobilised to support a successful mission with the 80C188EC on-board computer.

A.6 Software Development Processes

The range of reliability requirements on the software necessitates a flexible approach to software development. The initial approach to software development was based on the waterfall development model with the extension of rapid prototyping before completing the software requirement phase. Flexibility is built into the waterfall model by requiring the outputs of the different stages to be checked for specific types of software, only. See Table A.3 for the types of software and the associated checks which were recommended to be performed at the output of each stage.

A good design process will ensure reliable products [16]. However, the visibility of the correct progress requires intermediate document outputs. Initially, an internal framework document was put in place for each of the following outputs:

1. Software requirements document,
2. Software design document,
3. Software testing document,
4. Software coding standards.

It turned out that the inexperience of the engineers with software creation was the overwhelming influence to the way the software was created. Almost always the software was written as a prototype after some understanding of the requirements was established. The software evolved through one or more iterations before it was left as completed. With each new team member joining, the existing source code was reviewed, declared as not suitable, disposed of and the whole process was repeated.

The waterfall model is top-down and assumes complete prior knowledge of the task at hand. This is certainly not always the case in an engineering internship environment such as found

around the SUNSAT project where there is constant intake of newly graduated engineers. With the experience of software development on SUNSAT, the lightweight development process superceded the waterfall development process.

A.6.1 Lightweight Development Process

In order to improve the quality of the produced product a release document template was composed of the essential elements of requirements, design and testing documents. This document was required to be delivered with the software offered as completed, and it represented a minimum effort of the software engineer in terms of documentation. The minimum checks and outputs of the lightweight software development process are:

1. a specification, high level design and repository organisation document which provides the baseline for the development,
2. the source code of which the executable is demonstrated to perform the required functions as per specification, and
3. a release document indicating how to use the software, the acceptance testing performed, and the actual repository information.

Release Document Template

It is expected that each of the software components is described in a release document of which the template is given below.

```
Release document template - 9/1/95 SM

1. Scope
       Identity
       System item overview
       Reference documents

2. Function requirements
       Function
       Computer hardware architecture
       External interfaces - environment
       System modes
       For each logical object
           Processing
           Input
           Output

3. Engineering issues
       Constraints
       Portability requirements between SUNSAT processors
       Memory requirements
       Timing requirements
       Design options, choice and motivation

4. Logical design - composite system task blocks
      Logical system structure/composition
```

```
        Logical class/functional diagram
        System states and modes
        Memory space
        CPU loading and timing
        For each logical entity
             Processing
             Input protocol (channels/procedures)
             Output protocol (channels/procedures)

    5. Physical design - simple system task blocks
        Physical system structure
        Physical class/functional diagram
        System states and modes
        Memory space
        CPU loading and timing
        Module/task name
             Functional description
             Interface

    6. User manual = how to use this software
        Initialisation required
        How to use the functions
        Closing steps required
        Examples (can be Modula-2/other language code)

    7. Test procedures
        Formal qualification test preparations
             Hardware preparation
             Software preparation
             Other pretest operations
        Formal qualification test descriptions
             Test name
                  Purpose
                  Traceability
                  Initialisation
                  Test inputs
                  Expected results and margins
                  Numbered steps to do test
                  Assumptions and constraints
         Test result sheet (ticked off as the tests are done)

    8. Appendixes
           .def files
           .mod files
           .cmd location files
           .red files
```

The lightweight development process has the disadvantage that evidence of a correct output is only available on completion of the engineering process. There is no visibility of progress or a correct and successful outcome until completion of the project. This is not acceptable due to the risks involved in wasted resources, late delivery of software, and non-operating or non-complete systems. The lightweight development process further requires a software team with exceptional capability.

A.6.2 RDF Engineering Process

The RDF architecture provides a notation, architecture, and analysis capabilities to clearly describe and explore the behaviour of a system from the modelling phase through design refinement and implementation refinement to the final product. This section explores the RDF software engineering process embedded in the partial refinement software development process. The RDF engineering process is embedded in a software development process that recognises the flexibility required to cope with varying amounts of knowledge about a system and provides for bottom-up implementation of those areas of risk. The lightweight development process is superceded by the partial refinement development process which was introduced in Chapter 1. It is repeated here for the discussion. The partial refinement development process consists of the following steps. It is derived from the object oriented analysis and design process described by Booch in [7].

1. Establish core requirements (conceptualisation)
2. Develop a model of the desired behaviour (functional specification and analysis)
3. Create an architecture for the evolving implementation (design)
4. Evolve the implementation through partial refinement
   (a) Identify classes and objects
   (b) Identify semantics of classes and objects
   (c) Identify relationships between classes and objects
   (d) Specify interfaces and implementation of classes and objects
   (e) Refine a number of classes and objects suitable for implementation
   (f) Go back to identifying classes and objects
5. Manage post-delivery evolution (maintenance)

A.6.3 Software Engineering Process

The software development process must be applicable to software components ranging from reused existing ones to completely newly developed one. The RDF engineering process is shown diagrammatically in Figure A.4. The modeling activity consists of building the architectural model of the system to be implemented. The verification activity is to determine whether the design graph is realisable. The model can be refined to the level of interest either at a high level or a low level description of the system. On completion of the refinement process, and graph realisability verification, the next step is to implement the components on the chosen implementation medium. Implementation 1 is to enable measurements to be taken for feasibility analysis. It does not require the complete integrated task set to be running on a processor. Implementation 2 is the final implementation of the integrated task set. The validation step is to verify that all tasks meet their deadlines within the complete task set environment. If validated, then the task set is released for runtime execution.

The engineering process steps in the partial refinement development process where the RDF architecture has a direct influence in are items 2) to 4). During step 2) 'the model of desired behaviour' in terms of timing properties can be developed and analysed with RDF, during item 3) 'the creation of an architecture for the evolving implementation' the modeling and verification

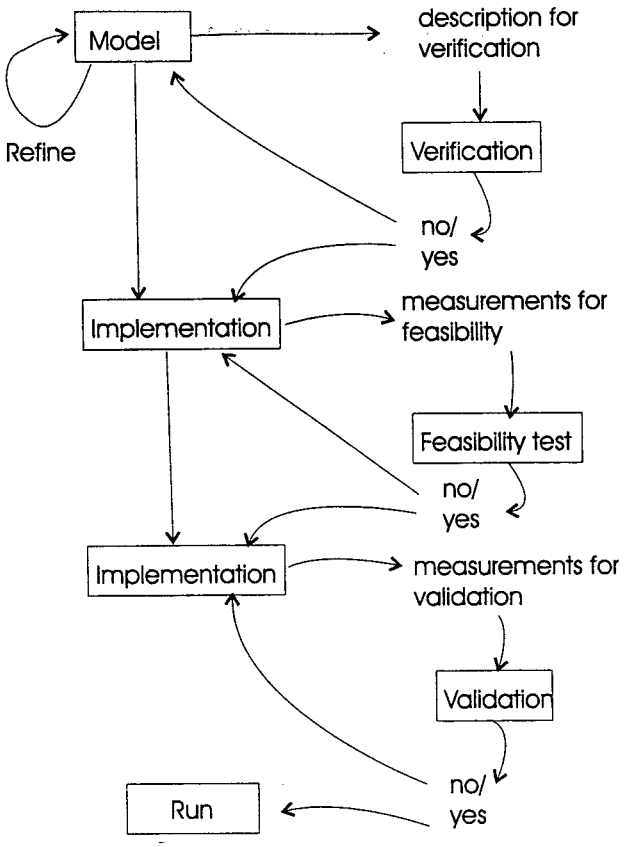


Figure A.4: The engineering process

steps in Figure A.4 can be applied, and during item 4) 'evolving the implementation through partial refinement' the implementation 1 and implementation 2 steps in Figure A.4 can be applied.

This discussion related to how the RDF engineering process interacts with the partial design refinement software development process. This section does not support the partial design refinement software development process as the best in all circumstances, but uses it as an example of how the RDF engineering process could be integrated into a typical software development process based on the experience gained in the SUNSAT project to date.

A.7 Software Team

The most significant progress in the development project was achieved when the following set of conditions on the software team was valid:

1. Already capable engineers and students who had made software a hobby and not only relied on the material taught in engineering school carried out the software engineering.
2. The same people are staying with the project until completion.
3. The software is being maintained by the same people who wrote it. There is, therefore, no learning curve and output is immediately productive.

4. The engineers have a clear understanding of the hardware architecture that the software runs on. This important as the hardware capabilities change over time due to radiation damage and launch phase damage.

5. The software is crafted software, implying that the final product is expected to be a piece of artwork that cannot be changed without changing the meaning of it. This implies that the software is of very high quality.

This kind of software has a certain quality that can only be afforded by the volunteer efforts of people committed to the goal of creating the best software possible. This is the same quality found in the open source movement where volunteers are contributing to each other's efforts.

A.8 Experience with Working in the Development Environment

The results of introducing a structured software development approach as suggested in this section are reported in [78]. The discussion that follows relates to the situation after eight years of progression. The following are the major conclusions, recommendations and results.

1. The choice of Modula-2 as programming language has in fact enabled the new intake of engineers to become productive after a shorter period of time. However, the lack of continuing support by compiler vendors has led to some time wasted due to bugs in code generated by the compilers. One would expect that compilers such as for C would not exhibit the same kind of errors.

2. The boot loader code has gone through the requirements, conceptual and detailed design phases, with formal checks on the output of each phase. It was implemented by a capable programmer and extensively tested before flight. It has indeed been working well over the operational time of the satellite thus far. It is difficult to evaluate how much the initial time spent on design improved its reliability, or by how much it shortened the time taken for its final implementation.

3. It was found that when the proper execution of the requirements and conceptual design phases was enforced, they took about 30% of the time, but that the coding thereafter is significantly faster.

4. For non-ultra reliable software we found that it is adequate to proceed with the design until the module interface level. The additional time spent on detailed design did not provide any additional benefits in saving time for the reliability required.

5. The RDF architecture is proving to be of great benefit as more than one person is working on the software for the same processor after the initial task identification and assignment step.

6. The strategy of resource adequacy for the ground station proved to offer many developers the opportunity to work independently of programming language and operating system. This facilitates true concurrent engineering. However, the processor labelled AX.25 TO SYNC in Figure B.2 of Appendix B gradually assumed the tasks of telemetry processor, telecommand processor, and SatFTP file management processor in addition to the switching of the hardware configuration in the groundstation. This was not a requirement, but

grew out of a single operator principle as the groundstation was more and more used in the testing of the satellite.

7. The choice of using a simple, non-commercial kernel of which total control over the source code is possible, has not proved wrong, yet. The simplicity ensures that one person can understand the complete kernel, facilitating complete transparency through the kernel when implementing on a specific hardware platform.

8. Although the RDF architecture was used at a high level to design the software of the satellite, it was not employed further, in particular not for modelling and analysing timing properties. This can be ascribed to lack of tool support and integration with the development environment. It is expected that one of the existing software CAD tool sets would be suitable to embed the RDF system in.

9. The RDF architecture is currently running on a PC platform and on the 80C188EC onboard processor on SUNSAT as the RTX kernel. Although there is effort saved in re-using the same application, it is found that the effort of writing and maintaining a kernel on different processors should not be underestimated. In particular, the driver software that manages different sets of hardware consumes a significant amount of resources to cater for complexity and errors generally found in the silicon. There is certainly an opportunity here to embed a simple input-output processor with next generation hard input-output devices to offer a standard and simple communication interface between a processor and an input-output device.

A.9 Conclusion on the Development Environment

The impact of the choices made for the development environment for SUNSAT has a lifetime of at least 12 years (7 years development and 5 years expected operation). Current efforts are under way to investigate the suitability of the Java language for a next generation flight software. The computational architecture has proven to be suitable to date and initial indications are that it can embed in a Java library. The RTX kernel has been ported to C, is used on a Fujitsu microprocessor for a mobile tracking application, and is running on an 8031 microcontroller for a prototype demonstrator. Further use of new microprocessors will certainly benefit from the fact that there are C compilers available for most microprocessors.

Appendix B

SUNSAT Software Requirements Specification

This appendix contains a high level requirement specification for the software on the on-board computer of SUNSAT. It was recorded in 1995/1996 by the author of this dissertation.

B.1 Overall SUNSAT Software Specifications

B.1.1 Function

The satellite must be controlled from the ground with telecommands and keep the ground station informed of its status with telemetry. The attitude of the satellite must be controlled within 5 degrees earth pointing under non-imaging circumstances and within 0.057 degrees for imaging. The satellite must function in the store and forward mode compatible with the PACSAT protocol suite and the terrestrial bulletin board protocol, and function as a parrot upon request. Scientific data must collected and disseminated from the GPS receiver, magnetometer, and the school experiments. The satellite must be able to take an image of any place in the world, store it on board for later playback, or transmit it in real time. A DSP processor must be available to execute custom modulation techniques and protocols.

B.1.2 Computer Hardware Architecture

The satellite consists of two general purpose On-Board Computers (OBCs) and a dedicated Attitude Determination and Control System (ADCS) processor. In addition, there are numerous support processors with dedicated functions. The following communication channels are found on the satellite:

1. SUNSAT serial bus, a command and control serial bus interconnecting OBC1, OBC2, telemetry, telecommand and the power system.
2. Instrumentation bus, a data transfer (19200 Baud) RS-485 bus interconnecting the scientific instruments with the on-board computers.

3. Dedicated serial links between
   (a) ADCS T800 and ADCS 8031,
   (b) OBC1 and ADCS 8031,
   (c) OBC1 and ADCS T800,
   (d) OBC2 and ADCS 8031,
   (e) OBC2 and ADCS T800,
   (f) OBC1 and the GPS Receiver,
   (g) OBC2 and the GPS Receiver.
4. Dedicated parallel links between
   (a) OBC1 and the RAMDisk,
   (b) OBC2 and the RAMDisk,
   (c) OBC1 and OBC2 respectively to the DSP modem,
   (d) OBC1 and OBC2 respectively to telecommand,
   (e) OBC1 and OBC2 respectively from telemetry.

The fault tolerance philosophy on the satellite dictates that each communication channel between two hardware subsystems must be backed up by at least one other communication channel. Refer to Figure B.1 for a diagram of the processor architecture and the information flow on SUNSAT.

B.1.3 Interfaces to the Environment

Refer to Figure B.2 for a ground station configuration.

B.1.4 System Modes

The satellite operation was planned to go through the following phases:

1. Pre-launch phase. The software will be frozen n weeks before shipping of the satellite for launch. This is to establish the default application that is programmed into the EPROM of each on-board computer.
2. Launch and Early Orbit Phase (LEOP). The software that will function from launch until all system functions are verified, the boom deployed, and the satellite ready for the next phase of commissioning and calibration.
3. Commissioning and Subsystem Calibration Phase. In this phase the full ADCS function and all instruments will be checked out for proper operation.
4. Payload Calibration and Performance Verification Phase. Check out of all payloads including the GPS Rx, imager, magnetometer and school experiments.
5. Main Mission Phase. Normal operation of satellite expected.

The satellite will have the following operating modes:

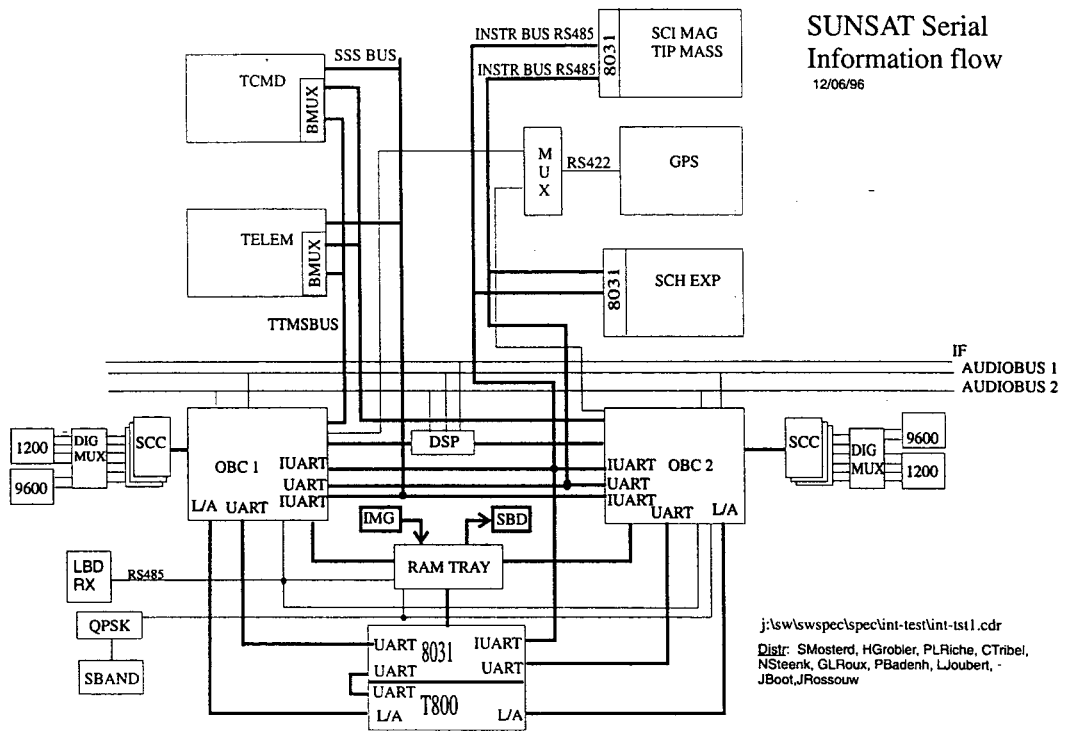


Figure B.1: Processor architecture and information flow on SUNSAT

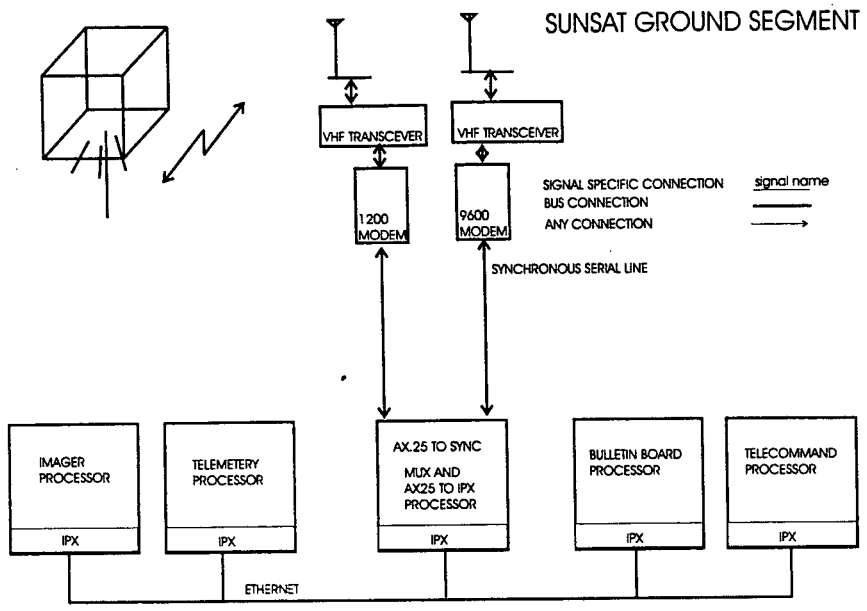


Figure B.2: Ground station architecture for integration

- Bootup mode. In this mode the satellite will function with minimum telemetry and minimum telecommand. Software will be uploaded and commissioned.
- Minimum power mode. In this mode the satellite will default to the simplest operational functions.
- Normal operating mode. In this mode the on-board computers will control the satellite and interpret and execute commands that are stored in the diary or received from an authenticated groundstation.

B.1.5 Software Component Types and Error Handling

The following types of software components are made provision for.

- A driver interfaces directly to hardware and is processor dependent.
- A server provides processor independent functions related to a specific device. It normally uses the driver which is platform dependent for the lower level services.
- A manager has its own thread of execution, can provide a status indicator without being prompted, and does have the ability to make autonomous decisions on system modes.
- A supervisor co-ordinates the overall function of a selection of managers.

Error Handling

All manager processes must have a status function. If a manager has its own thread of control, then it sends a status message containing a vector of all state variables of all its contributing parts and its own status periodically to the status collection mailbox. If the manager does

not have its own thread of control, then it has a status function which can be called by any other manager or supervisor to determine the current status of the manager. The period of inter-status transmission must be indicated in the configuration message when a manager is set up to provide regular responses.

Each manager can be configured to store any change of significant state information in the logging system, if the latter exists. Typically in the case of SUNSAT, this would be stored in a file on the file system. The telemetry function on SUNSAT is a good example of the satellite status information being stored periodically. All other systems should only store the date/time stamp and the new values of the state variables if a change in any of the significant state variables took place.

Drivers must return a result code reflecting the success of an operation when invoked by a procedure call. If a driver is an interrupt handler it would set a status flag based on its current status. This result code or flag must be read by the manager of the driver and returned as part of the manager's status message.

Reaction to Error Status

All software components must be classified whether they are mission critical or not. Error messages from mission critical processes must be sent to the overall mission control (groundstation) on a periodic basis. Error messages from processes that are not mission critical must be logged in the state variable logging system. The process must then act as it is described in its error configuration command. In the error configuration command of each process it must be clearly indicated what its expected actions are for each possible error state which can arise. The default action is fail silent (i.e. store last state change, go into idle state, and wait for activation command). The log must be readable after system failure to determine the likely causes of failure.

B.1.6 Specific Functions

Communication supervisor Co-ordinates the communication payloads on the satellite.

SCC driver Manages the 6 SCC channels at rates of 1200, 9600 and 51200 (for ground testing only) Baud.

AX.25 link level protocol server Provides reliable links between the groundstation and SUNSAT.

Receive multiplexer In place of AX.25 until it is stable and released for use. Receives signal from SCC driver via rope, consumes packet from circular buffer, looks at address field and sends the packet off to correct destination.

Transmit packetiser In place of AX.25 transmitter until it is stable and released for use. Receives packets in mailbox, breaks them up into short enough pieces, numbers them, and calls the SCC driver to transmit the packets.

Pacsat bulletin board system Bulletin board supporting the Pacsat protocols.

Terrestrial bulletin board system Bulletin board supporting terrestrial bulletin board format.

SUNSAT bulletin board system Bulletin board supporting SUNSAT standard set around May 1994. In use until other bulletin boards are stable and released. Triggered by incoming message from multiplexer, executes request, and outputs result to the port connected to the mailbox of the packetiser queue.
Status: Used for testing purposes.
Update: This system later evolved into the SatFTP system.

File system manager Consisting of a file system server using either a RAM disk driver or a memory driver for the on-board computer memory.

DSP manager Receives and stores the DSP program in the file system. Waits for commands from command and control processor to upload DSP software and to indicate the operational schedule of DSP processor.

Parrot manager Triggered by a sound file coming in. Stores sound file in file system. Also acts on the commands from command and control task to start monitoring the audio bus, and plays back after timeout or when signal is dropped. *Update: currently running on the DSP processor as it consumes too much of the on-board processor computing resources. See the analysis in Section D.1.*

ADCS supervisor Controls the modes of the ADCS system. Receives diary commands from the command and control processor.

ADCS manager Manages the interaction between the processors that run the attitude control algorithms, viz. OBC1, OBC2, ICP and the T800.

ADCS controller Control algorithms.

UART device driver The communication channel between the OBC and the ICP on the ADCS system.

L/A device driver The communication channel between the OBC and the T800 on the ADCS system.

Sensor device drivers The device drivers that read data from the sensors.

Actuator device drivers The device drivers that write data to the actuators. For both the magneto-torquers and the reaction wheels, the communication is with a remote microcontroller.

Instrument bus manager Consists of server and driver. Transmits packets according to the instrument bus protocol. Supplies standard communication manager interface to tasks which need to communicate over the bus.

Experiment manager The experiment manager commands instruments, collects data from instruments, and stores them in the file system.

Command and control supervisor Maintains a main schedule for all activities on SUNSAT. Contains overall status information of SUNSAT for the control of all other tasks. Receives schedule from the ground station for execution. The schedule is written in a diary format.

Monitor Triggered by command from SUNSAT serial bus to peek and poke memory and perform other low-level functions. Useful for debugging purposes.

SUNSAT serial bus manager Consists of a server and a driver. Transmits packets according to the SUNSAT serial bus protocol. Supplies standard communication manager interface to tasks which need to communicate over the bus.

Telecommand manager Consists of a telecommand server and a telecommand driver. Keeps an updated copy of the telecommand state of the satellite. Monitors and updates the state on commands received directly from the receiver multiplexer, the command and control supervisor, feedback received from the telemetry server and the telecommand driver.

Telemetry manager Consists of a telemetry server and a telemetry driver. Collects telemetry and transmits at a set rate to the output mailbox. Collects all telemetry as programmed and stores it in the file system for downloading as required.

Processor integrity manager Monitors the power consumption of the complete OBC board and shows alarms to the command and control task to act upon. " Washes" the memory periodically (read value and write back error corrected valued) and monitors all other hardware for degradation or failure.

RTX The kernel supplying the execution architecture of earliest deadline first data flow processing.

Bootloader A task to load in another process, locate it in memory, and link it into the task list. *Update: The bootloader can currently only replace the complete software image of the on-board processor.*

Boot code The initial code executed in the boot PROM to initialise the processor, to execute survival mode functions and to upload new software.

B.1.7 Overall Architecture

See Figure B.3 for the software architecture of SUNSAT. This diagram was drawn at approximately the same time as the brief requirement specification was written. The actual implementation currently executing might contain changes based on the experience of the software engineers involved.

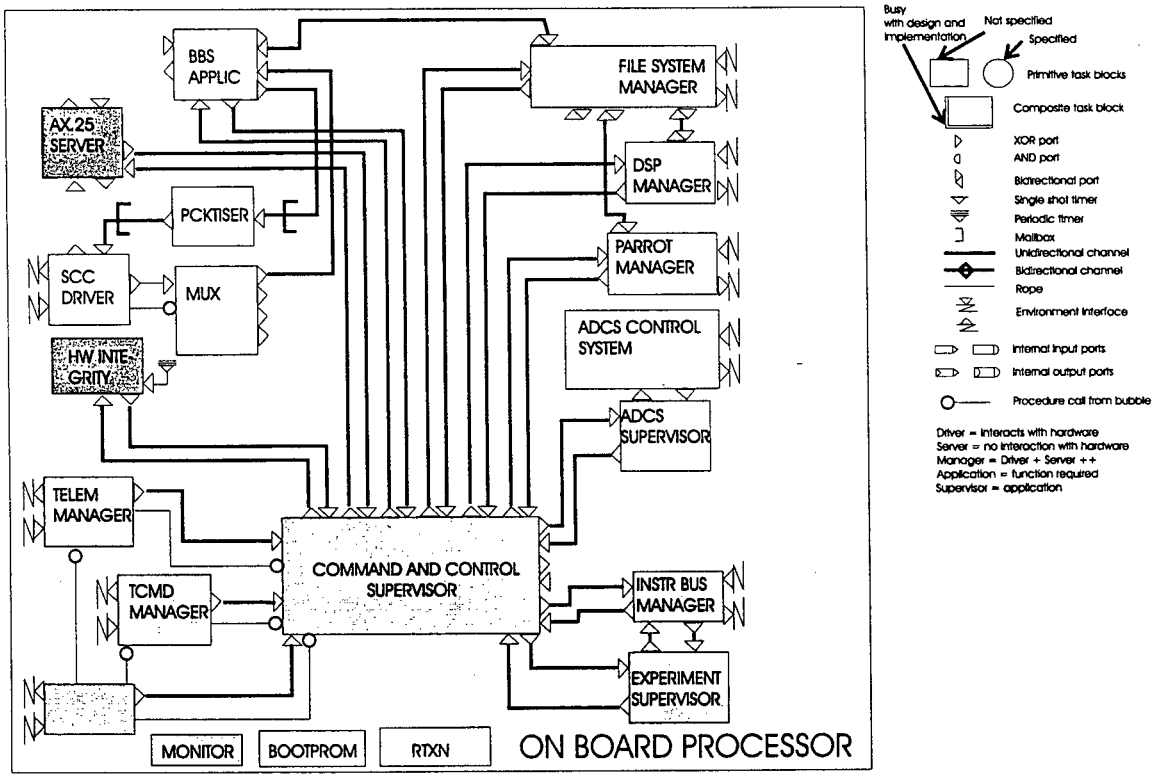


Figure B.3: Software architecture for SUNSAT

Appendix C

Feasibility Test Program in Matlab

This appendix contains the Matlab mfiles for task feasibility testing. See Chapter 4 for the tests which must be performed to determine whether a task set is feasible.

The mfile proutil.m calculates the processor utilisation for a task set $\tau = \{T_1, T_2, ..., T_n\}$. In the following function, c_i is the computational time for task T_i and p_i is the shortest inter-invocation period of task T_i.

$$(1)\quad \sum_{i=1}^{n} c_i/p_i \leq 1,$$

The mfile maxdecis.m uses the mfile skil.m to calculate the blocking that a task will undergo by tasks with a longer period that the task being tested. The follow inequality is tested for all tasks sorted in non-decreasing order by period:

$$(2)\quad \forall k, 1 \leq k < n; p_k \geq MAX_{i:p_i>p_k}\left(c_i + MAX_{0<l<p_i-p_k}\left(-l + \sum_{j=1}^{i-1}\lfloor (p_k + l - 1)/p_j \rfloor c_j\right)\right)$$

```
%processor utilisation calculation
%=================================================================
%proutil.m

function f=proutil(tau)

%calculates the processor utilisation for task set tau

f = sum(tau(:,1)./tau(:,2))

plot((tau(:,1)./tau(:,2))*100)

%blocking calculation
%=================================================================
%maxdecis.m
%function d = maxdecis(tau)

%calculates the decision matrix with maximum lag values for a
```

```
%specific task number k
%where k: for k = 1:n-1

n = max(size(tau)); % the number of rows in tau = number of tasks

%for all the tasks k from 1 to n-1, last task has longest period
for k = 1:n-1

  clear d

    for i = k+1:n,
      tau(i,2)
      tau(k,2)
      endff = (tau(i,2) - tau(k,2))

      if (endff > 0) % if periods the same, no check
       for l=1:endff,
         d(l,(i-k)) = skil(tau, k, i, l);
       end
      end %if
    end

  if (endff > 0)
   maxlag(k) = max(max(d))  % find the maximum vector in matrix d and
  end %if                   % then the maximum in this vector
  %for each task Tk store the largest blocking value in vector maxlag(k)

end

%=================================================================
%skil.m
function f = skil(tau, k, i, l)

%calculates f = skil(tau, k, i, l) as per Section 4.2
%tau is task matrix, k'th task, i'th , l = lag

for j = 1:i-1
  t(j) = floor((tau(k,2) + l -1)/tau(j,2)) * tau(j,1);
end

S = tau(i,1) - l + sum(t);

f = S;
```

Appendix D

Case Studies

D.1 Interrupt Sources on SUNSAT

The following analysis explores the expected maximum interrupt load on the 80C386EX on-board processor related to interrupt handling. The maximum processor capacity is calculated that need to be allocated to the pre-emptive task set. The interrupt sources were mostly assigned to minimise hardware latency for the most frequent interrupts. This is an approximation of the rate monotonic assignment scheme, as the most frequent interrupts would in most cases be the ones with the shortest inter-arrival times. The measurements of execution duration were reported in [31]. The interrupt sources are listed in the table in Figure D.1. The entries *(M, S, E)* in the second column labelled bf PIC no correspond to one of three Programmable Interrupt Controllers (PICs) that are connected in cascade mode. Here is a brief description for each of the interrupt sources:

1. Slave Programmable Interrupt Controller Input (IRQSlavePIC)
   This interrupt input is the OR combination of all interrupts on the second level Programmable Interrupt Controller (PIC) called *S*. The input rate is the sum of all simultaneously enabled interrupts on the second PIC. The interrupts on this input do not require any execution time as the execution time is accounted for under each of the individual interrupts.

2. SUNSAT Serial Bus (IRQSSBus)
   The SUNSAT Serial Bus operates at 9600 Baud with 8 bit data, no parity and one stop bit. With the start bit included, the maximum interrupt rate is thus $9600/10 = 960Hz$. Messages with the telecommand and telemetry systems are exchanged on this bus. The telecommand messages have a maximum length of 9 bytes and the telemetry messages a maximum length of 256 bytes.

3. ADCS Serial Channel (IRQADCS)
   A serial channel operating at 9600 Baud with 8 data bits, no parity and one stop bit, is connected to the Interface Control Processor (ICP) of the ADCS. The ICP can be instructed by the OBC2 (386EX processor) to send commands to the magneto-torquers. The ICP updates the OBC2 with the current sensor information in a 49 byte message every second.

4. Serial Communication Controllers (SCCs)
   Three SCC devices have two channels each that can operate at 9600 Baud maximum

which means 1200 interrupts per second. With three 9600 Baud modems and seven 1200 Baud modems the maximum data rate that the on-board computer can see is three times 9600 Baud plus three times 1200 Baud. This translates to a maximum interrupt rate from an SCC supporting one 1200 Baud channel and one 9600 Baud channel of 1350 interrupts per second. Recent increases in modem performance led to 19200 Baud communication on one SCC that translates to 2400 interrupts per second. This is on one channel only.

5. (IRQRAMDisk), (IRQ2Meg)
   Not used.

6. Kernel Timer Interrupt (IRQTimer1)
   A programmable timer on the on-board computer is programmed to provide a regular interrupt from which the kernel clock is updated. The timer interrupts the processor at a rate of 50 Hz, i.e. every 20 miliseconds.

7. (IRQTimer2), (IRQDRAM)
   Not used.

8. Second Slave Programmable Interrupt Controller Input (IRQ2SlavePIC)
   This interrupt input is the disjunction of all interrupts on the third level Programmable Interrupt Controller (PIC) called *E*. The input rate is the sum of all simultaneously enabled interrupts on the third PIC.

9. ADCS Link Adapter (IRQLA)
   This is a parallel to serial converter to communicate with the Transputer communication link. The expected maximum rate of 200 kByte/second with which the 386EX can write to the parallel interface is significantly lower than the link rate of almost 1 Mbyte/second. The information transferred over this link includes mode change commands to the ADCS, set points to the ADCS, and the reception of status messages from the ADCS. The status messages occur every second with a message length of 49 bytes. The commands to the ADCS are sporadic according to mode changes. This communication only takes place when the Transputer is performing the ADCS function.

10. (IRQWatchDog)
    Not used.

11. DSP Parallel Data Interface (IRQDSP)
    IRQDSP is an interrupt line from the DSP to the on-board processor indicating that it has data to transfer across its 8 bit data bus which is addressable with 4 address lines. The maximum rate at which the DSP can generate demodulated data is 150 kHz. A more realistic application would be to implement a 38400 Baud modem on the DSP which, with the ability to transfer 8 bytes per interaction, translates to a maximum rate of 600 interrupts per second.

12. Parallel Telemetry Interface (IRQTelem)
    The telemetry system collects the status information on the satellite at a rate of 150 or 1200 samples per second. It interrupts the on-board processor at this rate to read the current sample.

13. Parallel Telecommand Interface (IRQTCMD)
    The on-board computer can set telecommand signals through a sequence of write instructions to the telecommand latch. Any telecommand instruction that was successfully received by the telecommand system is available to be read by the on-board computer on the incoming telecommand latch. The readiness of a successful telecommand is signaled by an interrupt. The fastest rate at which telecommand messages can be sent by the

groundstation is 9600/8/6 = 200 commands/second. This is a system limit and will only be maintained for a short period.

14. Parallel Imager Interface (IRQImager)
The on-board computer can configure the imager parameters for an imaging operation. Once the imager is running, it provides an interrupt to the on-board computer when new inline telemetry data are available. The imager interrupt is only generated during imaging sessions while other services such as the DSP function are deactivated. Thus, it does not contribute to the worst case interrupt load.

15. Asynchronous Communication Controllers (IRQUARTS)
Three asynchronous serial channels are supported by 3 UARTS on the on-board computer. Two of the serial channels are to support the instrumentation bus operating at 19200 Baud with 8 data bits, 1 parity bit and 1 stop bit. This translates to a maximum interrupt rate on one channel of 19200/11 = 1.745 kHz. This is the maximum rate from the instrumentation bus as the dual channels are stand-bys for each other. The message length on the instrumentation bus is 32 bytes.

    The third asynchronous serial channel is connected to the GPS receiver experiment. The channel operates at 19200 Baud with 8 data bits, 0 parity bits and 1 stop bit. This gives a maximum data (and hence interrupt rate) of 19200/10 = 1.92 kHz.

16. EDAC Memory Error Interrupt (IRQEDAC)
The Error Detection And Correction (EDAC) circuitry detects and corrects single event upsets without interrupting the processor. Double event upsets are, however, not correctable and the processor is interrupted. In this situation the processor should take immediate remedial action such as re-boot. The EDAC interrupt source therefore does not add to the real-time load on the on-board processor.

17. Audio Bus Sampling and Play Back (IRQAtoD)
An Analogue to Digital Converter (AtoD) and a Digital to Analogue Converter (DtoA) are both connected to the audio bus of the satellite and the on-board computer. The on-board computer can, thus, read and write to the audio bus, for example implementing a speech record and playback function. Experiments have revealed that the interrupt handler consumes 60 microseconds per sample for sampling or playing back a sample. A sampling rate of 8 kHz is generally considered acceptable for acceptable audio quality.

The expected execution duration for the different levels of complexity of tasks to handle the interrupts is as follows.

Task description	Execution time (microseconds)
Absolute shortest processing time	11
Send or receive byte on serial channel	35
Sampling and playback of audio	60

A number of different pre-emptive task sets is listed in the table in Figure D.1. The column labelled *Scene 1* represents the pre-emptive task load if all interrupts are active under a maximum load condition. The contribution to the processor load by the IRQADC interrupt load at 48% of processor utilisation is overshadowing the other contributions. The total processing load for this task set is 92.7% which means that it is already over the 69% processor utilisation limit under which the RMA priority assignment algorithm guarantees that each task will meet its deadline. This task set is thus not feasible.

Preemptive task set on SUNSAT

No	PIC No	Description	Rate Hz	Duration micro sec	Period(pi) micro sec	Scene1 ci/pi	Scene2 ci/pi	Scene3 ci/pi
1	M0	IRQTimer0	0	15	0			
2	M1	IRQRTC	0	35	0			
3	M2	IRQSlavePIC	13115	0	76			
4	M3	IRQADCS	960	35	1042	0,0336	0,0336	
5	M4	IRQSSBus	960	35	1042	0,0336	0,0336	0,0336
6	M5	IRQSCC1	1200	35	833	0,0420	0,0420	0,0420
7	M5a	IRQSCC1	150	35	6667	0,0052	0,0052	0,0052
8	M6	IRQSCC3	1200	35	833	0,0420	0,0420	0,0420
9	M6a	IRQSCC3	150	35	6667	0,0052	0,0052	0,0052
10	M7	IRQSCC2	1200	35	833	0,0420	0,0420	
11	M7a	IRQSCC2	150	35	6667	0,0052	0,0052	
12	S0	IRQRAMDisk	0	0	0			
13	S1	IRQ2Meg	0	0	0			
14	S2	IRQTimer1	50	60	20000	0,0030	0,0030	0,0030
15	S3	IRQTimer2	0	0	0			
16	S4	IRQDRAM	0	0	0			
17	S5	IRQ2SlavePIC	12915	0	77			
18	S6	IRQLA	150	35	6667	0,0052	0,0052	0,0052
19	S7	IRQWDog	0	100	0			
20	E0	IRQDSP	600	100	1667	0,0600	0,0600	
21	E1	IRQTelem	150	35	6667	0,0052	0,0052	0,0052
22	E2	IRQTCMD	200	35	5000	0,0070	0,0070	0,0070
23	E3	unnamed	150	35	6667	0,0052	0,0052	0,0052
24	E4	IRQImager	150	100	6667	0,0150	0,0150	
25	E5	IRQUARTS (InstrBus)	1745	35	573	0,0611	0,0611	
26	E5a	IRQUARTS (GPS)	1920	35	521	0,0672	0,0672	0,0672
27	E6	IRQEDAC	0	100	0			
28	E7	IRQADC	8000	60	125	0,4800		
		Preemptive task load on processor				0,9180	0,4380	0,2211

Figure D.1: Pre-emptive task load on SUNSAT

Removing the task servicing the IRQADC interrupt, the load is shown in the column labelled *Scene 2*. The total processor load is 44.7%. It is significant to note that the major contributors to processor load are the tasks that include byte transfer communication. This points to an area where the hardware architecture should be improved to lower the load on the processor, particularly if faster communication speeds are desired. One experiment reported in [8] indicates that with DMA transfer the time to manage one byte decreases to 2.5 microseconds compared to the 35 microseconds of the interrupt mechanism.

The column labelled *Scene 3* represents a typical task load with one full duplex 1200 Baud and one full duplex 9600 Baud communication channel operating in addition to the GPS experiment. This results in a processor load of 23% leaving a processor capacity of 77% for executing the non-pre-emptive task set on the on-board processor. The resulting 77% means that in blocking analysis the execution duration of each non-pre-emptive task should be divided by 0.77 to account for the non-available processor capacity. The above analysis assumes the worst case conditions under which the on-board processor has to operate. This is indeed the appropriate assumption if a hard real-time requirement is considered.

D.2 RTX Implementation for Disjunctive and Conjunctive Trigger Semantics

This section contains the interface of an RTX implementation of which the performance evaluation is described in [109]. This particular implementation of RTX makes provision for conjunctive trigger semantics on input ports through the *MultiNChannel* procedure call in the interface which implements 2, 3 or 4 AND input port interfaces.

```
DEFINITION MODULE RTX ;

IMPORT Procs, Channels, tRTX, SYSTEM, Target;

CONST
  MultiMsgMax = 4;

TYPE
  PortName      = tRTX.Name ;

  Channel;
  Port;
  Process;

  MicroSeconds = LONGCARD ;

  Message       = SYSTEM.ADDRESS ;

  tMsgRec       = RECORD
    msg         : Message;
    msg_size    : CARDINAL;
    user_ref    : tRTX.UserRef;
  END;

  tMsgArray     = ARRAY [0..MultiMsgMax-1] OF tMsgRec;

  TimerRef ;
  Mailbox  ;
VAR
  Debug, Debug_Time, Debug_Proc, Debug_Chan, Debug_Msg, Debug_Idle : BOOLEAN;

(*-------------------------------------------------------------------------*)
(*                              Graph definition                           *)
(*-------------------------------------------------------------------------*)

PROCEDURE CreateChannel (    channel_name       : Channels.ChannelName ;
                             channel_user_ref   : tRTX.UserRef ;
                             sending_process    : Process ;
                             receiving_process  : Process ;
                             period             : MicroSeconds ;
                             message_size       : CARDINAL ;
                         VAR channel            : Channel) ;

PROCEDURE Multi2Channel ( Channel1,
                          Channel2              : Channel );

PROCEDURE Multi3Channel ( Channel1,
                          Channel2,
```

```
                        Channel3   : Channel );

PROCEDURE Multi4Channel ( Channel1,
                          Channel2,
                          Channel3,
                          Channel4   : Channel );

PROCEDURE CreateInputPort (    port_name          : PortName ;
                               port_user_ref      : tRTX.UserRef ;
                               receiving_process  : Process ;
                               period             : MicroSeconds ;
                           VAR port               : Port) ;

PROCEDURE CreatePort (    port_name          : PortName ;
                          port_user_ref      : tRTX.UserRef ;
                          signalling_process : Process ;
                          receiving_process  : Process ;
                          period             : MicroSeconds ;
                      VAR port               : Port) ;

PROCEDURE CreateProcess (    process_name : Procs.ProcessName ;
                             thread       : Procs.Thread ;
                         VAR process      : Process) ;

(*-------------------------------------------------------------------------*)
(*                        Inter-process communiction                       *)
(*-------------------------------------------------------------------------*)

PROCEDURE Receive (VAR  message      : Message ;
                   VAR  message_size : CARDINAL ;
                   VAR  user_ref     : tRTX.UserRef) ;

PROCEDURE ReceiveMulti ( VAR Msg : tMsgArray );

PROCEDURE Send (channel : Channel ;
                message : Message ;
                size    : CARDINAL) ;

PROCEDURE Signal (port : Port) ;

(*-------------------------------------------------------------------------*)
(*                           Timers and Alarms                             *)
(*-------------------------------------------------------------------------*)

PROCEDURE SetAlarm (    user_ref       : tRTX.UserRef ;
                        notify_mailbox : Mailbox ;
                        period         : MicroSeconds ;
                    VAR timer_ref      : TimerRef) ;

PROCEDURE SetTimer (    user_ref       : tRTX.UserRef ;
                        notify_channel : Channel ;
                        period         : MicroSeconds ;
                    VAR timer_ref      : TimerRef) ;

PROCEDURE StopAlarm (timer          : TimerRef ;
                     user_ref       : tRTX.UserRef ;
                     notify_mailbox : Mailbox) ;
```

```
PROCEDURE StopTimer (timer          : TimerRef ;
                     notify_channel : Channel) ;

(*-------------------------------------------------------------------------*)
(*                              Mailboxes                                  *)
(*-------------------------------------------------------------------------*)

PROCEDURE CreateMailbox (    name        : tRTX.Name ;
                             user_ref    : tRTX.UserRef ;
                             owner       : Process ;
                             no_of_slots : CARDINAL ;
                             slot_size   : CARDINAL ;
                             period      : MicroSeconds ;
                         VAR mailbox     : Mailbox) ;

PROCEDURE PutIntoMailbox (mailbox : Mailbox ;
                          msg     : Message ;
                          size    : CARDINAL) ;

(*-------------------------------------------------------------------------*)
(*                         System initialisation                           *)
(*-------------------------------------------------------------------------*)

PROCEDURE ReassignIdleProcess (process_name : Procs.ProcessName ;
                               thread       : Procs.Thread) ;

PROCEDURE StartSystem () ;

END RTX.
```

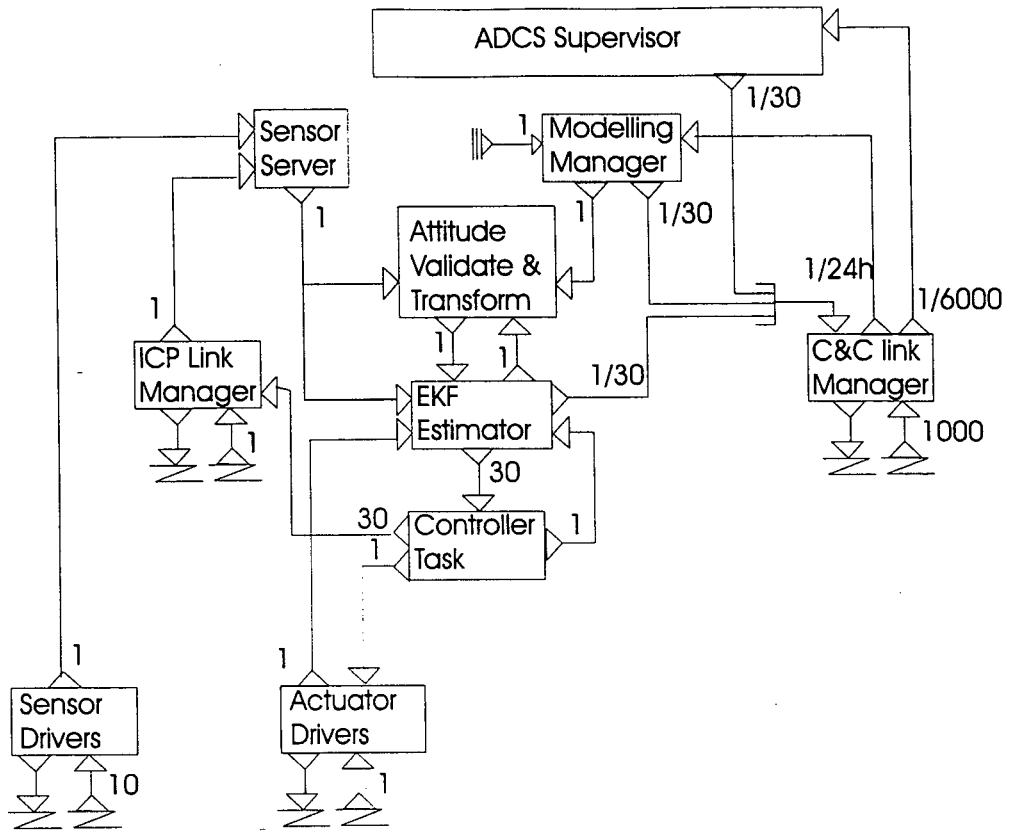


Figure D.2: An RDF of the ADCS process structure with derived token rates

D.3 ADCS Software Framework in RTX

The ADCS on SUNSAT was implemented as one monolithic software program compiled with the Rowley Modula-2 compiler and linked in the 3L environment. The ADCS software represents a task set that is periodic except for mode change messages received from the on-board data handling processor. The task set is furthermore statically schedulable as the sequence of computations is pre-determined by design. The ADCS software to date is running as the only load on a dedicated T800 processor, and there has not been a need to schedule it with the other on-board computer software.

In the case of a processor failure of the T800 and the current on-board processor (80C188EC), the 386EX processor would have to take over both the operation of house keeping, instrument management, and communication services in addition to the ADCS processor workload. In such a scenario, the tasks in the ADCS task set must be schedulable together with the rest of the on-board processor task set. For this scenario, the ADCS task load has been modeled in the RDF graph structure and a software framework in Modula-2 was developed.

The RDF graph with the set of output port rate equations solved is shown in Figure D.2. The Modula-2 source code of an RTX version of a slightly different ADCS task set framework is given below.

```
MODULE ADCS_FM;
(*************************************************************************
 *   MODULE ID:     ADCS_FM.mod
```

```
*   PATHNAME:      I:\rtxn\pc\scr\....
*   COMPILER:      TOPSPEED Modula-2 Ver 3.02
*   MEMORY MODEL: LARGE
*   FUNCTION:      Demonstrate the basic process configuration for the
*                  ADCS on a PC
*   AUTHOR:        L Joubert, adapted by S Mostert
*   CHANGE CONTROL:
*************************************************************************)

IMPORT
  RTX,
  RTXN_IO,
  IO,
  SYSTEM;
(*Terminal,Chan, Timer, Target, Strings;*)

CONST
  Model_Cal_Ref = 900;
  Model_AtVal_Ref = 901;
  Cal_AtVal_Ref = 902;
  AtVal_Est_Ref = 903;
  Est_Con_Ref = 904;
  Con_ICPOut_Ref = 905;
  Cal_ADCSMan_Ref = 906;
  Model_InMail_Ref = 907;
  Port_Ref = 908;
  External_Ref = 909;
  OBCS_InMail_Ref = 910;
  ICP_ADCSMan_Ref = 911;
  ADCSMan_InMail_Ref = 912;

VAR
  ICP_In_Process,
  ICP_Out_Process,
  Modelling_Process,
  At_Val_Process,
  Estimator_Process,
  OBCS_Comms_Process,
  DSI_Process,
  DRWI_Process,
  DMTI_Process,
  ADCS_Man_Process,
  Controller_Process,
  External_Process,
  Calibrate_Process              : RTX.Process;

  Port_Int                       : RTX.Port;

  Model_Cal_Channel,
  Model_AtVal_Channel,
  Cal_AtVal_Channel,
  AtVal_Est_Channel,
  Est_Con_Channel,
  Con_ICPOut_Channel,
  External_Channel,
  ICP_ADCSMan_Channel,
  Cal_ADCSMan_Channel            : RTX.Channel;

(*  Model_AtVal_Channel            : RTX.Multi2Channel;*)

  Model_InMailbox,
  ADCSMan_InMailbox,
  OBCS_InMailbox                 : RTX.Mailbox;

  Timer_Int,
  Start_Process                  : RTX.TimerRef;

  teller,
  Control_data,
  Estimator_data                 : CARDINAL;

  ICP_Down,
  Sensor_Error,
  Actuator_Error,
  Model_Update,
  GS_Command                     : BOOLEAN;

(*-------------------------------------------------------------------------*)
PROCEDURE ICP_In;  (*  *)
BEGIN
  RTX.Receive (mesg, mesg_size, ref);   (* Receive beginning of sec. *)
  mesg = timer_ref
  RTX.SetTimer(997,Timeout_ICP,1000000,Timeout1);

  RTX.SetTimer(999,External_Channel,1000000,Timer_Int); (*External interupt every sec.*)

  IF ICP_Down THEN
   END;
  RTX.Signal(Port_Int);  (* Begins 1 sec. interval to Model*)

END ICP_In;

(*-------------------------------------------------------------------------*)
PROCEDURE ICP_Out;  (*  *)
VAR
  mesg      : POINTER TO ARRAY [0..15] OF CHAR;
  mesg_size,
  ref       : CARDINAL;

BEGIN
```

```
  RTX.Receive (mesg, mesg_size, ref);   (* Receive beginning of sec. *)
  IO.WrStr('ICP_Out');  IO.WrLn;
  IO.WrStr('end');  IO.WrLn;

END ICP_Out;

(*-------------------------------------------------------------------------*)
PROCEDURE Modelling;  (*  *)
VAR
  mesg      : POINTER TO ARRAY [0..15] OF CHAR;
  mesg_size,
  ACK_size  : CARDINAL;
  ref       : CARDINAL;
  ACK       : CHAR;

BEGIN
  RTX.Receive (mesg, mesg_size, ref);   (* Receive beginning of sec. *)

  ACK := CHAR(0);
  ACK_size := 1;
  RTX.Send(Model_Cal_Channel,SYSTEM.ADR(ACK),ACK_size);
END Modelling;

(*-------------------------------------------------------------------------*)
PROCEDURE Calibrate;  (*  *)
VAR
  mesg      : POINTER TO ARRAY [0..15] OF CHAR;
  mesg_size,
  ACK_size  : CARDINAL;
  ref       : CARDINAL;
  ACK       : CHAR;

BEGIN
  RTX.Receive(mesg,mesg_size,ref);      (* Receive model data *)
  ACK := CHAR(0);
  ACK_size := 1;
  RTX.Send(Cal_AtVal_Channel,SYSTEM.ADR(ACK),ACK_size);
END Calibrate;

(*-------------------------------------------------------------------------*)
PROCEDURE At_Val;  (*  *)
VAR
  mesg      : POINTER TO ARRAY [0..15] OF CHAR;
  mesg_size,
  ACK_size  : CARDINAL;
  ref       : CARDINAL;
  ACK       : CHAR;

BEGIN
  RTX.Receive(mesg,mesg_size,ref);      (* Receive model data *)
  ACK := CHAR(0);
  ACK_size := 1;
  RTX.Send(AtVal_Est_Channel,SYSTEM.ADR(ACK),ACK_size);
END At_Val;

(*-------------------------------------------------------------------------*)
PROCEDURE Estimator;  (*  *)
VAR
  mesg      : POINTER TO ARRAY [0..15] OF CHAR;
  mesg_size,
  ACK_size  : CARDINAL;
  ref       : CARDINAL;
  ACK       : CHAR;

BEGIN
  RTX.Receive(mesg,mesg_size,ref);      (* Receive model data *)
  ACK := CHAR(0);
  ACK_size := 1;
  RTX.Send(Est_Con_Channel,SYSTEM.ADR(ACK),ACK_size);
END Estimator;

(*-------------------------------------------------------------------------*)
PROCEDURE ADCS_Man;  (*  *)
VAR
  mesg      : POINTER TO ARRAY [0..15] OF CHAR;
  MsgIn     : ARRAY[0..15] OF CHAR;
  mesg_size,
  ACK_size  : CARDINAL;
  ref       : CARDINAL;
  ACK       : CHAR;
BEGIN

  RTX.Receive(mesg,mesg_size,ref);      (* Receive data *)
  CASE ref OF
       911      : ICP_Down := TRUE;       (* ICP down *)
                  (*Activate timer on DSI*)
                  (*What about Cal process with choice between
                        ICP_In & DSI (Miedema)*)

       |912     : GS_Command := TRUE;   (* OBCS data available *)

       |906     : (* Scheduling of ADCS_Man process *)

       END; (*case*)

  IF ICP_Down OR Sensor_Error OR Actuator_Error THEN
    MsgIn := "Error";
    mesg_size := 5;
    RTX.PutIntoMailbox(OBCS_InMailbox,SYSTEM.ADR(MsgIn),mesg_size);
  END; (*IF*)
```

```
END ADCS_Man;

(*-------------------------------------------------------------------------*)
PROCEDURE Controller;  (*  *)
VAR
  mesg      : POINTER TO ARRAY [0..15] OF CHAR;
  mesg_size,
  ACK_size  : CARDINAL;
  ref       : CARDINAL;
  ACK       : CHAR;

BEGIN
  RTX.Receive(mesg,mesg_size,ref);       (* Receive model data *)
  ACK := CHAR(0);
  ACK_size := 1;
  RTX.Send(Con_ICPOut_Channel,SYSTEM.ADR(ACK),ACK_size);
END Controller;

(*-------------------------------------------------------------------------*)
PROCEDURE OBCS_Comms;  (*  *)
BEGIN
END OBCS_Comms;

(*-------------------------------------------------------------------------*)
PROCEDURE DMTI;  (*  *)
BEGIN
END DMTI;
(*-------------------------------------------------------------------------*)
PROCEDURE DRWI;  (*  *)
BEGIN
END DRWI;

(*-------------------------------------------------------------------------*)
PROCEDURE DSI;  (*  *)
BEGIN
END DSI;

(*-------------------------------------------------------------------------*)
PROCEDURE External;  (*  *)
BEGIN
END External;

(*-------------------------------------------------------------------------*)
PROCEDURE Init;
BEGIN
  (*--- ICP_In ---*)
  RTX.CreateProcess ('ICP Comms input', ICP_In, ICP_In_Process);

  (*--- ICP_Out ---*)
  RTX.CreateProcess ('ICP Comms output', ICP_Out, ICP_Out_Process);

  (*--- Modelling ---*)
  RTX.CreateProcess ('Enviroment Modelling', Modelling, Modelling_Process);

  (*--- At_Val ---*)
  RTX.CreateProcess ('Attitude Test & Validation', At_Val, At_Val_Process);

  (*--- Estimator ---*)
  RTX.CreateProcess ('Attitude Estimator', Estimator, Estimator_Process);

  (*--- OBC_Comms ---*)
  RTX.CreateProcess ('OBCS Comms in & output', OBCS_Comms, OBCS_Comms_Process);

  (*--- DSI ---*)
  RTX.CreateProcess ('Direct Sensor Interface', DSI, DSI_Process);

  (*--- DRWI ---*)
  RTX.CreateProcess ('Direct Reaction Wheel Interface', DRWI, DRWI_Process);

  (*--- DMTI ---*)
  RTX.CreateProcess ('Direct Magnetotorquers Interface', DMTI, DMTI_Process);

  (*--- ADCS_MAN ---*)
  RTX.CreateProcess ('ADCS Manager', ADCS_Man, ADCS_Man_Process);

  (*--- Controller ---*)
  RTX.CreateProcess ('Control Law Algorithm', Controller, Controller_Process);

  (*--- Calibrate ---*)
  RTX.CreateProcess ('Sensor Calibrating', Calibrate, Calibrate_Process);

  (*--- External ---*)
  RTX.CreateProcess ('External process', External, External_Process);

 (*--- Ports ---*)
  RTX.CreatePort    ('Signal beginning of 1 sec.', Port_Ref, ICP_In_Process,
                     Modelling_Process, 1000000, Port_Int);

 (*--- Channels ---*)
  RTX.CreateChannel ( 'External 1 sec. interrupt', External_Ref, External_Process,
                      ICP_In_Process,
                      1000000,   (*-- period --*)
                      1024,  (*-- mesg size --*)
                      External_Channel);

  RTX.CreateChannel ( 'Model data to Calibrate', Model_Cal_Ref, Modelling_Process,
                      Calibrate_Process, 1000000, 1024,
                      Model_Cal_Channel);
```

```
RTX.CreateChannel ( 'Model data to At_Val', Model_AtVal_Ref, Modelling_Process,
                    At_Val_Process, 1000000, 1024,
                    Model_AtVal_Channel);

RTX.CreateChannel ( 'Sensor data from Calib to Att Val',
                    Cal_AtVal_Ref, Calibrate_Process, At_Val_Process,
                    1000000, 1024, Cal_AtVal_Channel);

RTX.CreateChannel ( 'Attitude data to Estimator', AtVal_Est_Ref, At_Val_Process,
                    Estimator_Process, 1000000, 1024,
                    AtVal_Est_Channel);

RTX.CreateChannel ( 'Estimated data to controller',
                    Est_Con_Ref, Estimator_Process, Controller_Process,
                    1000000, 1024, Est_Con_Channel);

RTX.CreateChannel ( 'Actuators commands to ICP_Out',
                    Con_ICPOut_Ref, Controller_Process, ICP_Out_Process,
                    1000000, 1024, Con_ICPOut_Channel);

RTX.CreateChannel ( 'Activate ADCS Manager',
                    Cal_ADCSMan_Ref, Calibrate_Process, ADCS_Man_Process,
                    1000000, 1024, Cal_ADCSMan_Channel);

RTX.CreateChannel ( 'Activate ADCS Manager',
                    ICP_ADCSMan_Ref, ICP_Out_Process, ADCS_Man_Process,
                    1000000, 1024, ICP_ADCSMan_Channel);

(*--- Mailboxes ---*)
RTX.CreateMailbox ('Input Model Data from OBC',
                   Model_InMail_Ref, Modelling_Process,
                   3,     (* slots *)
                   100,   (* slot_size *)
                   1000000,  (* 1 s *)
                   Model_InMailbox);

RTX.CreateMailbox ('Input Data from OBC',
                   ADCSMan_InMail_Ref, ADCS_Man_Process,
                   3, 100, 1000000, ADCSMan_InMailbox);

RTX.CreateMailbox ('Errors to OBCS from ADCS_MAN',
                   OBCS_InMail_Ref, OBCS_Comms_Process,
                   3, 100, 1000000, OBCS_InMailbox);

(*--- Variables ---*)
Control_data := 0;
Estimator_data := 0;
ICP_Down := FALSE;
Sensor_Error := FALSE;
Actuator_Error := FALSE;
Model_Update := FALSE;
GS_Command := FALSE;

END Init;

BEGIN
 Init;
 teller := 0;
 IO.WrStr('Begin'); IO.WrLn;
 RTX.SetTimer(998,External_Channel,1000,Start_Process);  (* process ICP_Comms_In *)
 RTX.SetTimer(999,External_Channel,1000000,Timer_Int);   (* external process *)
 RTX.StartSystem;
END ADCS_FM.
```

D.4 Complete RDF Satellite Model

The following example shows the RDF model of a complete satellite. See Figure D.3 for an RDF diagram of software on a satellite with instrument managers, communication services and an attitude determination and control system. The analysis of the combined processor load of this task set on a 16 MHz 386EX, requires some knowledge of the execution times for the tasks. The execution times for the attitude determination and control system (ADCS) components, are estimates based on experiments performed when the ADCS system [101] was simulated on a 386 PC with a 387 coprocessor, both operating at 16 MHz. All the other assumptions with regard to execution times, were explained in the case studies in Chapter 5 and in Section D.1 of this Appendix. The execution times for the task set has not been adjusted from the values measured on the 13MHz 188EC. This is due to the fact that the 386EX runs only marginally faster at 16MHz and that no reliable comparative measurement values are available for the 386EX to be able to scale the existing estimates and measured values. From the table in Figure D.4 the following can be deduced:

1. The processor load of the ADCS task set is close to that of the processor capacity. The ADCS task set can thus execute only on its own on the 386EX processor.

2. The total processor load excluding the ADCS task set is more than 2 times the processing capacity of the processor. This means that the actual mission operational task set can only perform a subset of all the total functions at any point in time.

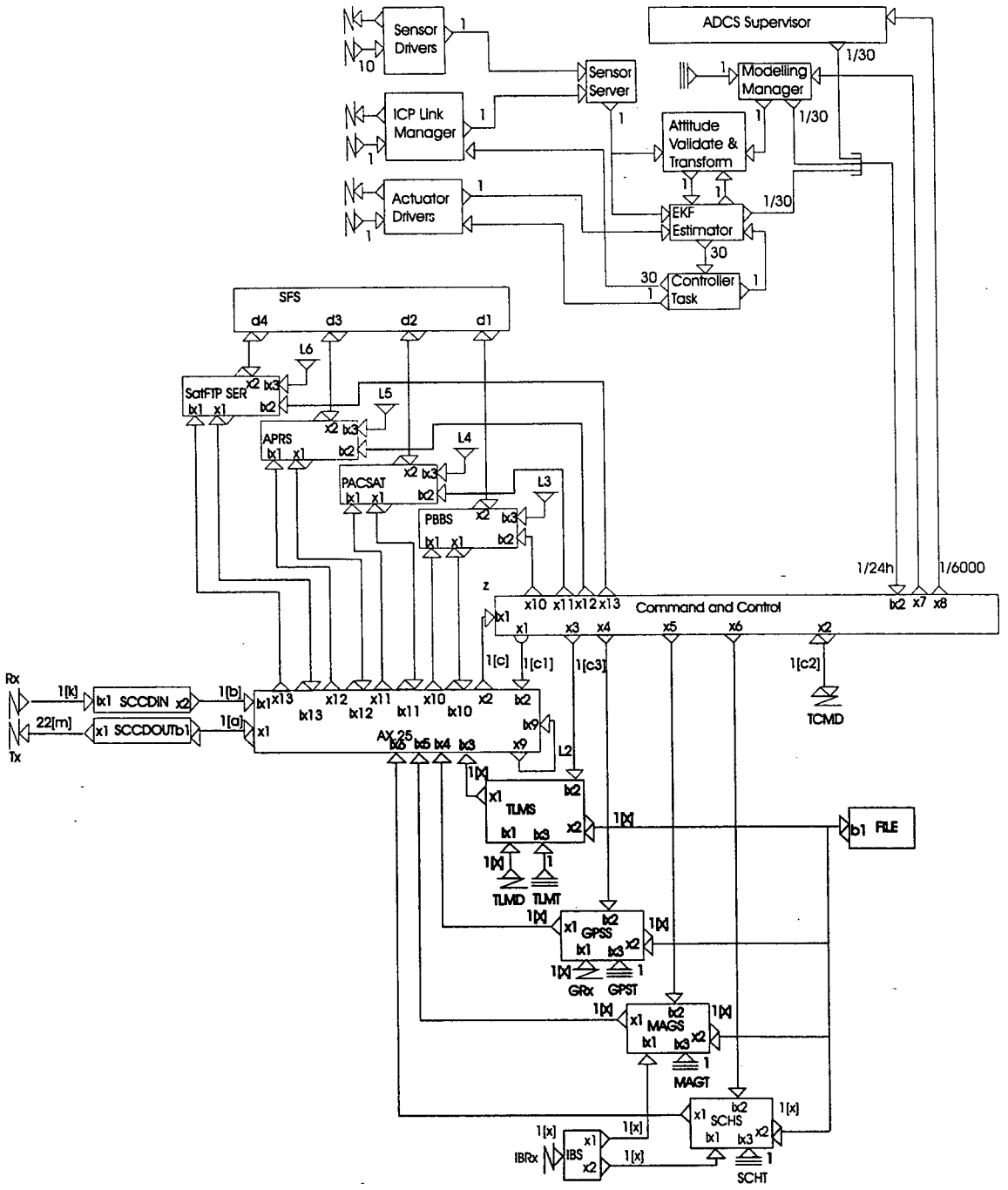


Figure D.3: RDF diagram of a complete satellite

Full satellite software with Instrument Mangers, Communication Services, ADCS

Adjustable parameters	Rate	units
BBS, APRS baud rates	1200	baud
PacSat, SatFTP baud rates	9600	baud
Bits per byte	8	bits
Instrument baud rate	19200	baud
Bytes per packet (APRS, CMD)	22	bytes
Bytes per packet (Instrument Bus)	32	bytes
Bytes per packet (Pacsat, SatFTP)	120	bytes
No of simultaneous input channels (SCC channels)	1	channels
Telemetry sample rate	150	samples/sec
Telemetry transmit timer rate	0,58824	frames/sec
realTimeRepeat	2	frames
GPS transmit timer rate	4,11765	frames/sec

Non preemptive task activation

Input no	Port name	Description	Rate /sec	Execution duration (ci)micro s	Period (pi) micro sec	ci/pi Scene 1
1	O(SCCDIN).ix1	Input rate for SCCIn Driver (now preemptive)	150	0	6667	0,0000
2	O(AX.25).ix1	Input rate from SCCIn	23,63636	927	42308	0,0219
3	O(AX.25).ix2	Input rate from C&C	0,681818	567	1466667	0,0004
4a	O(AX.25).ix3	Input rate from TLMS (realtime)	0,294118	1200	3400000	0,0004
4b	O(AX.25).ix3	Input rate from TLMS (storage)	0,588235	1200	1700000	
5	O(C&C).ix1	Input rate for C&C from AX.25	0,681818	1400	1466667	0,0010
6	O(TLMS).ix1	Input rate from TLMD	0,588235	28000	1700000	0,0165
7	O(TLMS).ix2	Input rate from C&C	0,681818	900	1466667	0,0006
8	O(TLMS).ix3	Input rate from TLMT	0,588235	28000	1700000	0,0165
9	O(AX.25).ix9	Input rate from timer L2	0,588235	683	1700000	0,0004
10	O(PBBS).ix1	Input rate from AX.25	6,818182	4000	146667	0,0273
11	O(PBBS).ix3	Input rate from timer L3	6,818182	2000	146667	0,0136
12	O(APRS).ix1	Input rate from AX.25	6,818182	4000	146667	0,0273
13	O(APRS).ix3	Input rate from timer L5	6,818182	2000	146667	0,0136
14	O(PacSat).ix1	Input rate from AX.25	10	4000	100000	0,0400
15	O(PacSat).ix3	Input rate from timer L4	10	2000	100000	0,0200
16	O(GPSS).ix1	Input rate from GRxD – assemble store frame	75	2000	13333	0,1500
17	O(GPSS).ix2	Input rate from C&C – get command	0,681818	900	1466667	0,0006
18	O(GPSS).ix3	Input rate from GPST – tx frame to ground	4,117647	28000	242857	0,1153
19	O(MAGS).ix1	Input rate from IBRxD	75	2000	13333	0,1500
20	O(MAGS).ix2	Input rate from C&C	0,681818	900	1466667	0,0006
21	O(MAGS).ix3	Input rate from MAGT	4,117647	28000	242857	0,1153
22	O(SCHS).ix1	Input rate from IBRxD	75	2000	13333	0,1500
23	O(SCHS).ix2	Input rate from C&C	0,681818	900	1466667	0,0006
24	O(SCHS).ix3	Input rate from SCHT	4,117647	28000	242857	0,1153
25	O(ADSuper).ix1	Input from C&C task	0,000114	2000	8,8E+09	0,0000
26	O(ADModMan).ix1	Input from C&C task	0,000114	2000	8,8E+09	0,0000
27	O(ADModMan).ix2	Input from Timer (Orbit propagator)	1	300000	1000000	0,3000
28	O(ADAVT).ix1	Input from SenServer and Mod Manager	1	2000	1000000	0,0020
29	O(ADEKF).ix1	Input from SenServer and Actuate feedback	1	350000	1000000	0,3500
30	O(ADContr).ix1	Input from EKF Estimator	30	5000	33333	0,1500
31	O(ADSenSer).ix1	Input from SensorDriver and ICP Comms, IGRF	1	300000	1000000	0,3000
32	O(Tx).ix1	Input from SCCDout (only TCMD ack)	23,63636	567	42308	0,0134
		Non–preemptive task load on processor				2,1125

Preemptive task set on SUNSAT

No	PIC No	Description	Rate Hz	Duration micro sec	Period(pi) micro sec	Scene1 ci/pi
1	M0	IRQTimer0	0	15	0	0,0000
2	M1	IRQRTC	0	35	0	0,0000
3	M2	IRQSlavePIC	17315	0	58	0,0000
4	M3	IRQADCS	960	35	1042	0,0336
5	M4	IRQSSBus	960	35	1042	0,0336
6	M5	IRQSCC1	1200	35	833	0,0420
7	M5a	IRQSCC1	150	35	6667	0,0052
8	M6	IRQSCC3	1200	0	833	0,0000
9	M6a	IRQSCC3	150	35	6667	0,0052
10	M7	IRQSCC2	1200	0	833	0,0000
11	M7a	IRQSCC2	150	0	6667	0,0000
12	S0	IRQRAMDisk	0	0	0	0,0000
13	S1	IRQ2Meg	0	0	0	0,0000
14	S2	IRQTimer1	50	60	20000	0,0030
15	S3	IRQTimer2	0	0	0	0,0000
16	S4	IRQDRAM	0	0	0	0,0000
17	S5	IRQ2SlavePIC	16065	0	62	0,0000
18	S6	IRQLA	1200	35	833	0,0420
19	S7	IRQWDog	0	100	0	0,0000
20	E0	IRQDSP	600	100	1667	0,0600
21	E1	IRQTelem	1200	35	833	0,0420
22	E2	IRQTCMD	200	35	5000	0,0070
23	E3	unnamed	1200	35	833	0,0420
24	E4	IRQImager	1200	100	833	0,1200
25	E5	IRQUARTS (InstrBus)	1745	35	573	0,0611
26	E5a	IRQUARTS (GPS)	1920	35	521	0,0672
27	E6	IRQEDAC	0	100	0	0,0000
28	E7	IRQADC	8000	60	125	0,4800
		Preemptive task load on processor				1,0440
		Total load on processor (> 1 not feasible)				3,1566

Figure D.4: Processor utilisation for a complete set of satellite software

Appendix E

Comparing RDF with DF*

This appendix examines an example from the signal processing domain of a synchronised dual filter being modelled in DF* [15] and then modelled in RDF. This is done to compare the modelling abilities of RDF and DF*, the latter being a contemporary data flow modelling technique. Also the semantics of the two modelling techniques are compared to determine the general applicability of the RDF architecture. The successful modelling of the dual synchronised filter with RDF will support the thesis that RDF is indeed suitable to model real problems.

E.1 Example: Dual Synchronised Filter

The following example illustrates the use of the extensions found in the DF* data flow system. The extensions are the handling of non-deterministic choice between two execution paths, and a data dependency based on a value received on an input port. It is useful to study the RDF solution to this example to understand the differences in modelling abilities between the two different paradigms. DF* is a control centric data flow language with cyclo-static properties while RDF is a data centric data flow language. In the example certain extensions to RDF are defined and tested in the context of describing the solution to the dual filter problem. However, it is shown that the extensions can be avoided by a different modelling strategy. It is also clear that the extensions do not contribute to a better understanding of the system. The conclusion is that the problem can be described in RDF in a manner that is closer to the implementation architecture than DF*.

E.1.1 Problem Statement

There is a data path with two filters. Both filters have a data input and an input for coefficients (parameters). If input data are available but no new coefficients are, the filter may process the data with the old coefficient. If multiple new coefficients are available, then the filters must select the most recent ones. The two filters must be synchronised with regard to the coefficients they use. Assume for simplicity that we want to synchronise on a sample basis. The following synchronisation is required then: if filter F1 uses the Ith coefficient (set) for processing the Kth sample, then filter F2 should also use its Ith coefficient (set) for processing the Kth sample. As a result, it is possible that F2 sometimes has to block, waiting for the needed coefficients to become available. Also because of synchronisation, it is possible that F2 may not yet use new

coefficients that are already available. Further, the data path may be pipelined. The following assumptions are made:

1. F1 and F2 have the same execution rate (number of tokens produced by F1 = number of tokens consumed by F2 on each edge between F1-F2).
2. Synchronisation is required on a sample (token) basis.

By a realisation the following requirements have to be met:

1. Coefficients do not have to be created or become available, resp., equidistantly distributed in time. The only restriction is that they are not created out of order.
2. There is no known relation between the creation time of the coefficients for F1 and the one for F2 (it is possible that coefficients 1,2,3 for F1 are already available, but only the first coefficient for F2 is available or vice versa).
3. When data are present and no new coefficient is available for filter F1, then F1 executes with the old (previous) coefficient.
4. When a new coefficient is available, F1 will use the newest coefficient for its next execution (no updates in the middle of an execution).
5. Synchronisation: if F1 uses the Ith coefficient for processing the Vth input sample, then also F2 must use its Ith coefficient for processing the Vth input sample.
6. If a new coefficient becomes available earlier than needed for F2, F2 must continue working with the old coefficient until the new coefficient is required according to the synchronisation constraint.
7. If necessary for synchronisation, F2 must block until a new coefficient becomes available.
8. Pipelining of the data path must be allowed.
9. The synchronisation overhead must be minimised.
10. If number of bits in the coefficients is large, we want to avoid copying them at a frequency equal to the data rate. However, we assume that this could be efficiently implemented if both the sending and receiving task (node) can be mapped onto the same processor. Hence, if this is the case, copying parameters at data rate is considered acceptable.

E.1.2 DF* Model and Description

The DF* model for the two filter problem consists of the data flow graph in Figure E.1 complemented by the two state machine graphs in Figure E.2 and Figure E.3. The semantics of the graphs can be deduced from the discussion of the equivalent RDF graph solving the same problem.

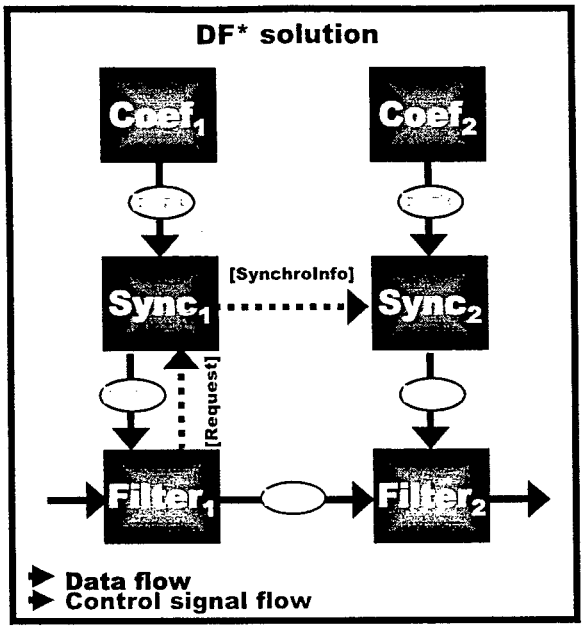


Figure E.1: DF* diagram of the dual filter problem [15]

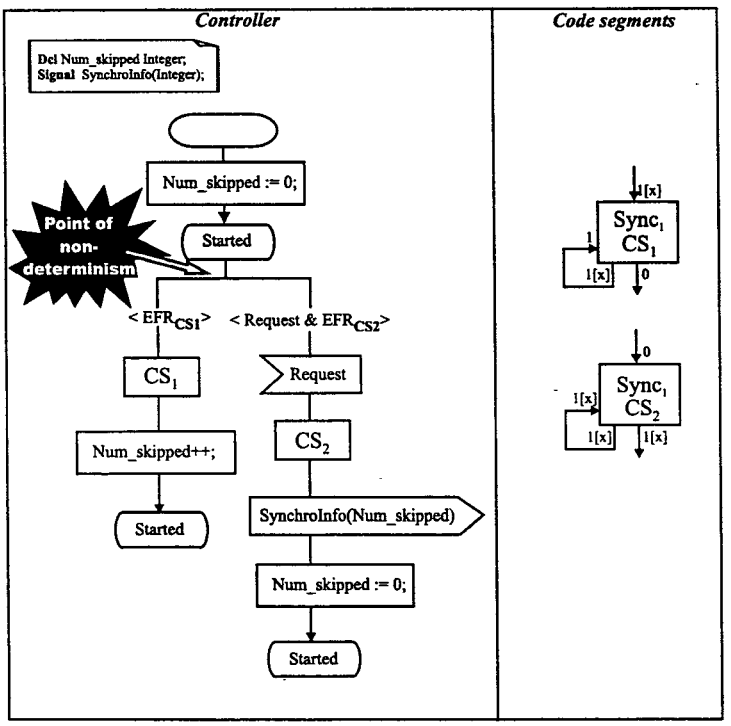


Figure E.2: State machine diagram of task node Sync1 [15]

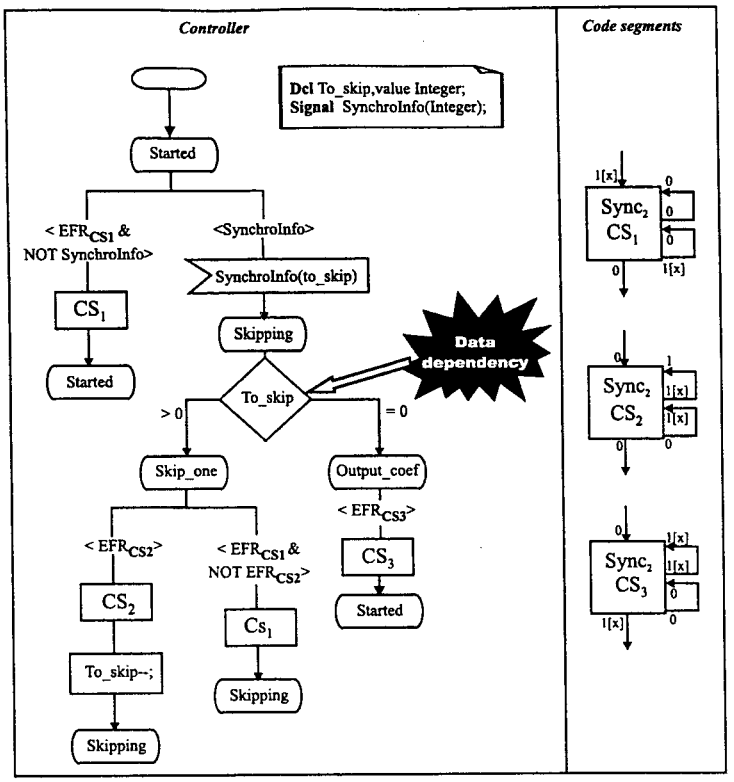


Figure E.3: State machine diagram of task node Sync2 [15]

E.1.3 RDF and State Graph Model

The RDF and associated state graph model is derived to compare the expressiveness of RDF in comparison to DF*. In the diagram of Figure E.4 the dual filter problem is modelled as a direct translation from DF* to RDF.

Referring to Figure E.2 the Synchroniser 1 task node is discussed below.

1. Code segment 1
   (a) Code segment 1 updates the current coefficient when a new one is received.
   (b) Extended Firing Rule 1:

$$Sync1.EFR1 = (ix_2.tk > 0) \wedge (ix_1.tk > 0)$$

   This corresponds to removing the coefficient from the *current coefficient queue* and replacing it with the coefficient just received on port ix_2.
   (c) Production Rule 1: $ox_1 = ix_2.val$
2. Code segment 2
   (a) Code segment 2 sends the current coefficient to Filter1 and sends an update of the number of skipped coefficient values to Filter2.
   (b) Extended Firing Rule 2:

$$Sync1.EFR2 = (ix_3.tk > 0) \wedge (ix_1.tk > 0)$$

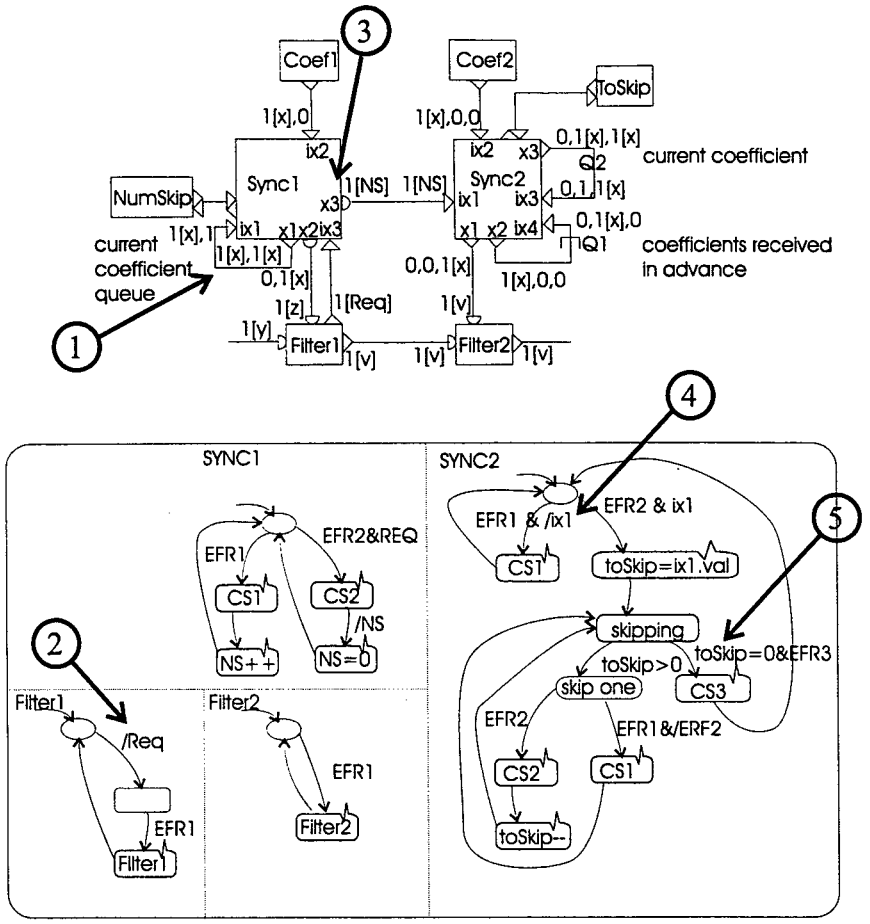


Figure E.4: RDF timing diagram of the dual filter problem

This corresponds to removing the coefficient from the *current coefficient queue* and requesting a coefficient from *Filter1*.

(c) Production Rule 2: $ox2 = ix_1.val \wedge ox3 = ix_1.val \wedge ox3 = NumSkip$ This corresponds to sending out the value of the coefficient to the storage queue and task node *Filter1*. In addition, the value of NumSkip is retrieved from the data store and sent to task node *Sync2*.

Referring to Figure E.3 the Synchroniser 2 task node is discussed below.

1. Code segment 1

   (a) Extended Firing Rule 1: $Sync2.EFR1 = (ix_2.tk > 0) \wedge (ix_1.tk = 0)$ A coefficient is available without a request for synchronisation from *Sync1*.

   (b) Code execution 1, arrival of new coefficient from *Coef2* which is added to the list of new coefficients.

   (c) Production Rule 1: $ox2.val = ix_2.val$ Store the coefficient in the holding queue.

2. Code segment 2

   (a) Extended Firing Rule 2: $Sync2.EFR2 = ix_1.tk > 0$ There is a synchronisation request from task node Sync1 waiting on input port ix_1.

(b) Code execution 2, skipping a coefficient, take the oldest coefficient off the queue and write it into the current coefficient queue. Decrement the skipping counter. Note that the feedback queues around Sync2 are used to store and re-use the current coefficient and the list of coefficients from which the next current coefficient is selected.

(c) Production Rule 2: $ox_3 = ix_4$ in the context of DF* and $ox_3 = datastore(front(coefficient\ queue)$ in the context of RDF.

3. Code segment 3

(a) Extended Firing Rule 3: $Sync2.EFR1 = if\ D(toSkip == 0)\ then\ fire\ if\ (ix_3.tk > 0 \wedge ix_1.tk > 0)$ This means that the only data flow requirement is that there is a synchronisation request on input port ix_1, and a current coefficient is available on input port ix_3. This corresponds to the two ports being considered as AND input ports. The ports can only fire if data store $toSkip == 0$. This is a data dependency on the enabling of whether the input ports may fire.

In addition, the input ports ix_3 and ix_4 also have an AND relationship when the current coefficient is updated. Thus, the input ports exhibit different trigger behaviours under different circumstances. This can be described with cyclo-dynamic RDF (CDRDF), where the semantics of the ports adapt according to the outcomes of the data dependency rules.

(b) Code execution 3, the current coefficient is sent to Filter2.

(c) Production Rule 3: $ox_1 = ix_3.val \wedge ox_3 = ix_3.val$ This corresponds to outputting the coefficient to the filter and storing it on the current coefficient data edge.

The textual version in an Occam [46] pseudocode for the internal operation of Sync2 is listed next. It indicates that the complexity of expression possible in DF* is beginning to approach what can be described in a language for programming parallelism such as Occam. It also indicates that Occam is a control flow paradigm controlling the flow of data.

```
ALT
  IF (toSkip == 0) AND input (ix1 AND ix3) {
    IF (ix1.val = 0) then
      //(provide coefficient and store for re-use)
      output (ix1.val = ix3.val) AND (ix3.val = ix3.val)
    ELSEIF (ix1.val > 0) THEN {
      // to catch up with coefficient used in Filter1
      // artificial use of queue as a more sophisticated memory
      // would allow direct retrieval of last coefficient
      D(toSkip) = ix1.val
      WHILE (toSkip > 0 AND ix4.tk > 0){
        input ix4 AND ix3
        D(toSkip)--
        output ox3.val = ix4.val
      }
    }
  }
  IF (toSkip > 0) AND input ix2{
    // wait for more coefficient values from Coef2
    output ox4.val = ix2.val
  }
```

E.2 Analysis

The fundamental difference between DF* and RDF is the separation of control and data paths in DF* and, indeed, with a different execution semantics for each. In fact, it is argued that the control data path of DF* and the state machines control the data flow paths and code segments to execute. In RDF the control and data flow are integrated with an associated state machine for each task node which is following the data flow trigger conditions. Another way to consider DF* is that the state machine enables different input port sets to fire based on the internal state of a task node and the control flow of each node. In RDF the data flow and control flow are embedded in the same communication network which makes the re-use of communication resources automatic. Specific system engineering requirements that can be modelled with DF* and the way how RDF addresses them are the following.

1. Non-determinism — conjunctive input ports
2. Data dependency — built into the output port guards
3. Event handling — constrained to environmental interfaces which provide a separated mechanism to synchronise events with the system behaviour
4. Dynamic tasks — consist of three parts, viz. dynamic code loading, dynamic data allocation, and dynamic processor loading due to varying execution demands in response to varying input rates

Data store accesses are explicitly modelled as time state nodes. The protocol on output ports and on input ports does not reflect an order, and the actual behaviour is only apparent at closer investigation of the state graph for the RDF. What is new is that a trigger condition can test whether there are no tokens on another input than itself. This removes some non-determinism from the graph. The actual state transitions can also test the value of a data store variable. Local variables in DF* have the same functionality as data stores in RDF. Particular aspects that should be noted are numbered from 1 to 5 in Figure E.4.

1. The current coefficient queue does not trigger the Sync1 task node into action. It merely acts as a temporary data store that is read when either ix2 or ix3 triggers the execution of Sync1. The input port ix1 thus forms an AND relationship with input port ix3 at one moment in time and with ix2 at another instant. The current coefficient queue is used implicitly as an ordered temporary data storage. It is the premise of RDF that memory should be shown explicitly. Furthermore, RDF does not make provision for the firing semantics as is available on a feedback queue, as this does not contribute to the timing analysis of a system. The protocol information available on the storage queues can be used exactly as it is on the data storage nodes.
2. The generation of an output token without any time control nodes initiating the token is not possible in RDF and its associated behaviour description language state graphs. Re-write the RDF graph in Figure E.4 such that a timer triggers task node Sync1 to produce the first coefficient and that task node Filter1 generates the *Req* token after the first sample was processed.
3. Consider input port ix_3. It has an OR relationship with input port ix_2. This relationship is succinctly represented in RDF with the OR relationship. In DF* similar semantics are achieved by making the port ix_3 firing only on a control signal *Req* while the input port

ix_2 fires with every data flow cycle. All data flow ports in DF* thus exhibit the AND trigger relationship, and the introduction of control flow allows a richer set of trigger conditions to be expressed.

4. The token availability status of an input port is tested in the firing condition. This corresponds to a priority selection construct similar to the PRIALT construct in Occam.

5. The data dependency shown here is based on a value stored in a data store. This translates to an input port guard based on the value of a data store.

E.2.1 Realisability of RDF Graph

It is useful to consider the differences of the semantics in DF* and RDF and the effect on the timing analysis for RDF. Assume that the guards enabling/disabling input ports are based on the value of a data store variable. This allows the input ports to have different trigger characteristics from execution cycle to execution cycle. The cyclic varying input port trigger characteristics form the basis of what could be called Cyclo-dynamic Data Flow.

Cyclo-dynamic Extension of RDF

The trigger behaviour for each input port is characterised by a protocol sequence mask. Each entry in this mask consists of the number of tokens consumed per execution cycle and an optional name by which the tokens that are consumed can be addressed. For example, the protocol sequences on each of two input ports are:

```
ix.1 : 1[x], 2[x], 1[x] (protocol sequence mask for input port ix1)

ix.2 : 1[x], 0, 0  (protocol sequence mask for input port ix2)
```

The combined trigger mask is obtained by performing the AND operation on the two protocol sequence masks and getting the resulting combined protocol sequence mask:

$$1, 0, 0$$

indicates behaviour on the ports as follows: AND port, OR port, OR port.

Definition 64 (Protocol sequence mask) *The protocol sequence mask is a vector for each input port with a 0 entry if no tokens are input in that cycle and an entry* > 1 *indicating that so many tokens are input from that port in that cycle.*

E.2.2 Timing Analysis of Cyclo-dynamic Augmented Graph

The worst case execution duration suitable for an analysis of the system, is the one that occurs under maximum load conditions. It is not necessarily the longest execution duration in the design graph as that execution duration might only occur during failure conditions when the processor load is low due to other constraints. For a cyclo-dynamic graph the triggering is through a sequence of messages with possibly different protocols. To find the worst case scenario, the timing dominant protocol sequence must be determined and the graph analysed for this case.

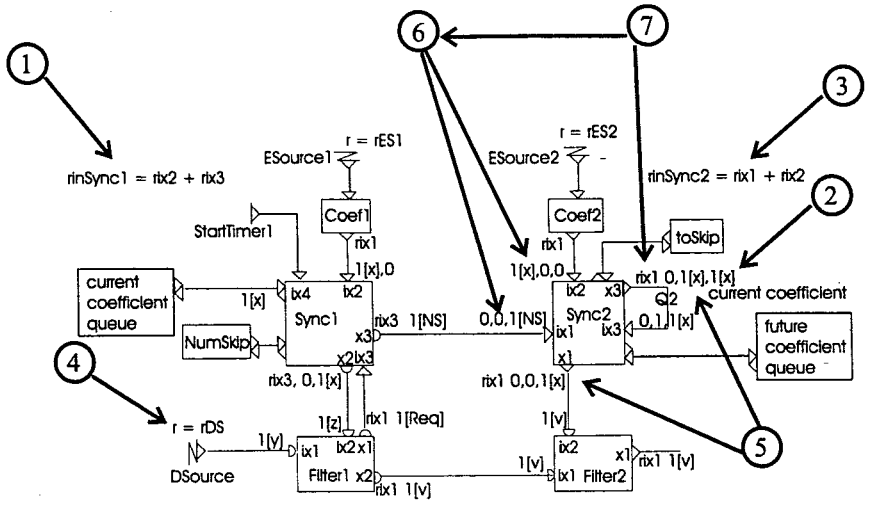


Figure E.5: RDF diagram of the dual filter problem suitable for timing analysis

Definition 65 (Timing dominant protocol sequence) *That protocol sequence which is to execute the most times in the operation of the task node.*

Analysing a graph for realisability requires that each of the output ports has its transmission rate equation solved. The inverse of the input transmission rate provides the deadline by which the process must be scheduled. In the context of the example the following additional information is required.

1. There are no primary trigger sources indicated. For this purpose the following nodes are added:
   (a) An environmental interface that supplies Filter1 with samples at a rate $r1$.
   (b) An environmental interface which supplies *Coef1* with samples at a maximum rate $rc1$ and a similar environmental interface for *Coef2* at the same rate $rc1$. As coefficients are supplied from a source with sporadic behaviour, the instantaneous rates can be different.
   (c) A single shot time to trigger node *Sync1* to supply the first coefficient for node *Filter1* as required to synchronise the two nodes.

See Figure E.5 for the changes made to the RDF to enable timing analysis. Also note that the feedback queue to store coefficients received in advance has been replaced by a data store with the same functionality.

Solving the Transmission Rate Equations

The following features of the dual filter problem's solution have to be looked at more carefully to determine the influence on timing: non-determinism, multiple code segments to execute, and data dependency in task node *Sync2*.

1. Non-determinism on the input has no effect on timing analysis.

2. Look for worst case rates, the others can be considered only if the modes in which they are present would have an impact on other task sets on the same processor.

3. The data dependency in *Sync2* results in either the system working at full rate providing coefficients or accepting new coefficients, which means that task node *Filter2* cannot run at that time or it is busy clearing the queue of coefficients received in advance, which means in turn that task node *Filter2* cannot run at that time. As the updating of coefficients represents data movement operations only, it does not constitute a significant processor load. In addition, *Filter2* is not executing which means that the processor resources available for *Filter2* are spare to use. The worst case situation is then with *Filter1* and *Filter2* running at the maximum incoming rate with the coefficients being updated at their maximum rate.

4. In task node *Sync2* or o_{Sync2}, the synchronisation of the coefficients is not expected to take place for every sample of the filter data. Then, the time dominant protocol sequence is receiving the synchronisation message from *Sync1* and, if $numberSkipped = 0$, retrieve the current coefficient and send it to *Filter2*.

Note the following on the annotated Figure E.7:

1. The input rate of *Sync1* is the sum of the inputs from Filter1 and Coef1. This is the rate at which Sync1 is to be scheduled in the worst case.

2. An output on X3 is generated for each coefficient used or updated, but not both simultaneously.

3. The input rate of *Sync2* is not a function of X_3 as it obeys an AND relationship with input port ix_1.

4. The input rate source into the system is from data source *DSource*. The other two input rate sources are *ESource*1 and *ESource*2, respectively. All transmission rate equations are to be solved in terms of these three time trigger sources.

5. In the strict data flow sense, the output rates for port x_3 and output port x_1 are the same. In the worst case timing sense, the only rate that matters is the one for *Filter2* to process samples at the rate of the *DSource*. It is interesting to note the different semantics for the three protocol sequences. In the first protocol sequence, no output is produced on either output port. In the second protocol sequence, an output is only produced on output port x_3 corresponding to an OR relationship between x_1 and x_3. In the third protocol sequence, both x_1 and x_3 produce outputs corresponding to the AND relationship between these two ports. This is the time dominant protocol sequence which dictates the worst case timing behaviour of task node *Sync2*.

6. There is an OR relationship between input ports ix_1 and ix_2. The fact that there is only one input protocol on ix_1 is equivalent to a complete OR relationship.

7. There is an OR relationship between all protocol sequences of input ports ix_2 and ix_3 as the result of an AND of the protocol sequence of ix_2 and ix_3 which is empty.

The input port $Sync2.ix_1$ is modified to have an input protocol sequence $0, 0, 1[NS]$ from its only $1[NS]$ protocol sequence. This now corresponds correctly to all firing combinations except for the output port $Sync2.x1$ which only emits a token if $[NS] = 0$. Thus, there is a data dependency not visible on the RDF graph. It is, however, visible in the state graph description

Output port	Input rate	Output rate function	Output rate
Filter1.x1	$r_{ix1} = r_{ix2}$	$f(r_{ix1}) = 1/xr_{ix1}$	$f(r_{ix1}) = r_{ix1}$
Filter1.x2	$r_{ix1} = r_{ix2}$	$f(r_{ix1}) = 1/xr_{ix1}$	$f(r_{ix1}) = r_{ix1}$
Sync1.x2	$r_{ix2} + r_{ix3}$	$f(r_{ix2}, r_{ix3}) = 1/xr_{ix3}$	$f(r_{ix2}, r_{ix3}) = r_{ix3}$
Sync1.x3	ditto	ditto because both are AND ports	$f(r_{ix2}, r_{ix3}) = r_{ix3}$
Coef1.x1	$r_{ix1} = r_{ES1}$	$f(r_{ix1}) = 1/xr_{ix1}$	$f(r_{ix1}) = r_{ES1}$
Filter2.x1	$r_{ix1} = r_{ix2}$	$f(r_{ix1}) = 1/xr_{ix1}$	$f(r_{ix1}) = r_{ix1}$
Sync2.x1	$r_{ix1} = r_{ix3}$	$f(r_{ix1}) = 1/xr_{ix1}$	$f(r_{ix1}) = r_{ix1}$
Sync2.x3	$r_{ix3} = r_{ix3}$	$f(r_{ix3}) = 1/xr_{ix3}$	$f(r_{ix3}) = r_{ix3}$
Coef2.x1	$r_{ix1} = r_{ES2}$	$f(r_{ix1}) = 1/xr_{ix1}$	$f(r_{ix1}) = r_{ES2}$

Figure E.6: Table with solved output rate equations

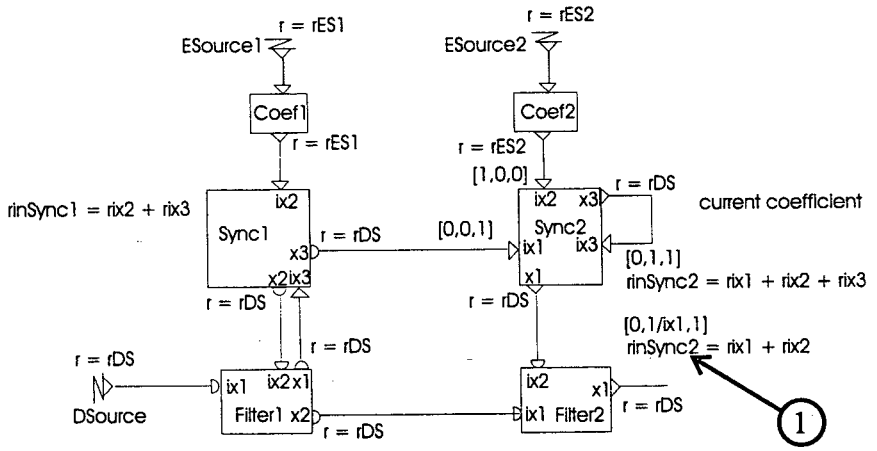


Figure E.7: RDF timing diagram of the dual filter problem

of the behaviour of node *Sync2*. The influence of the data dependency on the time dominant protocol sequence (worst case scenario) can be ignored as it restricts the system to run at its full rate. It will, however, be shown later that the data dependency knowledge leads to an improved knowledge of the worst case scenario, which leads to a smaller worst case demand on the processor. The solved equations in terms of the input source rates are shown in Figure E.6. As all output rate equations can be solved, the system is realisable.

Using Data Dependency for Timing Model Refinement

The timing relevant RDF is shown in Figure E.7. Note that the data dependency can be made visible as is shown at annotation 1. There it is shown that, although input port ix_3 can fire in an OR relationship with input ports ix_1 and ix_2, if it does fire, then port ix_1 is disabled from firing. The resulting input rate of messages/tokens into node *Sync2* is, thus, the sum of $r_{ix1} + r_{ix2}$, only.

The rates at which the tasks must process incoming messages/tokens are shown in Figure E.8. The minimum-inter token arrival period is indicated for each of the individual input ports. The sum of these input rates determines the maximum rate at which the task node is to be scheduled to process all these input messages in time.

Task node	Input rate
Filter1	$r_{in} = r_{ix1} = r_{DS}$
Sync1	$r_{in} = r_{ix2} + r_{ix3} = r_{ES1} + r_{DS}$
Coef1	$r_{in} = r_{ix1} = r_{ES1}$
Filter2	$r_{in} = r_{ix1} = r_{DS}$
Sync2	$r_{in} = r_{ix1} + r_{ix2} = r_{DS} + r_{ES1}$
Coef2	$r_{ix1} = r_{ES2}$

Figure E.8: Table with output rate functions

The next step in the timing analysis requires execution times for each of the task nodes on (a) particular processor(s). The execution times with the scheduling periods allow for processor load analysis and to determine whether the task set can be executed without missing any of the task deadlines.

E.2.3 Feasibility

For feasibility analysis, that is to determine whether the task set will execute timely on one of more processors, the computational times for each of the tasks on the target hardware is required. This is not taken further here as the goal was to test the modelling capability and to evaluate the influence of additional architectural features on the ability to analyse temporal realisability.

E.3 Re-visiting the Behaviour of the Dual Filter

Assume that *Coef2* does not produce any coefficients. The queue between *Filter1* and *Filter2* will contain a sequence of filtered values with the queue between *Sync1* and *Sync2* containing a sequence of how many values to skip between each sample and the previous sample. As long as this number is 0 *Sync2* will supply the correct coefficient to *Filter2*. Otherwise, *Sync2* will be waiting on input port is_2 for coefficient values until the index of the coefficient matches the index of the coefficient used in *Filter1* for that particular sample.

Alternative Solution

An alternative and simplified RDF which only shows the interaction required is shown in Figure E.9. Local variables are all stored in data stores and all interaction visible is mandatory for analysis of the system. There is still the case that input ix_1 is ignored as long as the coefficients are not up to date.

Single Function RDF Ports

An alternative solution is to keep the list of how many values to skip for each sample in a data store. Then every message on input port ix_1 can be processed as it arrives while maintaining the integrity of the solution. This preserves the semantics of the single relationship input port,

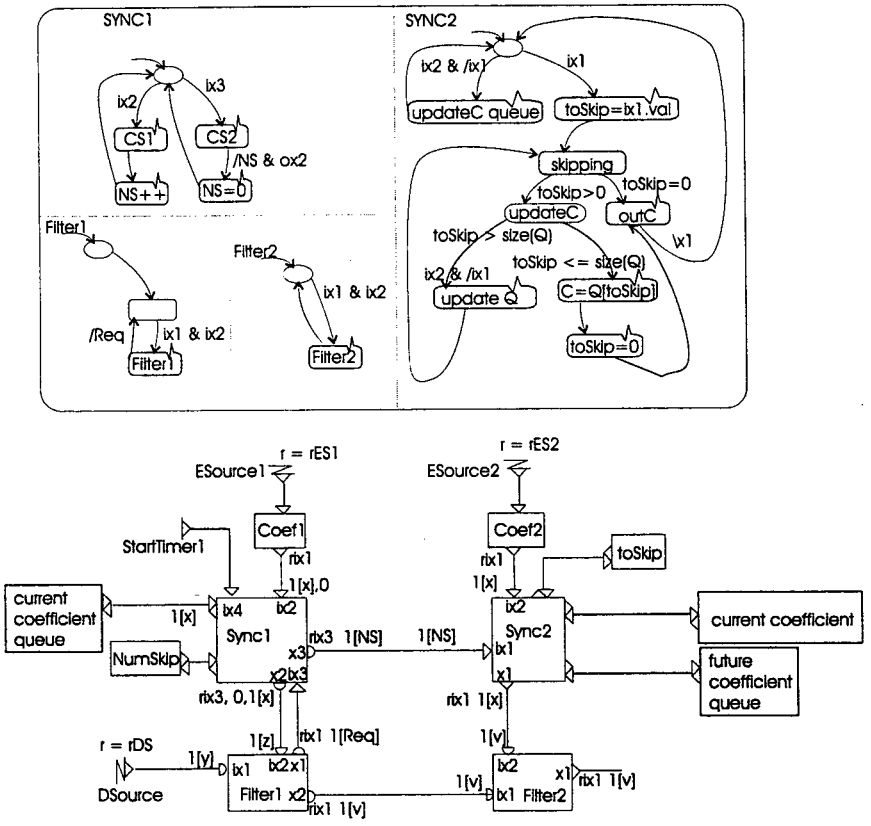


Figure E.9: Simplified RDF diagram of the dual filter problem

viz. the port is only an AND or OR port and does not change its function from one execution cycle to the next. The memory is then moved from the asynchronous edge to being managed as a data store by task node *Sync2*. Thus, the semantics of single relationship input ports are strong enough to represent the same system as the DF* system.

E.3.1 The Semantics of DF*

Adapting the semantics of RDF to completely match those of DF* has a number of implications which motivate against the semantics of DF* being incorporated in RDF, viz.:

1. The input port guards might cause the receiving task node to miss a system level deadline. The RDF solution would rather receive all and internally process different protocol sequence semantics.
2. Making the complete semantics visible in RDF causes the graph to be difficult to read and defeats the purpose of a graphical language. This is primarily a side effect of the data dependency influencing the firing enabled input port set. Compared to RDF, which can also accept multiple protocols, the difference is that any message adhering to the protocol specification is always accepted. In RDF data dependency is allowed, but not on changing the firing semantics of input ports.

E.4 Conclusion

The goal of this appendix was to compare the expressability of RDF with that of DF* by modelling the same system. In the comparison process some extensions to the semantics of RDF were investigated. The following differences between RDF and DF* are noted:

1. DF* can apply different trigger pattern templates with each execution cycle of a task node. RDF can accept any combination of messages per execution cycle.
2. DF* relies on the implicit memory that is available on communication channels for storing temporary data, while RDF can use the same mechanism or use a data store for storing temporary data.
3. The addition of a state machine description for the internal behaviour of a task node for both DF* and RDF leads to the same amount of expressiveness. The SDL notation used in DF* might be easier to comprehend visually than the state graph notation.

In conclusion, DF* is better suited to digital signal processing systems with a recurring regular pattern of data flow, while RDF is better suited for dynamic data flow systems with no recurring pattern of data flow. Both modelling environments have a preferred application domain each. RDF has the additional advantage of mapping directly to an implementation architecture corresponding one-to-one with the design architecture.