Downloaded from orbit.dtu.dk on: Dec 20, 2017

Technical University of Denmark

Formal methods for design and simulation of embedded systems

Jakobsen, Mikkel Koefoed; Madsen, Jan; Hansen, Michael Reichhardt

Publication date:
2013

Document Version
Publisher's PDF, also known as Version of record

Link back to DTU Orbit

Citation (APA):
Jakobsen, M. K., Madsen, J., & Hansen, M. R. (2013). Formal methods for design and simulation of embedded systems. Kgs. Lyngby: Technical University of Denmark (DTU). (PHD-2013; No. 289).

DTU Library
Technical Information Center of Denmark

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal

If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

Formal methods for design and simulation of embedded systems

Mikkel Koefoed Jakobsen

Kongens Lyngby 2012
IMM-PHD-2012-289

Technical University of Denmark
Informatics and Mathematical Modelling
Building 321, DK-2800 Kongens Lyngby, Denmark
Phone +45 45253351, Fax +45 45882673
reception@imm.dtu.dk
www.imm.dtu.dk

IMM-PHD: ISSN 0909-3192

Summary

Cyper physical systems (CPSs) are present in many variants in our daily life. The complexity of developing a CPS is quickly increasing and the interaction between different CPSs is increasingly important. The interaction of the systems is becomming more and more fluent and seamless.

This thesis presents the development of a formal systems modelling (ForSyDe) framework for modelling CPSs. The formalism of the framework makes computer aided design (CAD) a possibility for developing CPSs. The framework consists of four models of computation (MoCs): synchronous (SY), synchronous data flow (SDF), discrete event (DE), and continuous time (CT).

Usage of the framework is demonstrated with two use cases. A company use case featuring a hearing aid calibration device and the distributed energy harvesting aware routing (DEHAR) algorithm for wireless sensor networks (WSNs). These two use cases illustrate different design challenges. With the ForSyDe framework, the use cases are expressed as homogeneous and heterogeneous models.

The company use case illustrates that the ForSyDe framework handles systems with well defined interactions very well. The WSN use case illustrates that networked systems with complex interaction are more challenging to express naturally, yet the ForSyDe framework is able to express such systems.

Resumé

Cyper fysiske systemer (CPS'er) er til stede i mange former i vores daglige liv. Kompleksiteten i udviklingen af et CPS er hurtigt stigende, og samspillet mellem forskellige CPS'er bliver stadig vigtigere. Interaktionen af systemerne bliver mere og mere flydende og usynlig.

Denne afhandling præsenterer udviklingen af et formelt modelleringsværktøj af systemer (ForSyDe) til modellering af CPS'er. De formelle aspekter af værktøjet gør Computer Aided Design (CAD) lettere at udnytte til udvikling af CPS'er. Værktøjet består af fire beregningsmodeller (MoC'er): synkron (SY), synkron datastrøm (SDF), diskret begivenhed (DE), og kontinuert tid (CT).

Brug af værktøjet eftervises med to eksempler. Et eksempel fra en virksomhed byder på en høreapparat kalibreringsenhed og den distribuerede algoritme til rutning af beskeder ved hensyntagen til energiopsampling (DEHAR) til trådløse sensor netværk (WSN'er). Disse to eksempler illustrerer forskellige design udfordringer. Med ForSyDe værktøjet, er de eksempler udtrykt som homogene og heterogene modeller.

Eksemplet fra virksomheden illustrerer, at ForSyDe værktøjet håndterer systemer med veldefinerede interaktioner meget godt. WSN eksemplet illustrerer, at netværkssystemer med komplekse samspil er mere udfordrende at udtrykke på en naturlig måde, men at ForSyDe værktøjet trods alt er i stand til at udtrykke sådanne systemer.

Preface

This thesis was prepared at Informatics Mathematical Modelling, the Technical University of Denmark in partial fulfillment of the requirements for acquiring the Ph.D. degree in engineering.

The thesis deals with different aspects of mathematical modelling of systems. These aspects cover formal modelling, static analysis, and models of computation.

The thesis consists of a report on monograph form. During the period 2009–2012 a collection of 5 research papers were written on this subject, and elsewhere published.

Lyngby, December 2012

Mikkel Koefoed Jakobsen

Papers included in the thesis

Seyed Hosein Attarzadeh Niaki, Mikkel Koefoed Jakobsen, Tero Sulonen, and Ingo Sander. Formal Heterogeneous System Modeling with SystemC *Forum on specification & Design Languages (FDL 2012)*, 2012

Mikkel Koefoed Jakobsen, Jan Madsen, Seyed Hosein Attarzadeh Niaki, Ingo Sander, Jan Hansen. System level modelling with open source tools *Embedded World Conference*, 2012

Michael R. Hansen, Mikkel Koefoed Jakobsen, Jan Madsen. Modelling of Energy Harvesting Aware Wireless Sensor Networks *Sustainable Energy Harvesting Technologies - Past, Present and Future*, 2011

Mikkel Koefoed Jakobsen, Jan Madsen, and Michael R. Hansen. DEHAR: A distributed energy harvesting aware routing algorithm for ad-hoc multi-hop wireless sensor networks. *In World of Wireless Mobile and Multimedia Networks (WoWMoM)*, 2010

Mikkel Koefoed Jakobsen, Jan Madsen, Michael R. Hansen Formal Verification of an energy aware routing algorithm *21st Nordic Workshop on Programming Theory (NWPT09)*, 2009

Acknowledgements

I would like to begin by thaniking my supervisors associate professor Michael Reichhardt Hansen and professor Jan Madsen for their professional supervision. They have provided me with more points of view on my research which broadened my research and results. I would also like to thank associate professor Ingo Sander for his invaluable help on visiting the Royal Institute of Technology (KTH) as part of my research.

I thank all my colleges at the embedded systems engineering section at DTU for their moral support and creating an inspiring workplace.

I would also like to thank the SYSMODEL project (ARTEMIS JU 100035) and the IDEA4CPS project (granted by the Danish Research Foundation for Basic Research) for partially funding my Ph.D. study.

Finally, I would like to thank my family and especially my wife and son for their understanding and support during my last and intensive monts of writing this thesis.

Glossary

ADC analogue to digital converter. 76, 97

AMS analogue and mixed signal. 37–39

CAD computer aided design. i, 2, 139

CPS cyper physical system. i, 1, 139

CT continuous time. i, 4, 31, 39, 41, 42, 45, 49, 51, 70, 75–77, 90–92, 94–97, 140

CTL computation tree logic. 59

DAC digital to analogue converter. 76, 97

DAE differential algebraic equation. 38

DD directed diffusion. 11, 19, 21–25, 27–29, 114, 125, 131–138

DE discrete event. i, 4, 31, 41, 42, 45, 51, 70, 75, 77, 85, 89–92, 94, 95, 97, 140, 142

DEHAR distributed energy harvesting aware routing. i, 19, 21–28, 53, 54, 56, 62, 63, 114, 125, 128, 129, 131–138, 140

FFT fast Fourier transformation. 104, 105

ForSyDe formal system design. i, 4, 31, 42–47, 49–51, 65, 68, 70–72, 75, 77–82, 84, 89, 97–99, 101, 102, 109, 111, 139–142

MoC model of computation. i, 2–4, 31, 33, 34, 39–42, 44–47, 49, 51, 65–68, 70, 72–79, 82–87, 90–98, 104, 106, 111, 140–142

REM real ear measurement. 102

SDF synchronous data flow. i, 2, 4, 31, 34, 36–38, 41, 42, 45, 47, 51, 70, 75, 77, 82–84, 87, 92–94, 97, 104, 106, 107, 140

SFF system functionality framework. 100, 101, 106

SME small and medium enterprise. 99, 100

SoC system-on-chip. 37

SY synchronous. i, 31, 45–47, 51, 68, 70, 74–79, 82–85, 87, 92–95, 97, 140, 142

SYSMODEL system level modelling environment for SMEs. ix, 31, 47, 51, 77, 100, 141

T-SDF timed synchronous data flow. 38, 39

WSN wireless sensor network. i, 2–9, 12, 31, 51, 53–55, 60, 63, 65–67, 111–115, 139–141

Contents

Chapter 1

Introduction

CPSs are systems which integrate computational and physical elements. They are used in many places and products today. Many of these CPSs are also in the category of embedded systems. Designing CPSs is often not an easy task.

A CPS often has relatively strict non-functional requirements. Amongst others it can be a need for: reliability, safety, security, real time, power consumption, communication bandwidth, etc. The design space of a system may be more or less constrained by the individual non-functional requirements.

Seemingly simple systems in the design phase can quickly become complex systems when implemented. A simple functionality of the system can be made complex by e.g. real time requirements and energy limitations. As an example; a platform with a single core processor might not be able to deliver the computational power within the power budget, forcing the design of a multi core system.

An example of a simple system with well known and limited interaction with other systems is a hearing aid calibration device. This example will be used to demonstrate how a simple functionality is complex to implement due to conflicting non-functional requirements of low power and real time response. A change of the design process to a formal modelling approach has the potential to: increase the change of success, shorten the time to market, reduce price, etc. for companies. Such a system is categorised as a *static* system because of its simple, self-contained functionality. A static system can change its state over time, but the number of possible states is constant.

The design task only becomes more difficult when multiple systems depend on each other to perform a task. An example of a network of systems is a system

which consists of a varying number of sub-systems (in this case the sub-systems are static systems). Adding a sub-system will increase the state-space of the network of systems. Hence such networks of systems are categorised as *dynamic* systems.

The sub-systems of dynamic systems can for example be specialised static systems responsible for a subset of the functionality. Each of such static systems are often autonomous and can work in many different networks, which makes up similar dynamic systems. The static systems may be specialised to e.g. monitor certain parameters of the environment. Such parameters could: monitor structural integrity of buildings, monitor the indoor environment (light, heating, etc.), detect forest fires, and many other things.

An example of a network of systems is a WSN. WSNs are autonomous networks of cheap, self-sustained nodes that can collect information about the environment (i.e. the physical world). The common features of WSNs are; they must be able to collect, process, and communicate information. Often it is also required that they are cheap and service free in their lifetime (due to e.g. inaccessibility or shear numbers). The service free requirement implies that batteries cannot be replaced, size, price, environmental, and other constrains may also limit the energy capacity of batteries. Therefore it can be advantageous for nodes to harvest their own energy from the environment.

Data may need to travel through several nodes in a WSN in order to reach its destination. This requires a stable network, which implies that the energy reserves of nodes should not be depleted by such communication. Therefore intelligent routing data is necessary. Here intelligent routing represents a balance between the power consumed by the intelligent and the amount and distribution of power harvested and stored.

To make CAD choices possible, the design must be modelled in such a way that a computer can reason (calculate) about the model. Formal modelling can provide such means. Formal modelling is based on MoCs.

A MoC is a formal language definition for describing computation. Some MoCs also support various forms of formal analysis of models described herein. Another important parameter of a MoC is how expressive it is, i.e. the variety of models that can be described with it. MoCs that support formal analysis tend to be less expressive, as is the case for e.g. the SDF MoC. It can only support streaming models and no explicit timing. A model expressed in only one MoC is called a *homogeneous* model.

Heterogeneous models can make it possible to combine different MoCs that provide the best expressiveness for different parts of the system that is modelled. Thus the best expressiveness and analysability is obtained for each part of the

system, and the entire system can be simulated. To formalise the heterogeneous models, domains and domain interfaces must be defined. A domain can contain exactly one MoC and a domain interface provide a formal translation of communication between two domains. It is worth to note that two domains are not restricted to contain different MoCs.

1.1 The goal of the thesis

The goal of this thesis is to build a framework that supports modelling and analysis of dynamic systems. A use case of a WSN is used to illustrate modelling and analysis of a dynamic system. In this use case the WSN is a multi-hop mesh network of nodes. The nodes are to measure the environment and relay the relevant data to a base station. For various reasons (e.g. size, price, and zero maintenance) the nodes are required to harvest the energy needed for operation from the environment. The nodes also have a limited storage for energy.

In such a WSN it is necessary to know when and where to consume power. Each node can only do little on its own to change its power consumption if data transmissions should not be lost. On the other hand, if the nodes cooperate, they may be able to route data to the base station such that energy consumption is distributed to nodes that have enough energy.

Such a routing algorithm can be validated by simulation. But validation does not guarantee correct functionality, it can only make it likely to be correct. A verification of correctness can be able to guarantee the functionality. However, verification of functionality is a challenge for dynamic systems under the constraints of the non-functional requirements.

To make it easier to perform analysis of dynamic systems, a dynamic system is divided into static systems. Each unique static system is then analysed separately. In the case of the WSN there are two static systems, the base station and the nodes. These static systems are, however, highly dependent on communication with the network to define their operation.

To illustrate how a static system can be modelled and analysed a use case of a hearing aid calibration device is presented. This use case has a self contained behaviour, as opposed to the nodes of a WSN where their behaviour depends greatly on network interaction. In order to analyse a static system (or parts of it), it must be modelled in a MoC which supports the required type of analysis.

1.2 Contributions of the thesis

Contributions of this thesis includes the following. A dynamic energy harvesting aware routing algorithm for multi-hop WSN. Integration of three of the four MoCs SDF, DE, and CT in the Haskell [2] version of the framework ForSyDe. Optimisations of the DE MoC. Illustrating how to enable early design analysis of systems. Generic framework for modelling energy harvesting aware WSN.

1.3 Structure of the thesis

Chapter 2 Energy harvesting wireless sensor network. This chapter introduces WSN, energy harvesting, and routing in multi-hop networks. A routing algorithm for a multi-hop WSN is designed to take advantage of energy harvesting. The algorithm is validated by simulation.

Chapter 3 Related frameworks. This chapter describes related modelling frameworks.

Chapter 4 Analysis in UPPAAL. Presents the WSN use case modelled in the UPPAAL framework. Describes formal verification of the routing algorithm for multi-hop WSN.

Chapter 5 Theory of systems modelling. This chapter describes the theory of the ForSyDe framework, the individual MoCs and modelling techniques of homogeneous and heterogeneous systems.

Chapter 6 Static systems. Through the description of a company use case, modelling and analysis of static systems in ForSyDe is shown. The division of the system into application model and platform model and the integration of these two models is highlighted. The seemingly simple system of the use case did present challenges for the company that could have been avoided with early design models.

Chapter 7 Dynamic systems. Revisit the WSN use case and present a framework to better express the dynamic behaviour of the system.

Chapter 8 Perspectives and conclusion.

Chapter 2

Energy harvesting wireless sensor network

One of the key design goals in wireless sensor networks (WSNs) is long lasting or even continuous operation. Continuous operation is made possible through energy harvesting. Keeping the network operational imposes a demand to prevent network segmentation and power loss in nodes. It is therefore important that the best energy-wise route is found for each data transfer from a source node to the sink node. A new adaptive and distributed routing algorithm for finding energy optimised routes in a wireless sensor network with energy harvesting is presented in this chapter. The algorithm finds an energy efficient route from each source node to a single sink node, taking into account the current energy status of the network. By simulation, the algorithm is shown to be able to adapt to changes in harvested and stored energy. Simulations show that continuous operation is possible.

2.1 Introduction

Energy efficiency is a major concern in WSNs. As sensor nodes are typically battery powered, the energy usages has to be carefully managed in order to prolong the lifetime of the system. A sensor node in a WSN has two major functions 1) to collect and produce data from its physical environment and 2) to route data from it self and neighbouring nodes towards a base station which collects all data produced by the WSN for further processing. We assume an

ad-hoc, multi-hop network which is the common approach for large network deployments, where we cannot afford the energy required to transmit data directly from the node to the base station. In this chapter we are interested in energy-aware routing in such networks.

Energy efficient and energy-aware routing algorithms have been extensively studied. A common characteristic of most of these is the assumption of a battery which is gradually drained. Hence, the challenge is to find statistical or dynamic routing strategies which can assure the longest lifetime of the battery in any node of the network. By applying low-power hardware and software techniques for the design of the nodes, we can lower the rate at which the battery is depleted, and by reducing the duty-cycle, i.e., the time intervals at which the node is active, we can stretch the lifetime of the battery. As nodes close to the base station will be involved in more routing than those far away, a straightforward routing approach will quickly drain the battery of these nodes, effectively cutting of the rest of the network from the base station. Hence, energy-aware routing algorithms need to take the energy level of the nodes into account, i.e., finding energy optimised routes, where nodes with too little energy are avoided.

In order to further improve the lifetime and performance of WSN, there has been an increasing interest in energy harvesting, i.e., having each node to harvest energy from the environment. The environmental energy is a continuous and sustainable supply which, if appropriately used, can provide WSNs to last forever. Although attractive, energy harvesting is a very unreliable energy source which makes it challenging to use. A major challenge is to find where (and when) there is available energy to be harvested, and this should be done in an energy efficient manner. We suggest to supply the battery of the node with a solar panel as the energy harvester. This will allow each node to regain energy (i.e., charge the battery) while the node is inactive, and to use "free" energy when it is active.

In this chapter, we present an adaptive routing algorithm which is able to find and maintain energy optimised routes from any source node to a base station (called the sink or destination node in the following). By energy optimised routes we mean routes that avoid nodes with too little energy, effectively allowing these nodes to regain their energy level through energy harvesting. The proposed algorithm is adaptable and distributed, i.e., each node runs autonomously, taking routing decisions based solely on available energy on its neighbouring nodes. As each node makes local routing decisions, a route may change while the data is being routed. To assure a net energy gain, it is important to also account for the energy used by the routing algorithm itself.

In our setup we simulate the uncertainties of energy harvesting through

global parameters, such as time of day, and local parameters, such as amount of shadow for a given node position. This emulation of the environment is observed by the proposed routing algorithm and used to direct network traffic such that nodes in areas with lower energy are kept alive.

The chapter is organised as follows:

Section 2.2 The related work is presented.

Section 2.3 Presents the WSN and energy models used to develop and simulate the proposed routing algorithm.

Section 2.4 Describes the algorithm in detail.

Section 2.5 The setup and results of simulations are presented and discussed.

Section 2.6 Finally the conclusion is presented.

2.2 Related work

Many different kinds of energy aware algorithms exist today. They can be divided into three classes: energy efficient, residual energy aware, and energy harvesting aware algorithms.

Energy efficient algorithms [16, 22, 71] aim at increasing the lifetime of the network as a whole without measuring residual energy in the battery. They will for example distribute the routed packages to several neighbours to minimise the energy consumption of the nodes on the shortest path.

The energy aware algorithms [22, 24, 45, 51, 66, 67, 74, 76] are measuring the residual battery energy and are extending the energy efficient algorithms to take into account the actual available energy in the routing. These algorithms make the assumption that the residual battery energy is monotonically decreasing, and can therefore not accommodate for energy harvesting.

The energy harvesting aware algorithms [28, 38, 44, 72, 73, 75] do not make the assumption of monotonically decreasing residual battery energy. Furthermore, they may estimate the future harvested energy to improve performance.

Surveys of routing algorithms [10, 11, 50] do not yet cover energy harvesting routing algorithms. Existing techniques for managing harvested energy are mostly dealing with energy management at the node level [9, 18, 34, 35, 55, 68]. Typically these techniques can not make network wide decisions and the energy use of routing is not managed.

Harvested energy can be managed by local techniques [9, 18, 34, 35, 55, 68] (e.g. scheduling), these techniques can, however, typically not make network wide decisions and the impact of routing is not managed.

Most research into energy harvesting aware routing algorithms seems to be in clustering algorithms, these are however not in the same family as multi-hop algorithms.

An energy harvesting aware multi-hop routing algorithm is the REAR algorithm [24]. It is based on finding two routes from a source to a sink, a primary and a backup route. The primary route reserves an amount of energy in each node along the path and the backup route is selected to be as disjunct from the primary route as possible. The backup route does not reserve energy along its path. If the primary route is broken (e.g. due to power loss at some node) the backup route is used until a new primary and backup route has been build from scratch by the algorithm.

Another algorithm uses measurements of harvested energy more directly in its routing [73]. It does this in a simple way, by detecting whether a node is harvesting or not. It does not take stored energy or the amount of harvested energy into account.

A mathematical framework for energy aware routing for multi-hop WSNs, which can cope with renewable energy sources routing, is established in [44]. An algorithm based on this framework is presented, which is shown to be asymptotically optimal. The advantage of this framework is that WSNs can be analysed analytically, but the algorithm relies on a rather ideal assumption that changes in nodal energy levels are broadcasted instantaneously to all other nodes. This is, in many applications, neither realistic due to limitations in radios' ranges nor desirable as it would cause a large overhead and use of energy. In our work we present a distributed solution where changes of energy levels are communicated to neighbour nodes only.

An approach which is more related to ours is [75], where geographical routing is considered in connection with energy harvesting. In this work global geographical information, such as information about the position of the destination and the node, is combined with local node information, such as energy information of neighbour nodes, in order to find an energy efficient route to the destination. We are not exploiting geographical information. Instead we capture the global information in a so-called *energy-faithful distance* for every node. This energy-faithful distance for a node approximates the cost (energy-wise) of routing from that node to the destination. This adjustment is dynamically recomputed on the basis of changed energy levels in neighbour nodes, and simulations have shown that these computations can be performed efficiently.

2.3 Wireless sensor network model

The model of the WSN must capture the energy consumption and distribution in the network. The model of a node (i.e. modelling processing and communication) is related to energy consumption and storage and the environment model is related to the energy production.

2.3.1 Environment

The environmental model describes sensor input and the energy source for harvesting. The important features of the sensors are measurement capability and rate. The energy harvesting model describes the energy production for every node.

The energy model of the environment consists of two parts, a uniform energy source and a non-uniform attenuation. The uniform energy source $P_I(t)$ is the same for all nodes. The attenuation $f_{S,N}(t)$ of node N models obstacles in harvesting capability. Such obstacles could be clouds, trees, etc when considering solar energy harvesting.

The power production $P_{S,N}(t)$ of the energy source S in node N is modelled as:

$$P_{S,N}(t) = P_I(t) f_{S,N}(t) \tag{2.1}$$

We have experimented with functions $P_I(t)$ generated from concrete streams of real-life observations of insolation of a solar panel and with ideal values of insolation which models a full day with clear sky.

2.3.2 Network

The network consists of an arbitrary number of nodes, with one sink. The nodes can be identical in both hardware and software, or configured individually. All nodes produce measurements of the environment at a specified rate.

The nodes are placed in a 2D plane where the unit length is equal to the shortest radio range possible for the nodes. The nodes can be placed freely in the 2D plane. The radio range of nodes can be varied freely and is specified in the same unit length as the resolution. The nodes have perfectly circular radio coverage.

To facilitate the energy investigation, some energy consuming tasks must be deployed on the nodes. To this purpose an application outlined in Figure 2.1 is deployed in the network. It is triggered regularly by an interrupt ($\star$). The interrupt can be either periodic or stochastic.

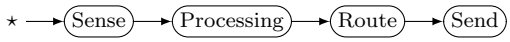


Figure 2.1: An outline of the application running on the nodes. It is initiated by an interrupt ($\star$). It is measuring some data, processing it, compiling a package and routing it towards sink.

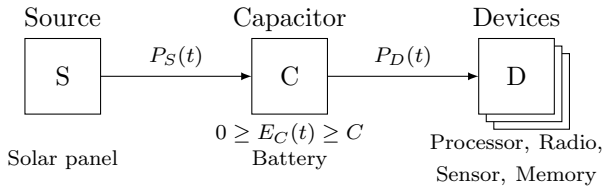


Figure 2.2: Overview of the energy model. To the left is an energy harvesting device (e.g. a solar panel) recharging the battery/capacitor in the middle. The devices to the right are consuming energy from the battery/capacitor.

The application takes some measurements from the environment, processes it and finally lets the routing algorithm find the best neighbour for routing and sends the package. The different parts of Figure 2.1 have different energy profiles, e.g. the radio used for sending consumes most energy.

2.3.3 Energy model

For each node in the network, the energy stored and power produced and consumed are recorded. All three parts can be configured to match a given platform. Only one source and capacitor is present in each node, while several consuming devices are present as shown on Figure 2.2.

The devices consists of a processor, a radio, a memory and a sensor. The tasks running on the node activate the devices when needed.

There are different kinds of energy storages, such as an ideal super capacitor (large lossless capacitor) or a battery. There is an upper bound C of the capacity of the energy storage, i.e.

$$0 \leq E_C(t) \leq C \tag{2.2}$$

where C is the capacity and $E_C(t)$ is the energy stored at time t. For the ideal

super capacitor the power model is

$$E_S(t_1, t_2) = \int_{t_1}^{t_2} P_S(t)\delta t \tag{2.3}$$

$$E_D(t_1, t_2) = \int_{t_1}^{t_2} P_D(t)\delta t \tag{2.4}$$

$$E_C(t_2) \leq E_C(t_1) + E_S(t_1, t_2) - E_D(t_1, t_2) \tag{2.5}$$

for $t_1 < t_2$, where $P_S(t)$ is the power harvested and $P_D(t)$ the power consumed at time t.

The power harvested $P_S(t)$ depends on the insolation, the shadow and the size and efficiency of the solar panel. All these parameters are customisable. Likewise $P_D(t)$ depends on the configuration of the devices and which state the devices are currently in.

2.4 Energy harvesting aware routing algorithm

This algorithm aims at dynamically finding sustainable routes in a multi-hop wireless sensor network with energy harvesting. A route is sustainable if the energy of nodes along the route is not exhausted. It is assumed that the stored energy in a node is measurable and through the changes of stored energy it is possible to calculate the consumption and production of energy.

To calculate a sustainable route to sink, information about both the available energy and the shortest distance from any node to sink is needed. The routing algorithm has two parts where one finds all shortest paths/distances to sink and the other applies distance penalties on paths to compensate for lower energy availability.

2.4.1 Shortest path

The calculation of the shortest path can be performed with several different existing algorithms, such as directed diffusion (DD) [27] and distance vector routing in general. Such an algorithm manages the structural information of the network. Furthermore it takes care of nodes being introduced into or removed from the network. The *simple distance* d_s is defined as the distance of the shortest path.

An example network is shown on Figure 2.3, where the network structure is shown in (b) and the simple distance is shown in (a).

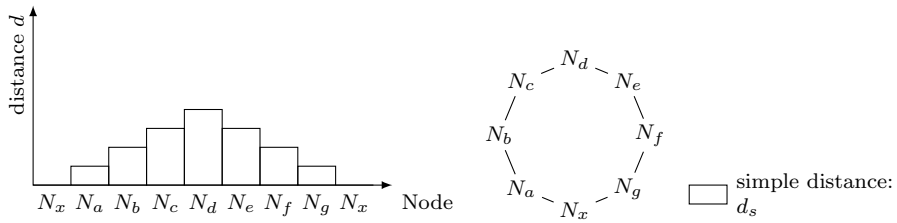


Figure 2.3: An example network displaying the shortest distance to sink (N_x). (a) graph shows each node's distance to sink while (b) shows the placement of each node.

2.4.2 Energy information encoding

The shortest path to sink is implicitly also the least energy consuming path to sink. This path does, however, not consider how much energy is available along this path or any other path.

To add energy awareness the available energy $e : E$ in a node must be measured, where $E \triangleq [0; 1]$ (normalised with respect to the energy storage capacity C). Then it is converted into a distance with the function $f : E \to D$ where $D \triangleq \mathbb{R}_{\geq 0}$. This distance reflects the energy deficit (the capacity of the energy storage minus the energy e) of the node. The distance is used as a distance penalty (d_p) to route through the node.

To be meaningful, f should be monotonically decreasing. The ideal situation is that f approaches zero when there is plenty of energy, i.e. $f(e) \to 0$ when $e \to 1$ and f approaches infinity when there is lack of energy, i.e. $f(e) \to \infty$, when $e \to 0$. In a concrete WSN these ideal situations must be approximated.

The example network is shown in Figure 2.4 where some nodes have an energy deficit (b) and how it is translated into a distance (a) for those nodes.

An example of a function for transforming the measured energy availability $e : E$ to the distance penalty $f_p(e) : D$ is shown in (2.6). This is further exemplified with Figure 2.5.

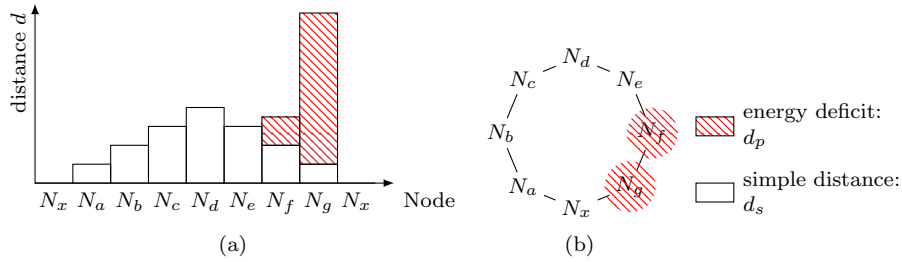
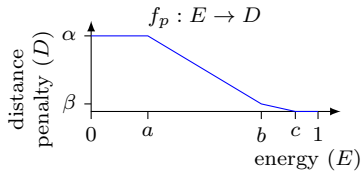


Figure 2.4: The example network in Figure 2.3 now displays an area with low energy availability (shaded area) in the layout (b). This results in local distance penalties in the graph (a).



$a = 0.25$
$b = 0.75$
$c = 0.90$
$\alpha = 50$
$\beta = 5$

Figure 2.5: Example of relation between energy availability and local distance penalty.

$$f_p(e) = \begin{cases} 0 & ,1 \geq e > c \\ \beta \frac{e-c}{b-c} & ,c \geq e > b \\ (\alpha - \beta)\frac{e-b}{a-b} + \beta & ,b \geq e \geq a \\ \alpha & ,a > e \geq 0 \end{cases} \tag{2.6}$$

The values a, b and c are different thresholds of energy availability. c determines the upper bound for sensitivity. a is the lower bound for energy availability. b describes the point of change between different sensitivities of variations in energy availability together with the penalty amplitude β. α describes the maximum penalty.

A limited energy availability is now applied to the example network (the shaded area) in Figure 2.4. This results in local distance penalties to the nodes N_g and N_f. In this particular example there is a local minimum distance to sink at node N_e. This is an undesirable situation as N_e has no neighbour to

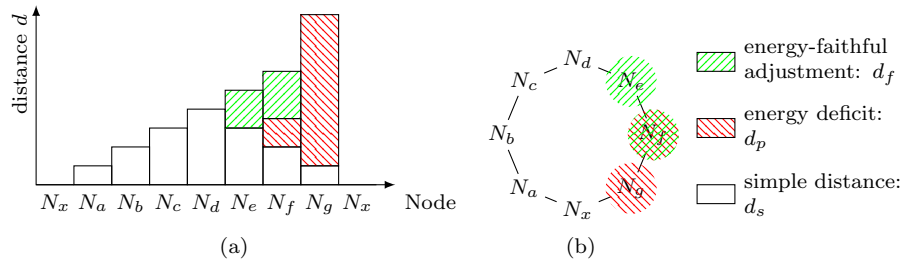


Figure 2.6: The example network from Figure 2.3 and Figure 2.4 is now completed with the energy-faithful distances. Now all nodes have a sustainable route to sink.

whom it would be (on the basis of the simple distance and distance penalty) natural to send packages to.

The *energy-faithful adjustment* (d_f) is designed to solve the problem with local minima which the distance penalties can create. Every node communicates changes in their energy information to its immediate neighbours. Each time a node receives an update from a neighbour, it checks if it is in a local minimum. If so, it increase its energy-faithful adjustment to solve the problem and reports this to its neighbours through an update.

This process also works the other way around, where a node checks whether its energy-faithful adjustment is unreasonable high and lowers it appropriately. In the example network, the energy-faithful adjustment is now added (see Figure 2.6). For convenience, the sum of the distance penalty (d_p) and the energy-faithful adjustment (d_f) is called the abstract distance (d_a). We shall now present an algorithm which is based on these ideas.

2.4.3 Algorithm

Each node runs the same set of algorithms to manage the distance penalties and the energy-faithful distances. These algorithms manage a set of variables which constitutes the state of a node. It also manages a copy of the state of all neighbouring nodes. Furthermore it is assumed that each node uses the same function $f_p : E \to D$ to calculate the distance penalty from the energy stored in the node.

Consider a node N (see Figure 2.7) having k_n neighbours $\mathcal{N} = \{N_1, N_2, \ldots, N_{k_n}\}$.

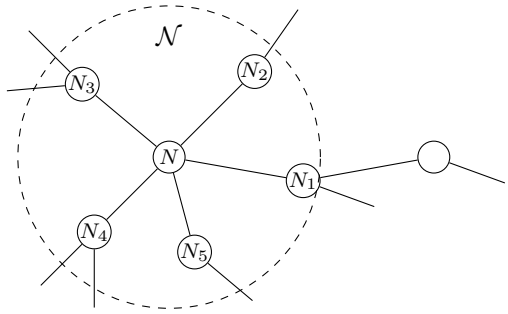


Figure 2.7: An example of a node and its neighbourhood. The node N has a set of neighbours $\mathcal{N} = \{N_1, N_2, N_3, N_4, N_5\}$. The dashed circle shows the neighbourhood area of N.

The node N has four state variables

$d_s : D$	Simple distance.
$d_p : D$	Distance penalty.
$d_f : D$	Energy-faithful adjustment.
$r : \{1, \ldots, k_n\}$	Index of neighbour to route to.

Furthermore, for each neighbour N_i of N, there are two state variables:

$d_{s_i} : D$	The simple distance of N_i.
$d_{a_i} : D$	The abstract distance of N_i.

Based on that information the neighbour N_r, with the shortest energy-aware distance to sink, is found.

Note that the distance penalties and energy-faithful distances are merged into one abstract distance before it is distributed, hence only the abstract distance of a neighbour is known to a node. The states d_s and d_{s_i} are managed by an algorithm for finding the shortest path and is therefore not changed by this algorithm.

The algorithm is divided into three parts, a main part A_1, and two sub parts A_2 (distance penalty and route update), and A_3 (energy-faithful distance update and broadcast) described in the next three paragraphs. The main A_1 is displayed in Algorithm 1. It reacts on an input event x. This event can either be an abstract distance update coming from one of the neighbours of N or a local energy update.

If the event is an energy update (line 1), the measured energy is converted to a distance penalty through f_p and passed to A_3. If the event is an abstract distance update (line 3), the event is passed to A_2 and A_3 is called with the current distance penalty.

Algorithm 1 The algorithm A_1 performs the state update of the routing algorithm.

Require: x
1: **if** x is a local energy update **then**
2: $A_3(f_p(x))$ {Energy availability, distance penalty update and broadcast}
3: **else if** x is an abstract distance update from a neighbour **then**
4: $A_2(x)$ {Penalty (from neighbour) and route update}
5: $A_3(d_p)$ {energy-faithful distance update and broadcast}
6: **end if**

A_2 / Distance penalty and route update

This algorithm is activated when an update p is received from a neighbour. It is carrying the id $p_{\text{id}} \in \{1, \ldots, k_n\}$ of the neighbour and the new abstract distance $p_{\text{a}} : D$. It determines the shortest energy-aware distance to sink from the known state of the neighbours and updates the route index r if necessary (see Algorithm 2). This algorithm has no output but updates the state r and the penalties d_{a_i} of the concerned neighbour.

If the update originates from the node that packages are currently routed to (N_r) and this has increased its distance to sink, then a search of the neighbour table must be performed to find the neighbour with the shortest energy-aware distance to sink. If the update originates from any other node than N_r and node N_i has acquired a shorter energy-aware distance to sink than N_r, then N_i is now the neighbour with shortest energy-aware distance to sink. In any other case, the route r is unchanged.

A_3 / Calculate energy-faithful adjustment and broadcast

A_3 takes as argument an update to the distance penalty, calculates the energy-faithful adjustment, updates the states d_p and d_f and determines whether to apply and broadcast the update (see Algorithm 3).

Four constants are used: c_a, c_{min}, c_{lower}, and c_{raise}. $c_a \in \mathbb{R}_{>0}$ is the minimum difference in distance to sink between node N and the neighbour

Algorithm 2 The algorithm A_2 takes as input a package p containing an abstract distance update p_{a} and the id p_{id} of the neighbour. Based on the update, it updates the states r and d_{p_i}.

Require: $p : P$

1: $i \leftarrow p_{\mathrm{id}}$
2: $d_i \leftarrow d_{s_i} + p_{\mathrm{a}}$
3: $d_r \leftarrow d_{s_r} + d_{p_r}$
4: $d_{p_i} \leftarrow p_{\mathrm{a}}$
5: **if** $r = i \wedge d_i > d_r$ **then**
6: $d \leftarrow \infty$
7: **for all** $n \in \{1, \ldots, k_n\}$ **do** {Find neighbour with shortest energy-aware distance to sink}
8: **if** $d > d_{s_n} + d_{p_n}$ **then**
9: $r \leftarrow n$
10: $d \leftarrow d_{s_n} + d_{p_n}$
11: **end if**
12: **end for**
13: **else if** $r \neq i \wedge d_i < d_r$ **then**
14: $r \leftarrow i$
15: **end if**

chosen for routing N_r when altering the energy-faithful distance of node N. $c_{min} \in \mathbb{R}_{\geq 0}$ is the minimum difference between N and N_r before an update of energy-faithful distance of N is forced. $c_{lower} \in \mathbb{R}_{>0}$ and $c_{raise} \in \mathbb{R}_{>0}$ are thresholds for avoiding communication of small changes to distance penalties and/or energy-faithful distances.

The distance difference (Δd_1) between the neighbour with shortest energy-aware distance to sink (N_r) and it self (N) is found. If this difference is positive then the energy-faithful distance must be increased and may have to be lowered if the difference is negative. To determine this, a new energy-faithful distance d'_a is calculated. If the difference Δd_2 between the current and new energy-faithful distance exceeds the bounds $\Delta d_2 < c_{lower}$ or $\Delta d_2 > c_{raise}$, or if node N has shorter distance to sink than any neighbour, then an update must be performed. The lower and upper bounds of Δd_2 are to prevent insignificant updates to conserve energy.

Algorithm 3 The algorithm A_3 takes as input an update to the distance penalty d'_p. It calculates the energy-faithful distance and updates the states d_p and d_a. Any update to the states are also broadcasted to the neighbours.

Require: $d'_p : D$

1: $\Delta d_1 \leftarrow d_{s_r} + d_{p_r} - \left(d_s + d'_p + d_d\right)$ {energy distance diff. between N_r and N}
2: **if** $(d_a + c_a) > -\Delta d_1$ **then** {Ensure it always hold that $d_a \geq 0$}
3: $\quad d'_a \leftarrow d_a + c_a + \Delta d_1$
4: **else**
5: $\quad d'_a \leftarrow 0$
6: **end if**
7: $\Delta d_2 \leftarrow d'_p - d_p + d'_a - d_a$ {local distance to sink change}
8: **if** $-\Delta d_1 < c_{min} \vee \Delta d_2 < c_{lower} \vee \Delta d_2 > c_{raise}$ **then** {Determine whether to update state and broadcast the update}
9: $\quad d_p \leftarrow d'_p$
10: $\quad d_a \leftarrow d'_a$
11: $\quad$ broadcast $(d_p + d_a)$
12: **end if**

A change in the network structure

A change in network structure is defined as a change to d_s and consequently d_{s_i} in the neighbours of the node in question. Such a change can affect the energy-

faithful distance in the network, but not the distance penalties. Therefore the energy-faithful distance must be updated for all nodes that are affected by the structural change.

A search for the neighbour with the shortest energy-aware distance to sink will always work correctly. Depending on the algorithm used for finding the shortest path to sink, it might be possible to optimise this calculation like the approach in Algorithm 3. The cost of distributing the new energy-faithful distance is not large, since they will mostly follow the same pattern as the structural updates and they can be merged into one transmission carrying both distance penalty and simple distance update.

2.5 Results

A simulator based on the above models and algorithms has been constructed to investigate the behaviour of the proposed distributed energy harvesting aware routing (DEHAR) algorithm. With this simulator several of network setups are investigated and the most interesting are discussed in this section. Note that the grid deployment of the shown figures are solely for simplifying the presentation, the DEHAR algorithm is, however, not depending on grid deployment.

Apart from the proposed routing algorithm a simplified version of the DD algorithm has also been implemented. This simplified version of DD implements only the ability of finding the shortest path to sink for every node in the network. This is used both as the foundation of the proposed algorithm and to show the difference between DD and the DEHAR.

Two distinct network layouts have been chosen to be displayed. One that shows the algorithms ability to find large "detours" in order to find sustainable routes (see Figure 2.8) and another which shows that the algorithm is dynamic (see Figure 2.9). The first network layout is used in Simulation S_1 and S_2 while the second is used for Simulation S_3. Simulations of both networks will be using the function f_p in (2.6) for calculating the distance penalties.

The solar insolation pattern in the presented simulations is a 12 hour full daylight followed by 12 hour full night scenario. The total solar insolation per day is calculated from the daily average solar insolation of a full month from real sensor nodes with solar panels. The reason not to use the real solar insolation data directly is that it makes it hard to see the effects of the energy harvesting aware algorithm.

The three simulations are used to explore features of the DEHAR algorithm and differ slightly in setup, apart from the network layout (see Table 2.1). The

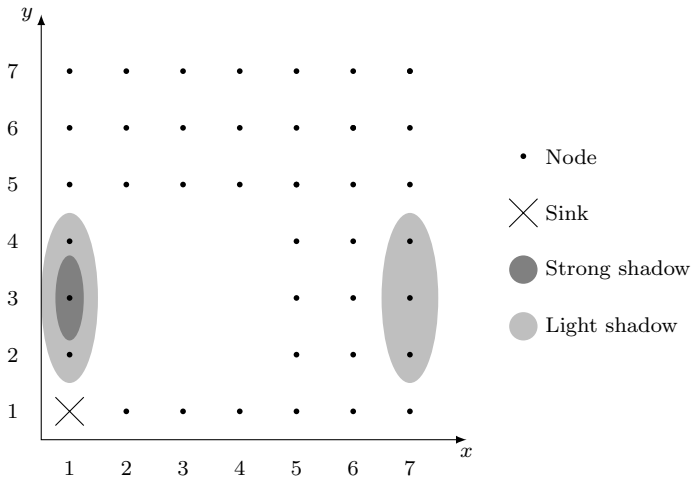


Figure 2.8: Network structure of Simulation S_1 and S_2. Radio range of nodes is set to 1.

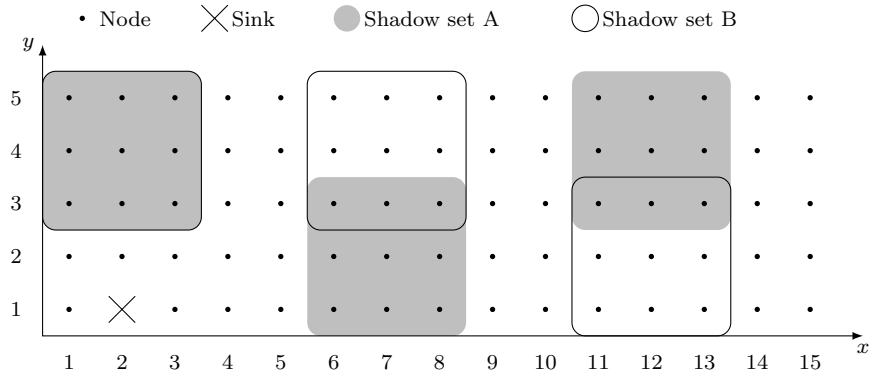


Figure 2.9: Network structure of Simulation S_3. Radio range of nodes is set to $\sqrt{2}$. The shadow pattern is switched every 360 hour (15 days).

Table 2.1: Differences between simulations. The average package rate is the rate at which each source node produces packages that are routed to sink.

	S_1	S_2	S_3
Radio range	1	1	$\sqrt{2}$
Average package rate	$\frac{1}{900}$ per sec.	$\frac{1}{60}$ per sec.	$\frac{1}{60}$ per sec.

change in package production rate is used to test the robustness of the algorithm towards changing energy consumptions.

2.5.1 Comparing DEHAR with DD

To evaluate the energy consumption of the DEHAR algorithm, it is compared against routing along the shortest route to sink using DD only. The DEHAR algorithm can be disabled in the simulator leaving the simulator in DD mode. Note that all penalties are initially zero, and the battery fully charged.

The results of the first 10 days (240 hours) of Simulation S_1 is shown in Figure 2.10. The first 108 hours of simulation show no difference as the energy aware algorithm has not yet detected any battery depletion. The power sensing part of the algorithm uses too little power to show on this scale.

To begin with DD uses less battery power than DEHAR, but only for a short time. This is due to the weak node positioned at the coordinates (1,3) (see Figure 2.8) using extra energy to inform the neighbours of its increased distance penalty and the extra energy used to route packages along an alternative path.

During the rest of the simulation the DEHAR continues to have an advantage during day and opposite during night, but overall having an advantage, since batteries contain more power. In this comparison the DD simulation will continue to deplete its weak nodes until they die. This is the case since the simulation is constructed to require re-routing of data (relative to DD) for all nodes to survive.

The quick rise of energy in both simulations is due to the outer nodes quickly recharging in the beginning of the day (far away from the sink). The nodes near the sink are charging more slowly or in some cases still discharging (on average) during day.

Results for Simulation S_1 for a total of 30 days (720 hours) are presented in the next section, i.e. the first 10 days (240 hours) are identical. After this, the two other simulations S_2 and S_3 are shown.

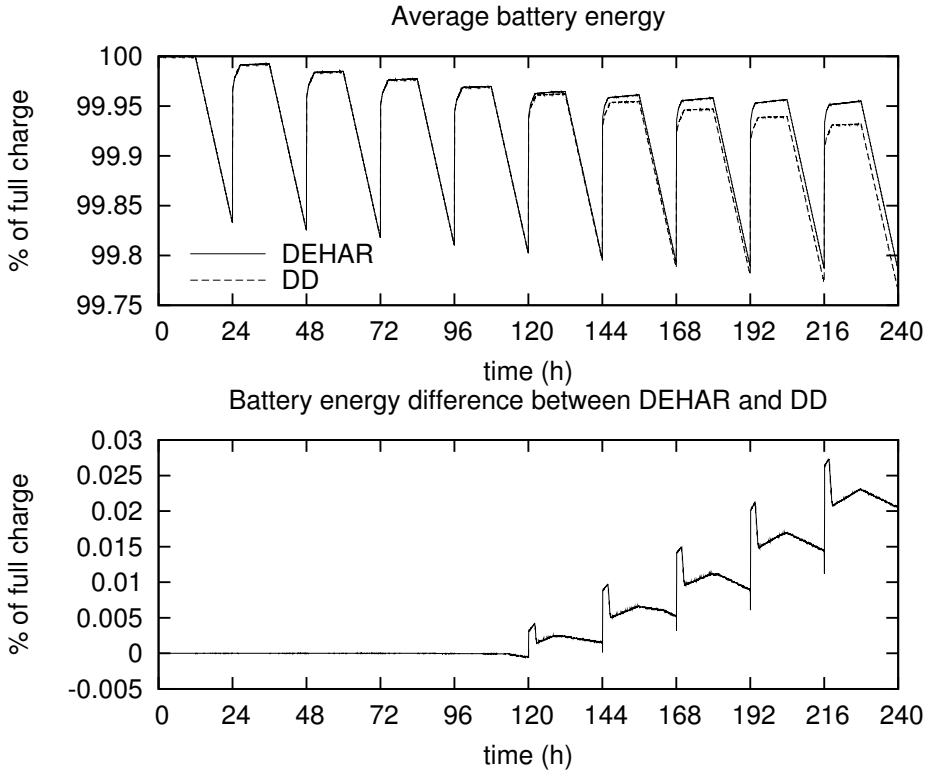


Figure 2.10: Simulation S_1 - 10 days (240 hours) simulation time. Starting with 12 hours daylight and 12 hours night. Data gathering interrupt every 15 minutes. Top figure shows the average energy of the individual simulation. Bottom figure shows the difference between the two simulations.

Low data rate simulation (S_1) The Simulation S_1 displays a routing trend as shown in Figure 2.11. This trend is constructed from the end of the simulation where the routes have settled. In Simulation S_1, during the first 30 days (720 hours), the DEHAR algorithm consumes 1.72% more energy than DD and is able to harvest 1.02% of the energy available (the energy not harvested is lost because the energy storage is full). The DD simulation show no difference and the weak nodes continue to deplete their batteries.

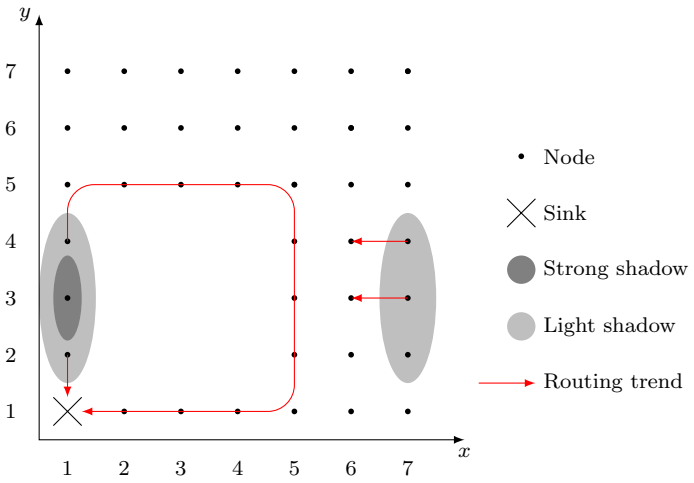


Figure 2.11: Routing trend of the DEHAR algorithm for both Simulation S_1 and S_2.

The simulation results are shown on Figure 2.10 and Figure 2.12. The variations in battery levels reflect the sum of power harvest, power consumption and battery capacity limit. The difference in power consumption of DEHAR and DD is shown on Figure 2.12 (top). It shows an identical power consumption until the routing algorithm kicks in after which the power consumption rises for the DEHAR. The bottom figure shows that the node with lowest battery energy is stabilising. The reason for the still fluctuating power consumption on the top figure is because some of the other nodes are still not stabilised completely. The DD continues to loose battery power on some nodes (see Figure 2.12).

The large fluctuations in energy consumption are mostly due to extra transmissions of data packages due to longer routes. The smaller fluctuations are partly due to status updates and the smoothing filter. The updates and data transmissions come in bursts when all the nodes are interrupted and does otherwise not use much energy.

High data rate simulation (S_2) Running the low data rate simulation again with data sensing interrupt changed to once every minute results in increased energy usage. Now the DEHAR results in a surplus energy us-

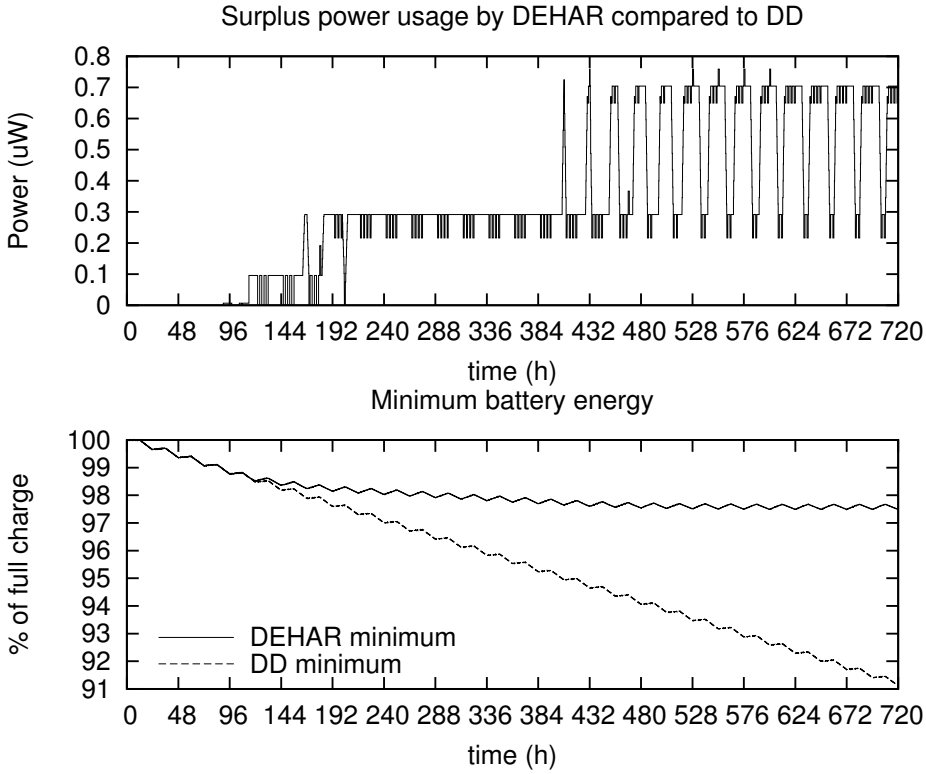


Figure 2.12: Simulation S_1 - Top: Plots of difference in energy consumption of DEHAR and DD. Bottom: Plot of lowest battery level in any node in both DEHAR and DD simulation.

age of 21.5% over DD and uses around 9.5% of the harvest-able energy (see Figure 2.13). The DD uses less than 8% of the harvest-able energy.

Comparing Figure 2.10 and Figure 2.13 it is seen that the DEHAR does not show much difference in average battery levels during day. During the night the batteries are depleted faster due to the increased energy usage where the DEHAR simulation have a slightly steeper descent in average battery level. This results in a faster convergence to the same routing pattern as for S_1. The DD protocol simulation will have node deaths earlier due to the increased energy

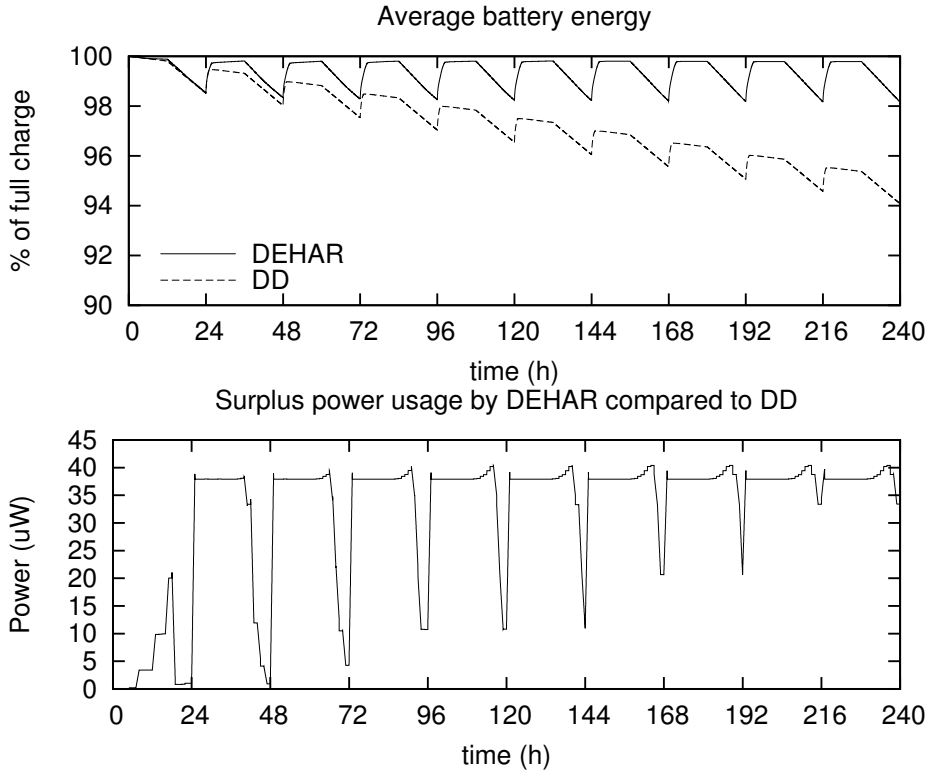


Figure 2.13: Simulation S_2 - Simulation results show an increased energy usage in both simulations. The DEHAR keeps a high average battery level.

consumption otherwise there are no notable differences between the two DD simulations.

Slalom simulation (S_3) The network layout for testing the dynamic aspects is shown on Figure 2.9. Simulation S_3 contains 45 nodes, some of which are under a shadow (e.g. have limited harvesting capability). The routing trend for the DEHAR is shown with arrows. The radio range has been set to $\sqrt{2}$ node distances so the nodes can reach up to eight neighbours. An interrupt frequency of once a minute has been used to initiate the data gathering and the simula-

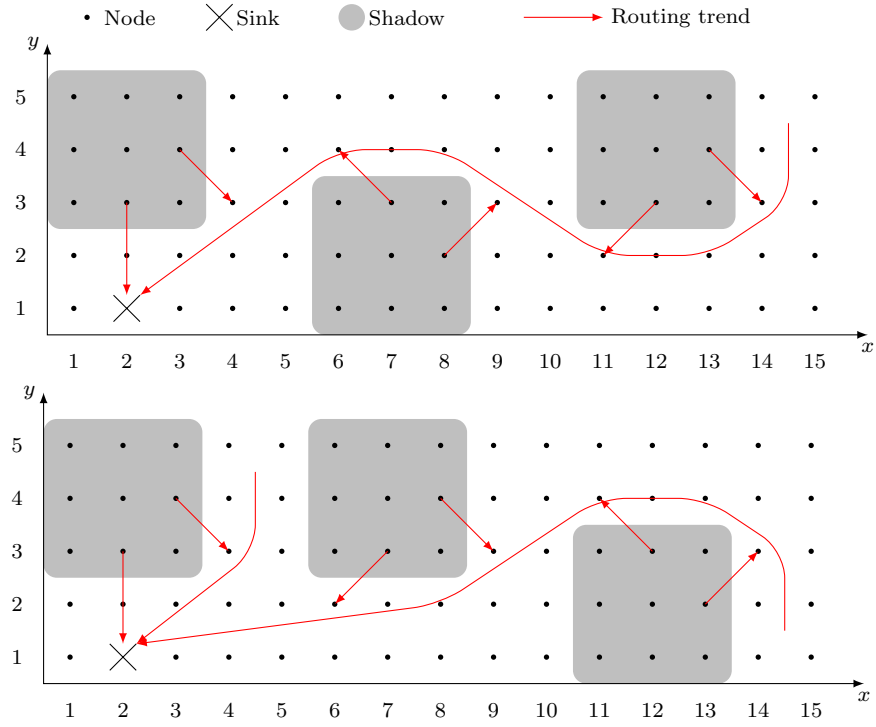


Figure 2.14: Routing trend of the DEHAR algorithm for Simulation S_3.

tion is run for 30 days (720 hours). After 15 days (360 hours) the shadows are moved to force the algorithm to re-route data again. The two trends are shown in Figure 2.14, where the upper layout shows the routing trend during the first 15 days and the other after the change of the shadow set.

The results of the simulation show that the data routes in the network are changed when the shadows move. When the shadows are moved it is clearly seen on Figure 2.15 that the batteries are recharged in those nodes that were previously under shadow. The extra energy usage by DEHAR is reduced. This can be due to the fewer hops needed to transport the data to sink.

As can be seen at the bottom picture of Figure 2.15 at least one node in DD has depleted its battery, thus it is not operational any more. This has happened

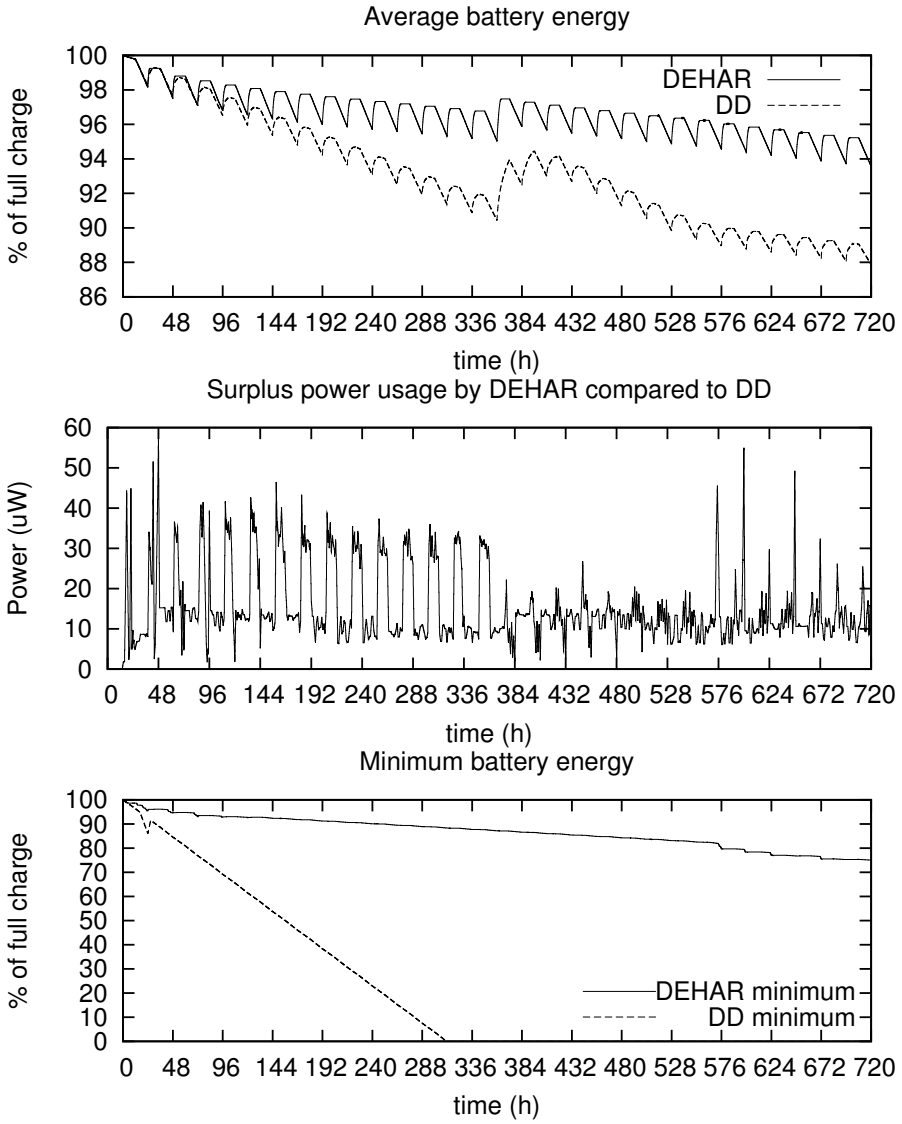


Figure 2.15: Simulation S_3 - Top: average battery levels. Middle: Power usage comparison. Bottom: Lowest battery energy levels for DEHAR and DD.

at the 13th day (approximately the 310th hour) of 30 days simulation. On the other hand with DEHAR no node has depleted its battery more than 70% during whole simulation period - 30 days.

In the slalom simulation the DEHAR uses 8.5% more energy than DD. The DEHAR uses around 13.4% of harvest-able energy while DD uses less than 12.3%.

Some nodes die in the DD simulation as seen on Figure 2.15 (bottom). At least one of those does not get powered up again when the shadows move.

2.6 Summary

The proposed algorithm is shown to find sustainable paths from any source to a sink. These paths are found dynamically by the distributed algorithm and without any node failing due to loss of power. This is in contrast to most energy (harvesting) aware algorithms which rely on finding the best known route and keep using it until some node fails along the path.

As long as the network continually has at least one sustainable path from each source to a sink, then the simulations show that the algorithm can find these paths – the algorithm can keep the network running indefinitely. This relies on nodes having sufficiently large energy storage and harvesting capability for the given environment and network.

Although not shown, the DEHAR algorithm is capable of handling multiple sinks. This does, however, require that the algorithm for finding shortest distance to sink is also capable of handling multiple sinks.

Though the results are encouraging, there are many possible improvements and extensions. Two of which are history dependent calculation of distance penalties and higher granularity of shortest distance to sink.

History dependent calculation aims at handling the drawbacks of the current distance penalty f_{pl} calculation, i.e. periodical (nights) or permanent low stored energy. A node which does not have full energy production is forced to first use some battery power before the distance penalty is high enough to force the use of alternative routes. It is desirable to let such nodes regain their energy after this has happened without increasing the amount of packages routed through these nodes. At night each node measures a consumption of battery power, and thus may increase the distance penalty. A history dependent calculation can let the stored energy increase without decreasing the distance penalty and thus obtain a higher amount of stored energy for nodes with low production. By estimating future energy production (introducing learning), energy usage

during nights can be handled more intelligently.

A history dependent calculation is, however, not without drawbacks. To maintain a energy history, the history must be stored in memory and more energy must be used to take the history into account when calculating the distance penalty.

The presented algorithm is using a distance measure based on the radio link (i.e. hops). This is basically due to the use of the DD algorithm, however, algorithms that provide finer granularity do exist and could substitute DD in order to obtain more accuracy in the distance measure.

Chapter 3

Related frameworks

Several frameworks exist, that support modelling of systems. The aim is to find frameworks that are able to express models of dynamic systems like the wireless sensor network (WSN) and the routing algorithm described in Chapter 2. The presented frameworks are compared and two frameworks are chosen. The two chosen frameworks will be used to model the WSN use case presented in Chapter 2. The SYSMODEL project, which funds the work of this thesis, requires ForSyDe to be one of these frameworks. The project partners of the SYSMODEL project require the four models of computation (MoCs) synchronous (SY), synchronous data flow (SDF), discrete event (DE), and continuous time (CT) and the ability to simulate models.

3.1 SystemC

C/C++ is probably the most widely used language in programming. There exist many commercial and high quality open-source compilers for it and a bulk of applications are also already implemented in C/C++. These have been the motivations for the EDA community to investigate C-based approaches for system design with the following advantages:

- to be familiar and easy to learn for the designers,
- do not need the exhaustive effort (and probably research) to implement new compilers,

- already existing algorithms and designs could be ported to the new language with smallest amount of re-engineering.

SystemC [6] is a language which takes such an approach. It is mainly a class library built on top of the C++ language which adds to the base language the features needed to model hardware—such as concurrency and a way to model time. While the syntactical aspects of the base language are mostly preserved, the semantics of SystemC is completely different [64].

The designer models systems in SystemC as a network of communicating processes. Computational processes are encapsulated in *modules*, while the communication among them is performed through *channels*. This clear separation between computation and communication has several advantages such as re-usability of the designed modules and allowing the designers to refine them independently. Figure 3.1 depicts the base components of a SystemC model.

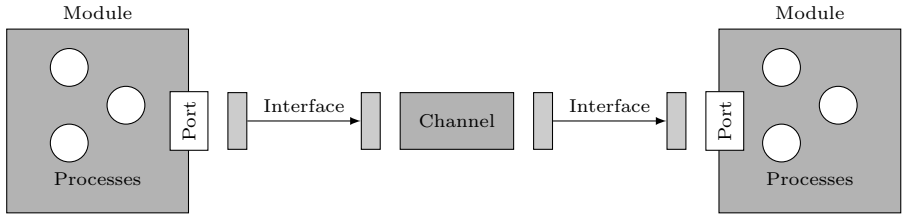


Figure 3.1: A SystemC model is composed of processes which are encapsulated in modules. Modules are connected to the outside world by means of their ports and communicate with each other via channels. Interfaces define the interaction with channels independent of channel implementation.

3.1.1 Transaction level modelling

Unlike what the name implies, TLM does not correspond to a specific level of abstraction, but to a set of abstraction levels above RTL. The precise definition of TLM has been for a long time a subject of debate. Cai and Gajski [17] have suggested 6 models in a two-dimensional abstraction space. The axis of their defined space are computation and communication abstraction levels. In addition, they have tried to describe how these models could fit in different design domains, namely modelling, validation, refinement, exploration, and synthesis. Donlin [19] has also defined a set of abstraction levels for TLM in a slightly dif-

ferent manner. He has tried to propose a set of use-models and describe which abstractions levels can be used in each use-model.

Rose et al. have proposed TLM SystemC as an extension in the form of a class library [61]. They have pointed out the main motivational points for enabling SystemC users with TLM as:

- providing an early platform for software development
- system level design exploration and verification
- the need to use system level models in block level verification.

The most important feature of the TLM library is a set of interface classes, each with their associated function calls to access channels (instead of directly dealing with pin level details). Blocking vs non-blocking, and bidirectional vs uni-directional access to the channels are the main semantic clarifying points for communication which are introduced by different interfaces. Recently, TLM 2.0 is introduced which promotes the previous version with the ability to model memory mapped buses which are register accurate and also provides more debugging facilities [54].

3.1.2 Heterogeneous SystemC

Today's electronic systems are becoming more and more complex. They include analogue and digital blocks, software and hardware in one place. The designer needs to capture the behaviour of such systems in different levels of abstraction using appropriate tools. MoCs [32] provide a framework by which a heterogeneous model could be captured. Unfortunately, SystemC, with its discrete-event simulation kernel, does not provide a direct way to model abstract MoCs. HetSC [25] tries to address this problem by introducing a set of rules and guidelines for modelling each MoC. In addition, it enables a smooth integration of different MoCs in the same specification. These are done by means of a methodology specific library on top of the standard SystemC. HetSC can work together with other extensions to SystemC such as SystemC-AMS [70], etc. Figure 3.2 shows the HetSC framework graphically.

HetSC claims that it makes heterogeneity in system design available in two directions: *vertical heterogeneity* and *horizontal heterogeneity* (see Figure 3.3). Horizontal heterogeneity enables the integration of several MoCs in a design at the same time. On the other hand, vertical heterogeneity is the ability of the modelling tool to support the evolution of each MoC during design refinement.

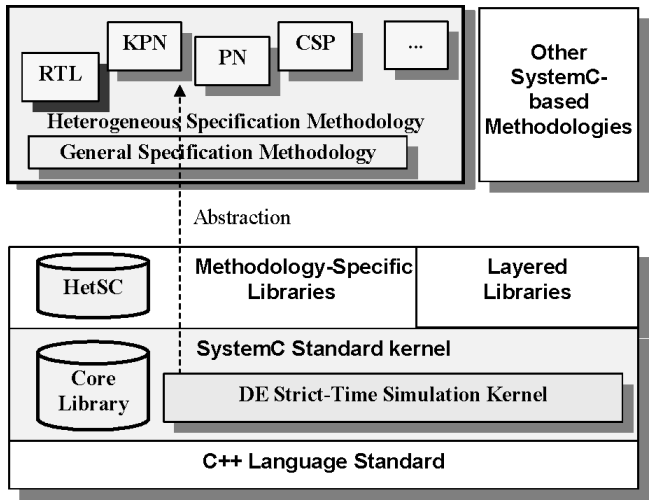


Figure 3.2: Framework for heterogeneous specification in HetSC [25].

At the highest level of abstraction—which is the specification level—a specific part of the system is modelled using a Kahn Process Network (KPN) MoC. During refinement, this part is first converted to a SDF model, from which a software code is generated later, and finally it is run on a DSP (vertical heterogeneity). Note that in the base specification model and also in the intermediate models obtained during each refinement step, the system is modelled using different MoCs (horizontal heterogeneity).

3.1.3 ARTS

ARTS is a SystemC based abstract system-level modelling and simulation framework [47–49], which allows the designer to model and analyse the different *layers*, i.e. application software, middle-ware and platform architecture, and their interaction prior to implementation. In particular, ARTS captures *cross-layer* properties, such as the impact of OS scheduling policies on memory and communication performance, or of communication topology and protocol [46] on deadline misses. Hence, ARTS can be used to explore and trade-off different design choices.

An ARTS *system model* (Figure 3.4c) of an embedded computing system is formed by *mapping* components of an ARTS *application model* onto computing

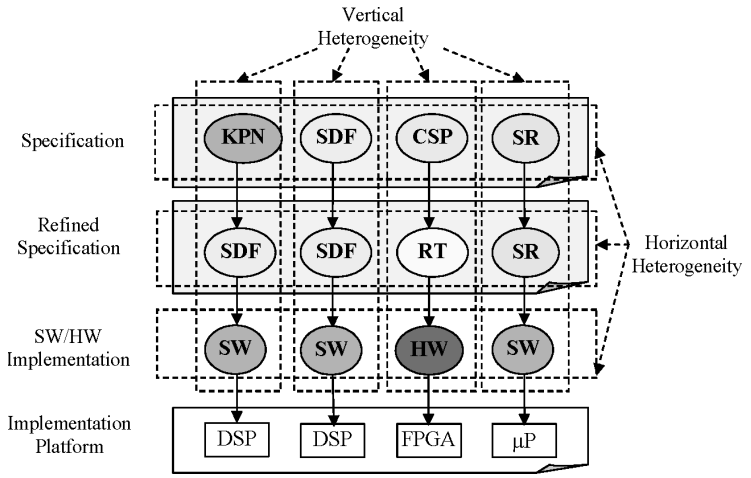


Figure 3.3: Horizontal and vertical heterogeneity [25].

components of an ARTS *platform model*. The application model is represented as a set of task graphs (Figure 3.4a), where each *task* represents a sub-element of the application and is considered as an atomic unit during mapping. The platform model is composed of computing components interconnected through communication components (Figure 3.4b), where each computing component takes care of executing the tasks mapped to it. Hence a computing component may be a programmable processor, a dedicated hardware accelerator or an operating system on a processor. All components interact through an event driven model.

The ARTS framework is implemented in SystemC which has several advantages. Besides being an industry standard for transaction-level modelling and system-level design, SystemC offers the possibility to co-simulate hardware and software, hence bridging the different layers, and to co-simulate systems at different levels of abstraction, i.e. simulating transaction-level components together with cycle-true components. This allows critical components to be refined to lower and more accurate levels of abstraction, while the rest of the system is still kept at an abstract level.

Figure 3.5 shows a design flow using the ARTS framework. A user has to provide the *application model* described as a set of task graphs (each representing a specific application), and the *platform model* which consists of platform com-

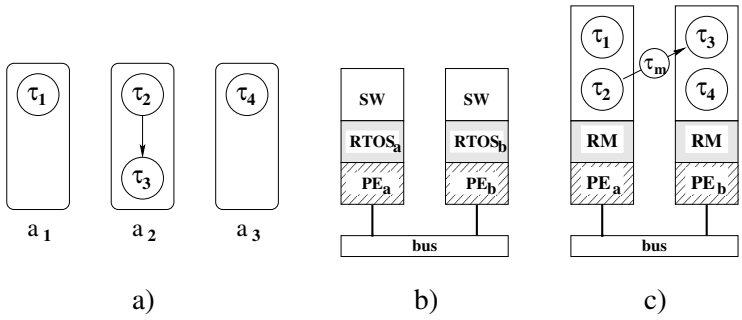


Figure 3.4: ARTS model example, a) application model, b) platform model, c) system model.

ponents, i.e. processing elements (PE) and communication networks, described in one file and a platform instance described in another file. These descriptions are expressed in a simple language called the ARTS scripting language. The user then has to provide a *mapping* of the application onto the platform, also in terms of an ARTS script. Prior to the mapping, a characterization of each task mapped onto each processing element has to be done. This characterization tells whether a task can execute on a processing element, and if so, how many cycles it will take to execute, how much memory is required, etc.. When loaded into the ARTS framework, a SystemC model of the complete system is created and simulated. The output from the simulation is a set of files providing runtime profiles and system characteristics, such as task execution profiles, bus contentions, memory and energy profiles, etc.. This allows the user to investigate the merits of the solution and to explore alternative solutions by applying different mappings or by changing the platform and/or the application.

3.1.4 SystemC kernel extensions

In the SDF [43] model of computation, a set of concurrent processes communicate via communication channels with unbounded FIFOs. Each process consumes data tokens from the input-side FIFOs, operates on them, and outputs the resulting data tokens to the output-side FIFOs. In each firing, the number of input/output data tokens for each process are fixed, and the scheduling of SDF models can be done at compile-time. Although SDF models can be modelled in SystemC using the existing modules and FIFO primitive channels,

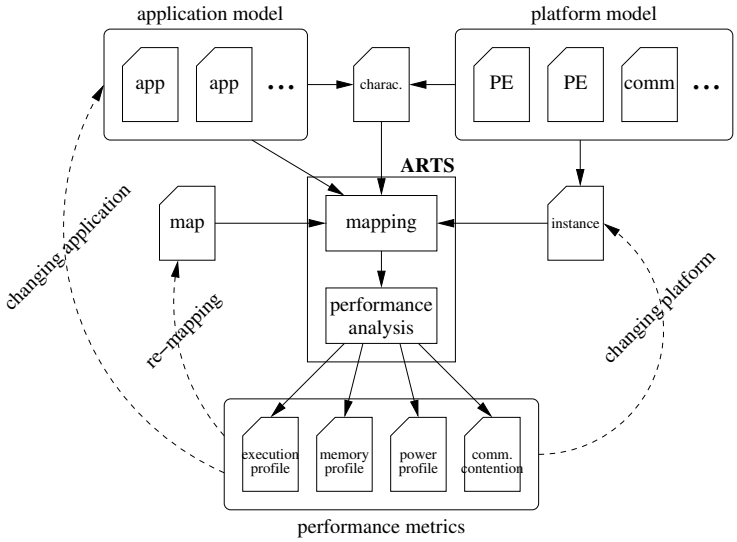


Figure 3.5: Using the ARTS framework.

the SystemC kernel uses a dynamic scheduling, which can lead to significant simulation overhead caused by excessive context switching.

In [58], the authors propose a new modelling style based on the extensions to the SystemC kernel, i.e. SDF modules communicate with each other through queue data structure, instead of the regular signal ports and channels in SystemC. Thus, the designers are required to follow some guideline rules to write simulation models. A simulation speed improvement up to 75% has been reported. However, since their method is based on the modification of the SystemC simulation kernel, new patches are needed when the SystemC kernel is updated.

3.1.5 SystemC-AMS

To capture the heterogeneity in today's systems-on-chip (SoCs) consisting of digital, analogue and mixed signal (AMS) hardware and software components, SystemC-AMS [70] has been proposed as an extension of SystemC to enhance the modelling capabilities on continuous-time components, besides the digital components and software parts within SystemC. Instead of the approach taken

in [58] to modify the SystemC simulation kernel, it is a single modelling and simulation environment built on top of SystemC discrete-event simulation kernel in a layered approach, as illustrated in Figure 3.6. The AMS related extensions are the following.

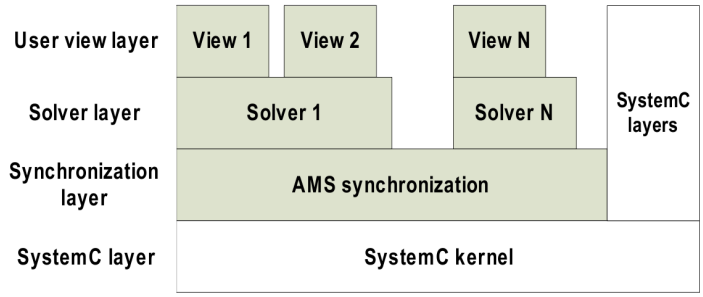


Figure 3.6: SystemC-AMS layered structure [70].

- The *synchronization layer* is based on a statically scheduled, multi-rate timed synchronous data flow (T-SDF) simulator. Based on this layer, different continuous-time solvers can be built and different models of computation can be integrated.
- The *solver layer* consists of specialized implementations to simulate the AMS components. Usually, their continuous behaviours are modelled as differential algebraic equation (DAE) to be solved in simulation.
- The *user view layer* defines the way to write executable continuous-time models for designers.

The SDF ports, signals, and modules are inherited from existing SystemC modules. Thus, it does not require any changes to the SystemC simulation kernel. Meanwhile, different models of computation are executed on top of the T-SDF simulator, which allows compile-time scheduling to speed-up the simulation.

For T-SDF blocks, timing information is denoted as the sampling-time of data tokens. The scheduling order is statically scheduled in a cluster of data flow modules, each data flow cluster is wrapped by a separate discrete-event cluster process managed by a coordinator, which handles the synchronization

with the event-driven SystemC kernel. For AMS modules, the continuous-time behaviours are encapsulated in data flow blocks.

However, the current SystemC-AMS is only a step towards a framework with seamless support on continuous-time MoCs. For instance, the numerical stability of the existing solvers still needs to be further investigated, to avoid simulation inaccuracy and invalid system behaviours. Furthermore, the CT solvers, which can be currently plugged-in, are limited to those with fixed sampling time period, caused by the fixed timing intervals between consecutive data samples in the T-SDF models.

3.1.6 OSSS and OSSS+R

With the emergence of run-time reconfigurable logic, FPGA systems tend to be more flexible and more cost efficient. However, traditional languages (e.g., VHDL, Verilog, SystemVerilog, SystemC) lack the ability to express the reconfigurations at run-time. Based on this, OSSS+R [65] has been proposed as a SystemC based software framework for high-level modelling of reconfigurable digital hardware systems. OSSS+R is designed with synthesis semantics in mind to support the synthesis of partially run-time reconfigurable (RTR) systems. It is based on OSSS [36], which is a synthesizable extension to the SystemC hardware description language, and uses the high-level synthesis tool FOSSY.

Based on the simulation of OSSS+R models, the designer can estimate the resource requirement on RTR components, while making sure that the system performance and functionality are not violated. Furthermore, the automated synthesis from a single SystemC design language can ease the RTR implementation and narrow the increasing design productivity gap.

Recently, OSSS+R has been integrated with other SystemC modelling frameworks HetSC [25] and SystemC-AMS [70] as part of the ANDRES project [26]. A design flow for adaptive heterogeneous embedded systems(AHES) has been developed, with a methodology and tools for automatic hardware and software synthesis for adaptive architectures.

3.2 SystemVerilog

SystemVerilog is an extension to the widely used Hardware Description Language Verilog, plus additional features used in hardware verification. Figure 3.7 shows the key components of SystemVerilog. This Hardware Description and

Verification Language (HDVL) was first developed by Accellera and later became an IEEE standard (IEEE 1800-2005).

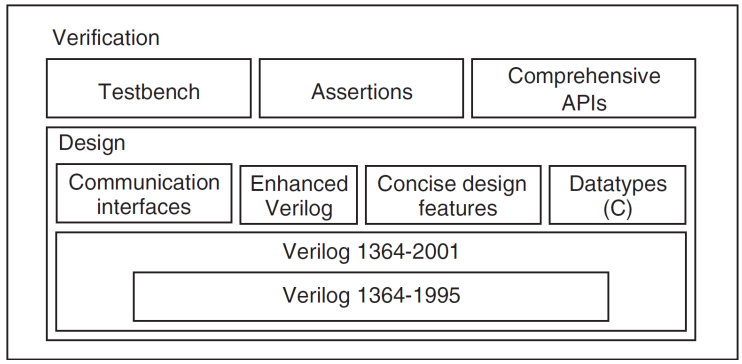


Figure 3.7: Key components of SystemVerilog [60].

SystemVerilog keeps the hardware description semantics of traditional HDLs untouched. It exploits some advanced software concepts, but they are only used for static verification and static elaboration [57]. SystemVerilog does not provide a means for modelling abstract models of computation or analogue blocks. Thus, it is not a language of choice for heterogeneous system specification.

3.3 Ptolemy

Ptolemy [20] is a widely used modelling, simulation, and design framework for heterogeneous embedded systems, in which the interactions between different components are described by various well-defined MoCs. Currently, Ptolemy is written in the object-oriented language Java. Using a visual design environment, the abstract semantics of the system in different MoC domains are constructed in a component-based manner and combined hierarchically in the same framework. Also, it covers software synthesis [15], code generation [14], co-design of mixed software-hardware system, and verification of some models.

Hierarchical abstract syntax

In Ptolemy, models are represented as clustered graphs of *actors* and their connections via *communication channels*. The abstract syntax for a hierarchical

model is illustrated by the example in Figure 3.8. There are two kinds of actors: composite actors and component actors. Composite actors are actors that can contain a hierarchy of component actors and their relations. Component actors are at the bottom of the current hierarchy and a composite actor may contain other composite actors, so the hierarchy can be arbitrarily nested. Also, a director defines the execution semantics in the current composite actor. For instance, in the graph, a top level composite actor contains component actor A1 and A2, which is realized in domain D1. Actors have ports, which are their communication interfaces. A port can be an input port (denoted as filled arrows), an output port (denoted as non-filled arrows), or both. There are communication channels between each pair of connected actors to establish their connections. The causality and concurrency in the system are explicitly expressed by the communication and data dependencies between components.

This abstract syntax can be represented in other ways, besides the graphic way in Figure 3.8, e.g., in XML files or abstract syntax tree (AST).

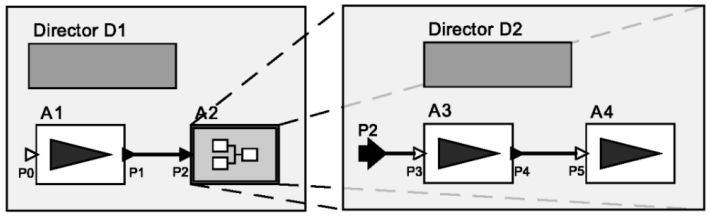


Figure 3.8: A hierarchical model in Ptolemy [20].

Model of computation domains

In Ptolemy, the domain director defines semantics to the top composite actor in the current diagram. It defines the communication mechanisms and how the diagram is to be executed. Each domain in the system implements a MoC, which is suitable to model some individual systems or sub-systems. For instance, the CT domain is good for modelling systems with continuous dynamics, the DE domain for digital system, the SDF for signal processing applications, the synchronous reactive (SR) domain for concurrent and complex control logic. In Ptolemy models, a director placed in a domain manages the execution of the model of computation. For example, a director D1 defines the flow of control and the communication semantics.

Given a synchronous language as the coordination language, the designers can get systems with DE semantics by adding discrete time information into them, get CT semantics by adding continuous dynamics, or get SDF semantics by removing the global order in signals in systems. The hierarchical composition of heterogeneous MoCs is based on a common *abstract semantics* [42] of them, which is the intersection of the semantics of different domains.

The Ptolemy framework can perform verification of some models expressed in the SR or DE MoCs, by automatic transformation from the Ptolemy model description to the mathematical models used in verification.

3.4 Generic Modeling Environment

The Generic Modeling Environment [7] (GME) is a configurable modelling GUI environment based on UML class diagrams. The configuration process itself is also a form of modelling, i.e., the modelling of a modelling process, which is also called *meta-modelling*. The configuration of GME is the first step that must be taken before anything meaningful can be done with it. The output of the meta-modelling process is a compiled set of rules, which configure GME for a specific application domain. In this sense, GME uses a single platform for both meta-model and domain-specific design.

In [52], it has been used to build a heterogeneous modelling framework EWD, which is based on the ForSyDe semantics.

Meta-model

The meta-model syntax defines what types of objects can be used during the modelling process, what attributes will be associated with those objects, and how relationships between those objects will be represented. It also contains a description of any constraints that the modelling environment must enforce at model creation time. The UML class diagrams are used to specify modelling entities and relationships.

The static semantics in GME models are specified with constraints using the standard predicate logic semantics, called object constraint language (OCL) [8].

Model refactoring

The Constraint-Specification Aspect Weaver (C-SAW) framework is a model refactoring plug-in engine for GME. It is emerging as a desirable means to im-

prove design models using behaviour-preserving transformations. The refactoring aspects and strategies provide several operators to support model aggregates (e.g., models, atoms, attributes), connections and transformations. It is integrated with GME, and can thus provide access to modelling concepts that are within the domain-specific design, e.g., design transformations defined in the ForSyDe meta-modelling environment.

3.5 UML/Marte

Marte [5, 33] is a model-driven development platform for Real Time and Embedded Systems with models written in UML. Marte depends on a Real Time Operating System (RTOS) for modelling. This dependency also influences the models of software in Marte. The generic software model API depends on and exposes the RTOS API to the actual software model. Hence the software models modelled in Marte depend on the chosen RTOS.

The hardware model API is designed to be separate from the software model API and is independent of the RTOS. The design flow in Marte is to construct the software and hardware models separately using the analysis, verification and validation API in Marte. The hardware and software models can be co-evaluated by connecting them through an interface. The main focus of Marte for analysing models are performance and schedulability analysis, but through a general framework it is possible to introduce other quantitative analysis techniques.

The dependency that Marte has to the chosen RTOS presents some drawbacks, such as needing knowledge on the RTOS, making compromises on the design based on the chosen RTOS and the fact that many RTOS has complex modelling patterns to express simple models. Furthermore the documentation of Marte is sparse. There are few or no tutorials on how to get started, and the basic modules of Marte are only documented individually.

3.6 Modelica

Modelica [4] is a modelling environment mainly targeting physical systems. The models are constructed in the Modelica programming language. Modelica in itself is only a library and does not provide any interface for usage. There exist however several interfaces for MATLAB/Simulink, Maple, Scilab, Dymola, LMS Imagine.Lab AMESim, JModelica, etc. In particular the MATLAB interface of

Modelica works similarly to MATLAB/Simulink.

Building models in the Modelica language is much like programming. The model is in most cases compiled into a binary (much like the SystemC approach) which also contains the complete simulator.

The models in Modelica can consist of several domains (mechanical, electrical, logical, etc.) as long as proper interfaces between the domains are defined. The behaviour of the models are described by differential, algebraic and discrete equations which are solved by the Modelica simulator.

3.7 MATLAB/Simulink

MATLAB [3] is a high-level programming language with graphical visualization, numeric computation and many software packages. These software packages are build on the programming and numeric calculation capabilities of MATLAB.

Simulink [3] is a software package for MATLAB which provides modelling and simulation capabilities of physical systems such as electric circuits to MATLAB. The Simulink package provides a graphical interface to build systems of blocks and subsystems. A subsystem is represented as a special block.

Simulink provides many configurable blocks with predefined features, such as different kinds of signal sources (e.g. noise), sinks (e.g. scopes), filters, addition, integration, etc. Special blocks also exist for interfacing with MATLAB functions.

The connections between the blocks can carry any type supported by MATLAB, e.g. scalars, vectors, matrices. The predefined blocks support most of the types directly.

Simulink is targeted at continuous and discrete signal processing, control design, model-based design, physical modelling, verification and modelling. MATLAB provides functionality to construct graphical interfaces for a custom MATLAB application.

3.8 The ForSyDe framework

ForSyDe (formal system design) [23,63] is a formal design methodology that targets heterogeneous embedded systems. ForSyDe uses the theory of MoCs [41] as its underlying formal foundation, which gives access to powerful analysis techniques during system design. ForSyDe designers do not need to have expertise in the underlying formal foundation, but will have access to the ForSyDe library,

which encapsulates the mathematical base of ForSyDe. ForSyDe provides libraries for several MoCs, which allows the development of executable system models from which an analysable mathematical model can be extracted. This analysable model can then serve as a base for different tools in the following phases of the design flow, such as design space exploration and synthesis.

3.8.1 System model

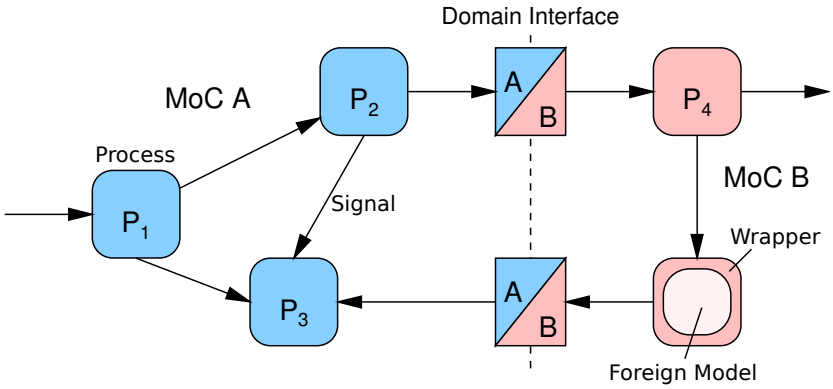


Figure 3.9: A ForSyDe System Model

Figure 3.9 illustrates the ForSyDe system model. A system is modelled as a concurrent process network, where processes belonging to the same MoC communicate via signals. ForSyDe system models do not have a global state, only local states are allowed. Rules for individual processes and mechanisms for concurrency and composition are defined within each MoC. At present ForSyDe provides libraries for four different MoCs, which allow to model heterogeneous embedded systems containing not only software but also digital and analogue hardware at an abstract level: SDF MoC, SY MoC, DE MoC, and CT MoC. Processes belonging to different MoCs communicate via domain interfaces, which define how signals from one MoC are interpreted in another MoC. A typical example for a domain interface is an abstract analogue-to-digital converter, which connects the CT MoC to the SY or the DE MoC.

Every novel design methodology has to cope with the existence of existing models or legacy code. Since these foreign models are not based on the ForSyDe formalism they cannot be treated as ForSyDe processes. However,

using ForSyDe wrappers, foreign models can be integrated into an existing ForSyDe model and co-simulated with the 'pure' ForSyDe processes.

3.8.2 Process constructors

A key concept in ForSyDe is the concept of process constructors. Process constructors lead to well-structured system models, which can then easily be converted to mathematical descriptions for which powerful analysis and synthesis methods exist.

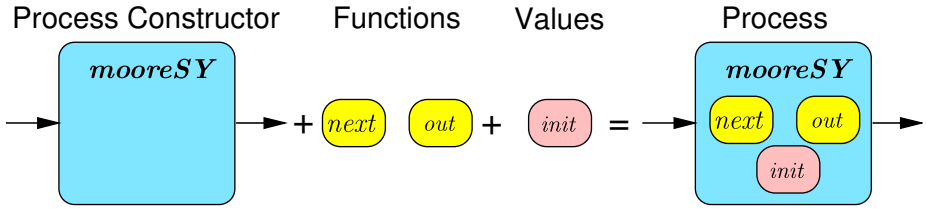


Figure 3.10: Process Constructor *mooreSY*

Each process is created by a process constructor. The process constructor defines the MoC, its interface to the environment, and a number of arguments that have to be supplied to the process constructor. Figure 3.10 illustrates this concept by means of the process constructor *mooreSY*, which is used to model a finite state machine and belongs to the SY MoC. *mooreSY* takes two functions and a value as arguments. The function *next* returns the next state of the state machine based on the present state and the current input values, the function *out* returns the output value based on the present statue, and the value *init* gives the initial state.

The ForSyDe methodology obliges the designer to create processes using process constructors. This gives several important benefits: (1) A system model is well-structured and well-defined, because each process constructor has a mathematical formulation, (2) process constructors separate communication (process constructor) from computation (function), (3) process constructors can have an implementation pattern (the process *mooreSY* can be efficiently implemented in software or hardware using a well-known design pattern).

3.8.3 Implementation of the ForSyDe library

ForSyDe has originally been implemented in the functional language Haskell. Haskell fits perfectly with the formal foundation of ForSyDe, since it enforces side-effect free processes, provides lazy evaluation, and allows to express the process constructors with higher-order functions. The Haskell implementation supports four MoCs, and provides a hardware synthesis back-end, which allows to generate synthesizable VHDL from a SY MoC ForSyDe model [23].

After demonstrating the potential of ForSyDe with Haskell, a SystemC version of ForSyDe has been developed within the European Artemis project SYSMODELSYSMODELWWW to increase industrial usability. SystemC is an industrial standard and widely used in industry for system modelling. However, SystemC lacks a clear formal semantics, which makes it very difficult to use it in its plain form for other purposes than modelling and simulation. Inside the SYSMODEL project we have created SystemC libraries based on ForSyDe for four MoCs, which ensures that ForSyDe SystemC models have a formal base and that abstract analysable models, such as SDF-graphs, can be easily extracted from an executable ForSyDe SystemC model. Although, compared to Haskell, SystemC suffers from some drawbacks, in particular SystemC does not enforce side-effect-free processes, there is in addition to industrial acceptance another very important advantage: SystemC is a C++ class library and all functions are written in C/C++, which allows to directly implement the function arguments to the process constructors as C-code on the target processors.

The ForSyDe SystemC libraries implement process constructors as abstract classes, where arguments to the process constructors are implemented as pure virtual functions and initial values as arguments to the class constructor. For each process, the designer derives from the abstract class implementing the desired process constructor and writes the required virtual functions and then provides the class constructor arguments during instantiation. Then the processes are connected via ForSyDe SystemC channels. Simple SystemC channels implement ForSyDe signals, while more complex channels can implement domain interfaces.

3.9 The UPPAAL framework

UPPAAL [13,37] is a framework that provides modelling, simulation and verification. The framework models collections of non-deterministic processes with finite control structures. Real-valued clocks, communication channels, and shared

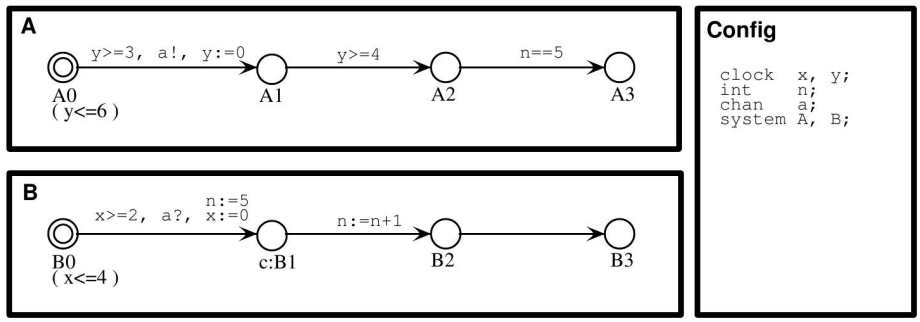


Figure 3.11: Example of a UPPAAL model [37].

variables are available to use in the models. It is designed to check invariant and reachability properties. This is done by exploring the state-space of the models.

With on-the-fly searching techniques and symbolic techniques UPPAAL reduces verification problems to manipulation and finding solutions of simple constraints. UPPAAL can provide traces to exemplify why a property is satisfied or not.

UPPAAL models consist of one or more models expressed in finite automata as shown in Figure 3.11 (A) and (B). The models can interact with variables and communicate through channels specified in the config. Each model specifies an initial state (A0 and B0) and a number of other states connected by transitions. A state can have restrictions of how long the automata may stay in this state before taking a transition. A transition can also have restrictions on when it may be traversed. A model will deadlock if it can neither stay in a state or traverse any of the transitions away from the state.

UPPAAL implements timed automata, and the notion of staying in a state is in UPPAAL terms to take a delay transition. In Figure 3.12 an example model showing synchronisation between three models is shown. In this example it is required that all three models synchronise atomically. To ensure that the sending model (S) does not take a delay transition in state (S2) it is marked as *committed* (the c: prefix to the state name). This guarantees that the state is left immediately after entering it.

The UPPAAL models can be simulated through manually taking a transition in one model at a time. UPPAAL provides a list of possible transitions to

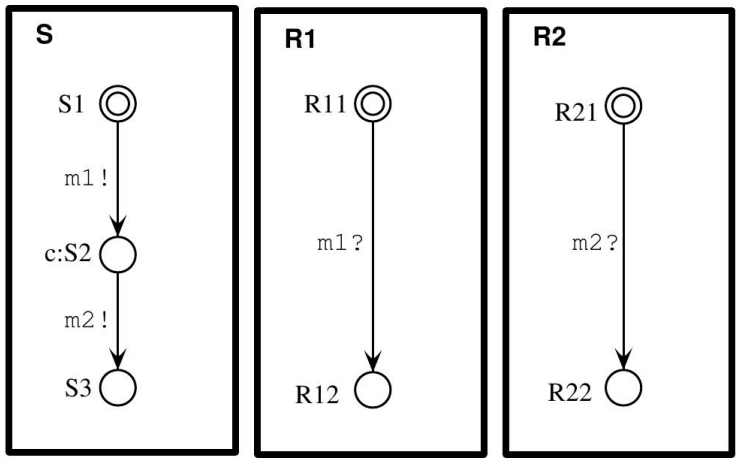


Figure 3.12: Example of communication between sub-models in a UPPAAL model [37].

take. Another possibility is to let UPPAAL explore all such simulations, i.e. to perform a static analysis/verification of the models. This can be expressed with queries of the form: $\phi ::= \forall\Box\beta|\exists\Diamond\beta$ where $\beta ::= a|\beta_1 \vee \beta_2|\ \not\beta$. a is an atomic formula. This formula can be an atomic clock, constraint, or data.

3.10 Summary

An overview of all the modelling frameworks is shown in Table 3.1. It compares the following features of the frameworks: Heterogeneous models, dynamic models, domain specific models, generic models, simulation support, and support for formal analysis.

Some of the abilities of the frameworks are put in parentheses. SystemC does not officially provide either a CT MoC or the possibility to model dynamic systems. However, an extension named SystemC-AMS provides the CT MoC and by careful programming it is possible to change the structure of a model at runtime. Furthermore, ForSyDe is also implemented as an extension to SystemC [56].

In the Generic Modeling Environment (GME) framework, there are no prede-

	Heterogeneous	Dynamic	Domain specific	Generic	Simulation	Formal analysis
SystemC	✓	(✓)	SY, TLM, DE, (CT)	✓	✓	×
SystemVerilog	×	×	HDL	×	✓	✓
Ptolemy	✓	×	SR, SDF, DE, CT	×	✓	✓
Generic Modeling Environment	✓	×	(✓)	✓	×	✓
UML/Marte	✓	×	Software, hardware	×	×	✓
Modelica	✓	×	Mechanical, electrical, logical	×	✓	×
MATLAB/Simulink	×	×	CT	✓	✓	✓
UPPAAL	×	×	Finite non-deterministic timed automata	×	✓	✓
ForSyDe	✓	(✓)	SY, SDF, DE, CT, ...	✓	✓	(✓)

Table 3.1: Overview of abilities of modelling frameworks.

fined definitions of modelling, it only defines a meta modelling language. Therefore, one must first to define the MoCs required for modelling.

ForSyDe as presented in this thesis does not support dynamic models or formal analysis directly. However, it is possible to describe dynamic systems through for example a non-finite state machine. The formal analysis of ForSyDe models is made possible through external tools by extracting the structure of the ForSyDe models.

The first choice of framework is the ForSyDe framework (requested by the SYSMODEL project). The SYSMODEL project also requests that ForSyDe is extended from the original version (only with the SY MoC) presented in [62] with the SDF, DE, and CT MoCs. Since the SystemC framework has a well establish usage in industry, the ForSyDe framework is to be implemented in SystemC in the SYSMODEL project. However, a reference implementation will be made in the language Haskell.

Three other frameworks are interesting to look at: Ptolemy, GME, UPPAAL. They each provide some form of formal analysis of models.

Ptolemy and UPPAAL can perform similar types of verification, however, Ptolemy does not provide verification of all models expressed in the MoCs of Ptolemy. UPPAAL provides a dedicated modelling language which supports verification of all models expressed in UPPAAL. GME use another approach to formal analysis. GME can check a set of constraints on a model.

The ability of checking invariant and reachability properties of UPPAAL are interesting with respect to the WSN use case. So it is an interesting alternative framework to ForSyDe.

Chapter 4

Formal analysis of DEHAR in UPPAAL

Tests of the performance of wireless sensor network (WSN) applications and algorithms are mostly performed by simulation or real world tests. This only provides validation of certain execution traces. Verification is a stronger tool but it is not always feasible for large applications and networks. Verification of a small model of a WSN is demonstrated for a specific routing algorithm distributed energy harvesting aware routing (DEHAR) presented in Chapter 2. UPPAAL is used to verify the functionality of the algorithm for specific network structures and specific power production patterns.

4.1 Introduction

A WSN is an autonomous low energy system, which is not easily reachable (i.e. to perform service). It consists of many independent nodes (denoted by N_i) which communicate with each other using radios. In this particular WSN it is assumed that each node is capable of recharging by harvesting limited amounts of energy from the environment. Furthermore it is assumed that each node can only reach a limited number of other nodes (neighbours) in the network (a multi-hop ad-hoc network).

The main task of a WSN is to collect data from the environment (here produced by an **Application**) and transfer it to a computer for processing. This is done by routing these data messages to a base station (denoted by X), which

is able to reach f.ex. the internet. When this data message reaches the base station it is considered to be routed successfully.

4.1.1 DEHAR algorithm

This routing of data messages must be ordered in order not to waste energy. The network is constructed with the knowledge of the shortest path to sink (simplification for this project). Since the nodes have limited energy resources, some nodes might run low on energy. Therefore an algorithm to modify the routes to take available energy into account is added. This algorithm is known as DEHAR.

The aim of the DEHAR is to find the optimal path based on the two parameters energy and distance. It is assumed that the nodes must run this algorithm with only the knowledge of their neighbours in a distributed manner.

All these assumptions and tasks add up to five distinct tasks for each node: An application, receiving and sending messages, updating a nodes available energy and updating information about neighbours. Furthermore a node must be initialised at start and can potentially run out of power at a later point.

The energy production is modelled together with the environment and not the individual node. The base station is assumed to have infinite energy and does only need to collect the data messages produced in the network.

4.1.2 Verification goals

The WSN is modelled in UPPAAL [69] to verify the ability of a given network design to maintain network structure and function. Maintaining the structure and function implies that the network must not split into multiple sub networks, data messages must not be lost or delayed too much and the network should provide a certain spatial density (number of nodes in a region to collect data from the environment). It is also interesting to verify whether the DEHAR performs as expected and changes routes when the network is stressed.

The model presented, does have some pitfalls. One of the larger ones is the discretisation of the power production. In the model the power production is performed every 60 clock ticks which is somewhat too large. The reason not to increase sampling of power production is to keep down the verification time.

4.2 Network model

The network model is constructed in the tool UPPAAL. Models used for verification need to be simple. To achieve this, the WSN is analysed as follows: A WSN consists of several nodes and a base station. These nodes and the base station interact with each other via radio and are influenced by the environment. In the particular subset of WSNs it is assumed that the nodes are capable of recharging by harvesting energy from the environment. The nodes are in this assignment assumed to have only one processor which controls all other hardware, e.g. that they can be modelled with a single model.

To keep the system simple the number of models must be kept to a minimum; Each node must have its own model (can be from the same template) since they are by nature parallel and synchronised by radio communication only. Furthermore both the base station and the environment must each have a model since they have unique tasks to perform in parallel.

The model of the environment models the energy harvested from the environment, which consists of modelling a global sun strength and a local shadow. This results in each node having a unique energy production.

The base station model is responsible for collecting all data packages produced in the WSN. Otherwise the base station does nothing.

The node model, on the other hand, has many different tasks which must be modelled. First the main goal of any WSN is to collect data, which the application is responsible for doing. Secondly the routing of data towards the base station must be maintained dynamically.

Global declarations contain the structural information about the network in several constants. There are three groups of the most important constants: network structure, environmental energy and node design.

The network structure constants consist of: The number of Nodes (in this example one base station + four nodes), The maximum number of Neighbours per node, the actual number of neighbours (numNeighbours[i]) per node, the neighbours neighbours[i][n] and the distance to base station (hops[i]).

The node design constants are: fixed point precision for energy calculation (Decimals), BatteryCapacity and the energy usage of tasks (ApplicationUsage, RouteUsage, UpdateEnergyUsage and UpdateNeighbourUsage).

The environmental energy is based on a global insolation and a local shadow. Both are constructed with four constants each: number of distinct items/events, and index in the item list, a value and a time of how long each value applies.

The list of constants is: insolation_value[s], insolation_time[s], insolation_index, insolation_items, shadow_value[d], shadow_time[d], shadow_index and shadow_items.

There are also some global functions, where the most important one is getDepletion(batteryCharge). This function implements the algorithm of DEHAR. Other functions can charge(i, energy) a node and consume(i, energy) power. Most of the other functions are convenience functions to simplify the model layout and increase readability.

4.2.1 Node template

The node template is the most complex of them all. It is shown in Figure 4.1. It is initialised when leaving the state **Init**. The node is allowed to stay a fixed amount of time in **Idle** until either the **Application** or the **UpdateEnergy** task is activated, or if there are data messages to transmit the **RouteSend**. The **UpdateEnergy** (and **UpdateNeighbour**) will (can) invoke the **UpdateNeigbour** in all of the neighbours of a node. The **RouteSend** will invoke **RouteReceive** in a specific neighbour decided by the local variable route.

Each of the above mentioned states (except **Init** and **Idle**) will consume energy on the edge leading to them. If the node should run out of power, it enters the state **OutOfPower** and stays there until it has harvested enough power to restart.

The routing of data messages is done only by synchronisation with data[i] and counting the number of packages stored temporarily. The sender chooses the destination channel and receiver listens on its own channel.

The synchronisation of energy updates works the other way around by letting the sender broadcast on its update[i] channel and the neighbours listen on its neighbours channels. Whenever the sender synchronises on update[i] it updates height_update with its current energy information. This broadcast synchronisation will spread through the network like rings in the water until terminated by the guard hasValidRoute() in the edge from **UpdateNeighbour** to **Idle**. This guard is normally true.

The nodes starting time (leaving state **Init**) are interleaved to minimise concurrent actions which will increase the state space.

Declarations for the node template contains local data structures and functions. The most important is the route variable which points to the neighbour which is best to route data messages to. It also has a variable used to maintain this route named augmentation and a table of the neighbours status in

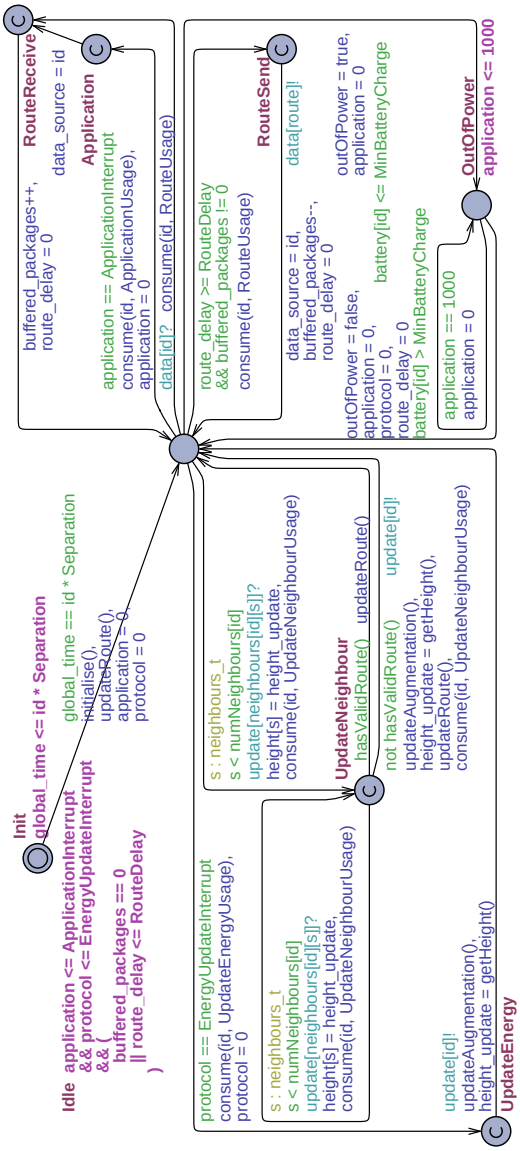


Figure 4.1: The node model

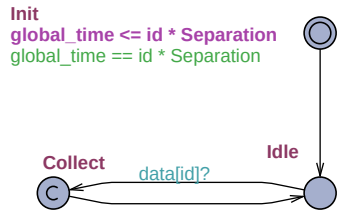


Figure 4.2: The base station model

height[n]. For the specific implementation of counting the number of packages stored temporarily before routing is kept in buffered_packages.

The five local functions consist of one for initialising the node (initialise()), calculating the status of this node (getHeight()), updating the route (updateRoute()), updating the local routing parameter with updateAugmentation(), and checking whether an optimal route has been reached (hasValidRoute()).

4.2.2 Base station template

The base station is built of three states shown in Figure 4.2. The initial state **Init** which leads to the **Idle** state. Whenever the base station is to receive a data package it enters the **Collect** state on synchronisation with the data[id] channel.

The reason for the state **Init** is to control the sequence of initialisation of all models (all models but the base station needs initialisation in this particular system). Control of the sequence of initialisation is performed to limit the state space.

4.2.3 Environment template

The environment model needs to perform three tasks, update the global insolation, the local shadow and update the battery status of the nodes. The model is shown in Figure 4.3. The model used in the verification, however is a bit more complex because of a runtime error in the composed functions chargeAll(getSunStep()). There is no mathematical difference between the two models, the difference should only be that the one used performs all calculations directly in the updates in the edges instead of using functions.

The functions updateShadow() and updateSun() each cycle through a range. The value then represents an index in an array of shadows and insolation val-

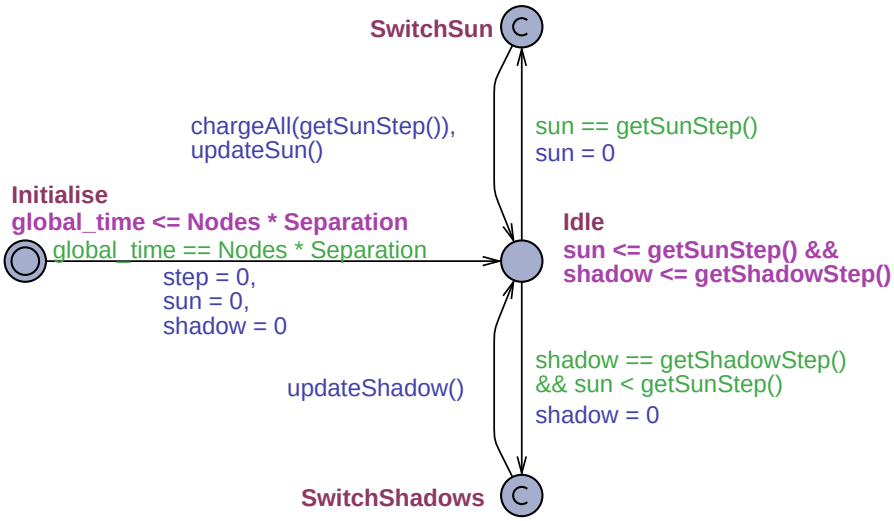


Figure 4.3: The environment model

ues. E.g. if day and night are modelled as either 1 or 0 the insolation array insolation_value is $[0, 1]$ or the reverse depending on if the model starts in day or night. The insolation array has an associated array for the duration of each insolation state, which could be $[12 \cdot 60 \cdot 60, 12 \cdot 60 \cdot 60]$ for 12 hour duration of each state modelled in seconds. The function getSunStep() queries the duration of the current insolation state and likewise with getShadowStep().

4.3 Verification

The computation tree logic (CTL) queries are available in the "circle.q" file in the same sequence as described in this section. This query file together with the model can be found in Appendix A.5. The last queries which fail due to a bug in UPPAAL are put in the comments only. The queries are expected to be satisfied if not noted otherwise. Those queries where the node number is denoted by i are expanded into a query for each node in the query file.

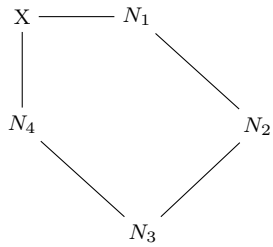


Figure 4.4: A node layout of 4 nodes and a base station arranged in a circle. 'X' marks the base station while the nodes are represented with N_i for $i \in [1, 2, 3, 4]$.

4.3.1 Network structure

The equations (4.1)-(4.4) are constructed to verify the routing of data in the WSN. Where (4.1) and (4.4) proves that nodes N_1 and N_4 always transmit packages to the base station. Nodes N_2 and N_3 are not neighbours of the base station and can potentially have two different routes to it. This is proven with (4.2) and (4.3).

The variable data_source is a global variable set to the source of radio transmission in the edge going from $N_i^{\mathsf{RouteSend}}$ to N_i^{Idle} and from $N_i^{\mathsf{Application}}$ to $N_i^{\mathsf{RouteReceive}}$. The reason to use this variable is; since $N_i^{\mathsf{RouteSend}}$ will always be reached at a later point (due to $N_i^{\mathsf{Application}}$) for any node i, it must be ensured that the right receiver is matched in the verification. It is guaranteed that data_source is not updated until after the neighbouring node has reached **RouteReceive** because the edge updating data_source is synchronised with the edge entering **RouteReceive**.

$$N_1^{\mathsf{Application}} \rightsquigarrow \left(X^{\mathsf{Collect}} \wedge \mathsf{data\_source} = 1\right) \tag{4.1}$$

$$N_2^{\mathsf{Application}} \rightsquigarrow \left(\left(N_1^{\mathsf{RouteReceive}} \vee N_3^{\mathsf{RouteReceive}}\right) \wedge \mathsf{data\_source} = 2\right) \tag{4.2}$$

$$N_3^{\mathsf{Application}} \rightsquigarrow \left(\left(N_2^{\mathsf{RouteReceive}} \vee N_4^{\mathsf{RouteReceive}}\right) \wedge \mathsf{data\_source} = 3\right) \tag{4.3}$$

$$N_4^{\mathsf{Application}} \rightsquigarrow \left(X^{\mathsf{Collect}} \wedge \mathsf{data\_source} = 4\right) \tag{4.4}$$

The X represents the base station and N_i represents the nodes in the network. The superscript for the nodes and base stations describe a state of them.

The main use of this set of verification queries is to verify the network structure. The data structure which stores the network structure is an adjacency list neighbours$[i][j]$ with $0 \leq i <$ number of nodes and $0 \leq j <$ number of neighbours. If this adjacency list does not have bidirectional connections for all neighbours the network does not behave as expected. This set of verification queries is of course network structure specific.

An alternative approach to verifying the network structure is to check that the route of a node always corresponds to the subset of nodes which are neighbours of it. It is required, however, that the test is not applied when in **Init** since there the route is 0 (not initialised yet). The query is shown in (4.5).

$$\forall\Box \left(N_i^{\text{route}} \in \text{neighbours}[i] \vee N_i^{\textbf{Init}}\right) \tag{4.5}$$

This query, however, does not verify that the neighbour reacts on the transmission, it only verifies the intent to send. neighbours$[i]$ is the set of neighbours of N_i. This query has to be expanded into verifying the disjunct equality for each member in the set before implementing in UPPAAL.

4.3.2 Battery charge and routing performance

If it can be verified that no node in the network will run out of power at any time, then there will be no routing performance degradation. This can be performed with the query (4.6).

$$\forall\Box\neg\text{outOfPower} \tag{4.6}$$

where outOfPower is a global boolean variable which is true as long as any node is in the state **OutOfPower**. (The use of the clock **application** $<= 1000$ in **OutOfPower** is to avoid zeno behaviour and limit the number of state transitions.)

This query might impose too hard constraints (if required to be true) on the network. The designer could allow that some nodes can run out of power with the requirement that the network is not segmented. This can be achieved by a query for deadlocks (or the absence of these).

$$\forall\Box\neg\text{deadlock} \tag{4.7}$$

The reason that this query implies the above statement is that only the **Idle** state of a node accepts a data transmission (synchronisation on data[i] $\forall i$). If a node is in **OutOfPower** then no other node may try to synchronise with this nodes data channel. For the broadcast channels it must not be the case that all neighbours of a node is in **OutOfPower**. This, however, does not impose any extra restrictions on the network than the data channels with respect to deadlocks. (Restrictions are e.q. battery size, minimal power production and other design parameters in order for the network to perform as intended by the designer. I.e pass the required queries.)

The query (4.7) must be satisfied but (4.6) needs not to be satisfied. If (4.6) is not satisfied then the the query (4.8) can be used to verify which node(s) enters the state **OutOfPower**.

$$\exists \Diamond N_i^{\mathsf{OutOfPower}} \tag{4.8}$$

Whenever this query is satisfied for i then N_i is at some point out of power.

4.3.3 Alternate routes

The main objective of the routing algorithm (DEHAR) is to find the optimal route with two parameters, energy and distance. Therefore it is interesting to see whether the routes do change in the network. Queries to check for the use of alternative routes are 4.9 and 4.10.

$$\exists \Diamond \left(N_2^{\mathsf{route}} \neq 1 \wedge N_2^{\mathsf{RouteSend}} \right) \tag{4.9}$$

$$\exists \Diamond \left(N_3^{\mathsf{route}} \neq 4 \wedge N_3^{\mathsf{RouteSend}} \right) \tag{4.10}$$

By inspection of the network structure, it can be verified that N_2 has the shortest route to the base station through N_1. Likewise N_3 prefers N_4. The query asks for the possibility that a data package is transmitted when the current best route is not along the shortest path. These queries will give a negative result if the particular node does not change the route during its life (forever).

4.3.4 Energy change leads to optimal route

The DEHAR is to find the optimal route from the given towards the base station. This is achieved by radio broadcasts between the nodes (i.e. synchronisation on update[i]). In the model it has the side effect that several nodes can be active

at the same time, therefore **UpdateNeighbour** must be input enabled on the broadcast channel.

To check that this system works amongst all the nodes, the query (4.11) can be used.

$$\left(N_i^{\text{UpdateEnergy}} \vee N_i^{\text{UpdateNeighbour}}\right) \rightsquigarrow N_i^{\text{hasValidRoute()}} \tag{4.11}$$

Sadly this query is not stable in UPPAAL, and can result in any of three outcomes: satisfied, not satisfied or a crash of the verification engine.

4.4 Summary

It is shown that the model of the WSN, with a given structure and node design, can potentially always route data from any node to the base station. The routing can not be verified completely due to a bug when verifying that any node will always converge at an optimal route. If however the nodes always converge at an optimal route the other queries can prove that the given network will be able to perform the required routing. I.e. no routes go through dead nodes (the deadlock query). It is also verified that the DEHAR algorithm does use alternative routes when stressed with low energy availability.

There are, however, as stated some discretisation problems that must be improved before this verification can be assumed to hold for a real WSN implementation. There are also other differences which are not included in the presented model, such as: task scheduling, power consumption in idle, radio transmission time, processing time, etc.

The verification of the model becomes very time consuming when lowering the power production (globally and locally) of the nodes, making it difficult to verify larger models.

Chapter 5

Theory of systems modelling

There exists many different approaches to modelling systems. Some are generic while others are domain specific. The challenge is to find the best matching domain specific approach for the purpose. The domain specific UPPAAL framework used to model the wireless sensor network (WSN) in Chapter 4 captures static systems well, but not dynamic systems.

The more generic a modelling approach is the more different kinds of systems it can express and the harder it gets to express/extract domain specific knowledge of a system (because of lack of the right structure). In contrast, a domain specific modelling approach is designed to express certain types of models and may be inappropriate to use to express other types of models.

A model of computation (MoC) defines a formal method of performing calculations. This definition of a MoC is very broad and a MoC can be made to fit each unique generic or domain specific modelling approach. The MoC makes it possible to perform calculation (simulation) of a model and the structure of domain specific MoCs makes is possible to analyse the structure of the model.

An analysis of the structure of a model can provide various information. Examples are: extracting a static schedule of a set of tasks when mapped onto a set of processing devices, analysing reachability of states in a state machine, estimating power consumption, estimating memory usage, etc.

In this chapter, the theory of ForSyDe is presented.

Figure 5.1: Example of functional model.

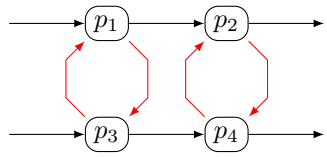


Figure 5.2: Example of functional model (p_1 and p_2) connected to a platform (p_3 and p_4).

5.1 Basic concepts

A set of concepts, used throughout the rest of the thesis, are defined here. These concepts are homogeneous and heterogeneous models and static and dynamic models.

A model of a system expressed in one MoC is a *homogeneous* model. An example is shown in Figure 5.1 and is discussed in detail through the functional model of the SIB use case in Chapter 6.

It may not always be practical to express a system in one specific domain. Instead of just using a generic MoC to express the system, it can be beneficial to divide the system into multiple sub-systems expressed in different MoCs. Such a model is a *heterogeneous* model. An example of a heterogeneous model is to add a platform model described in another MoC to the functional model of Figure 5.1 as shown in Figure 5.2.

The examples (both functional model and the combined functional and platform model) are examples of *static* models. A static model is a model where the state space of the model is static. None of the possible inputs to the model can increase the state space of the model. In contrast, a *dynamic* model can change its state space through input. This is the case in the WSN use case (see Chapter 2) where nodes (each having a constant state space) may be added or removed throughout the lifetime of the network. Thus the state space of the network model is changing, i.e. it is dynamic.

When placing homogeneity/heterogeneity on the x-axis and static/dynamic on the y-axis we have four types of models (see Figure 5.3). The use cases

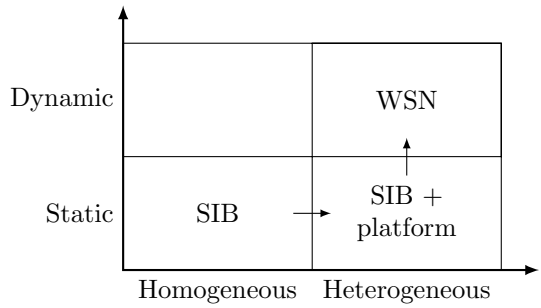


Figure 5.3: The model space of static/dynamic models versus homogeneous/heterogeneous models. The models of the use cases in this thesis are placed in the grid.

can then be plotted in this graph. The SIB use case (see Chapter 6) and the WSN use case (see Chapter 2 and Chapter 7). The SIB use case is modelled in two ways: the core functionality only and combined with a platform. The core functionality is modelled as a static and homogeneous model as seen in Figure 5.3, while adding the platform makes it a heterogeneous model.

The WSN use case is a set of nodes. A node has a core functionality and a platform like the SIB use case. The network of nodes is dynamic, hence the WSN model is dynamic and heterogeneous. The analysis of WSN using UPPAAL (see Chapter 4) is an attempt to express a dynamic system as a homogeneous model. The actual model expressed in UPPAAL is, however, a static model. It is not impossible to express a model of a WSN in only one MoC, but it does have some challenges and drawbacks.

The model in UPPAAL expresses a static homogeneous model, since only a static number of nodes are present. Even if one models nodes entering the network as turning on a node from a pool of offline nodes (and vice versa for removing), it is still a pool of a static size. Thus, if UPPAAL is able to calculate the truth value of a query, it reasons about all nodes regardless if it is in the offline pool or active. Hence, the model is static, as the analysis covers all states of node activity and connectivity.

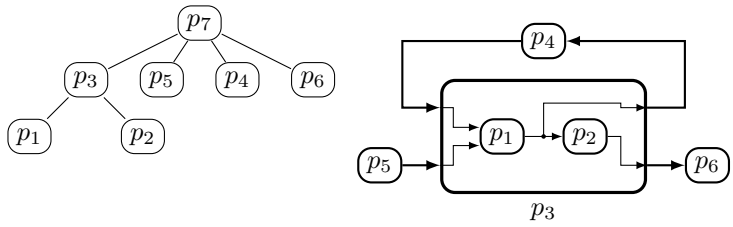


Figure 5.4: Structure of ForSyDe models. p_3 and p_7 are hierarchical processes, the rest are leaf processes. **Left**: Hierarchy of processes. Siblings are part of the same process network. **Right**: The process network.

5.2 Modelling with ForSyDe

ForSyDe is one of many heterogeneous modelling frameworks (see Chapter 3). The key features of the ForSyDe framework are simulation of models and extraction of model structure.The extracted structure can be used for formal analysis and for translation into other languages, such as synthesis of synchronous (SY) MoC into VHDL. ForSyDe also provides domains and domain interfaces to structure heterogeneous models. ForSyDe existed before the work of this thesis was done, as described in the following section. Hereafter, the additions to ForSyDe are described.

5.2.1 The original ForSyDe

ForSyDe [62] originally defines the concepts: leaf process, hierarchical process, process network, and process constructor. The SY MoC in ForSyDe defines a set of processes which, connected by signals, constitutes a process network. A process can be of two types, a leaf process (the functionality is specified by a function) or a hierarchical process (containing a process network). Processes are concurrent and communicate only via signals. In Figure 5.4 the processes p_1, p_2, p_4, p_5, and p_6 are leaf processes and p_3 and p_7 are hierarchical processes. The process p_7 is only present in the hierarchical view (left) and represents the entire process network in the model (right).

Signals are sequences of values with a global ordering $\boldsymbol{s}_1 = \langle v_1, v_2, \ldots, v_n \rangle$. The ordering is defined by the index of the value in the signal (sequence). I.e. values with the same index occurs at the same time in all signals as in $\boldsymbol{s}_2 = \langle w_1, w_2, \ldots, w_n \rangle$ where v_i and w_i occurs at the same time.

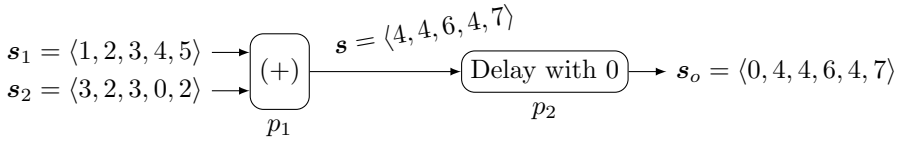


Figure 5.5: Example of a process network in the original ForSyDe.

To illustrate the above, a signal is written as a number of tokens enclosed in angular brackets, e.g.: $\boldsymbol{s} = \langle s_1, s_2, s_3 \rangle$. The signal sequence is constructed by a catenation operator (:), and the notation relates to the other signal notation in the following way:

$$\boldsymbol{s} = s_1 : s_2 : s_3 : \langle\rangle = s_1 : s_2 : \langle s_3 \rangle = s_1 : \langle s_2, s_3 \rangle = \langle s_1, s_2, s_3 \rangle \tag{5.1}$$

The empty signal ($\langle\rangle$) is itself considered a special token of signal termination.

A small example of a process network, and a simulation of it, is shown in Figure 5.5. It displays the input signals $\boldsymbol{s}_1$ and $\boldsymbol{s}_2$ to the process p_1 which adds its inputs, the intermediate signal $\boldsymbol{s}$ and the output signal $\boldsymbol{s}_o$ where the last signal has been delayed by the process p_2. The initial value of $\boldsymbol{s}_o$ is 0 in p_2.

A process is constructed using a process constructor. A process constructor defines the number of inputs a process has. A leaf process always has one output. A process constructor has other arguments like function(s) and/or initial value(s). There exists infinitely many process constructors, one per unique number of inputs. Each of these process constructors must be defined before they can be instantiated as processes. This is explained in detail in the next section and illustrated for the one-, two-, and three-input processes in Figure 5.6.

Definition of processes, process networks, and process constructors

- A process constructor is a template for leaf processes.
- A leaf process is an instantiation of a process constructor.
- A hierarchical process is a process containing a process network.
- A process network is a set of processes connected by signals.

5.2.2 Generic definition of models of computation

ForSyDe originally only provided an implementation of the SY MoC. It has now been extended to contain the four MoCs: SY, synchronous data flow (SDF), discrete event (DE), and continuous time (CT). The concepts of process constructors, leaf processes, hierarchical processes, and process networks of the original ForSyDe are kept. The definition of signals is changed slightly to be sequences of *tokens* instead of values. (Values are then a specific type of tokens.) The concept of a *process atom* is added to the new ForSyDe.

The reason why the concept of process atoms is added is: A process constructor was originally made from scratch every time an extra input where required in the final process. The implementation of these process constructors contains much redundancy and becomes more complex the more inputs a process requires. The challenge is that there does not exist a proper set of process constructors which can express any other process constructor. For some MoCs it can be trivial to express a 3-input process constructor as the combination of two 2-input process constructors. However, for some MoCs this is not possible.

Process atoms are closely related to process constructors. (In some MoCs they are actually the same as process constructors.) There is a finite set of process atoms for each MoC. The process atoms can be combined to any other process constructor. This solves the challenge of infinitely many different process constructors in the original ForSyDe and helps to avoid implementation errors when building new process constructors. An example of how process atoms are combined to process constructors with varying number of inputs is shown in the right column of Figure 5.6.

Figure 5.7 illustrates how the process atoms are used in models, compared to the original ForSyDe way shown in Figure 5.4. This also illustrates that the concepts of leaf/hierarchical processes becomes a bit vague. The process atoms in Figure 5.7 could be interpreted as processes and all of p_i could be interpreted as hierarchical processes. This is an unintended side effect of drawing the process atoms in the model. The intention is that the collection of process atoms that replaces the process from the original ForSyDe together represents a process.

The four MoCs currently defined in ForSyDe have three process atoms in common. The map atom ($\odot$), the zip atom ($\oplus$) and the delay atom (Δ).

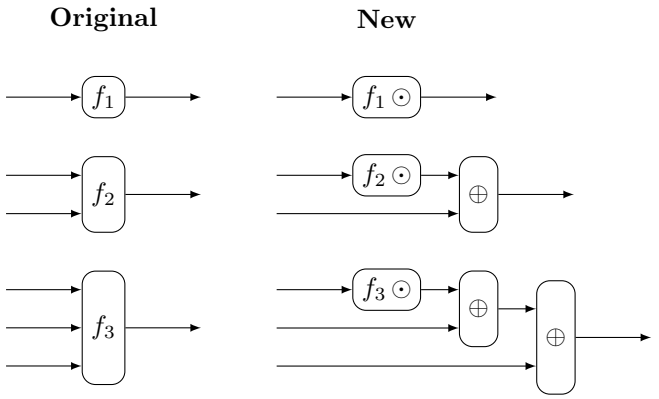


Figure 5.6: Comparing the original ForSyDe with the contributions of this thesis. The left column represents process constructors in the original ForSyDe and the right column the equivalent by using the process atoms. Examples are shown for one, two and three input process constructors and the sequences continue for more inputs. The original ForSyDe must define a new process constructor for each added input, while the new structure just adds an extra process atom.

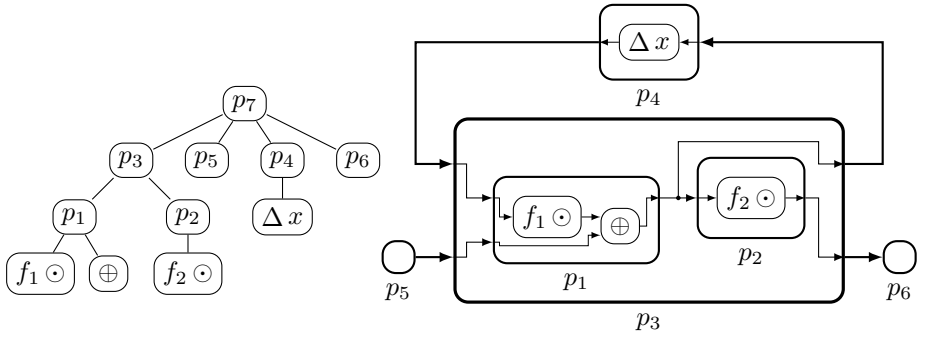


Figure 5.7: Structure of ForSyDe models using process atoms. p_3 and p_7 are hierarchical processes, the rest are leaf processes. **Left**: Hierarchy of processes. **Right**: The process network. (See also Figure 5.4.)

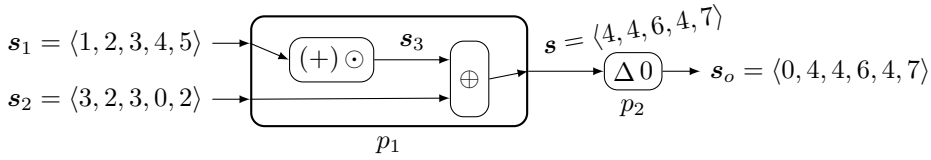


Figure 5.8: Example of a process network in the new ForSyDe. This example is the same as in Figure 5.5 replacing process p_1 with process atoms.

Definition of process atoms

- $\odot$ Maps the provided function f onto values of the input signal.
- $\oplus$ Applies the functions of the first signal with the values of the second signal. The ordering of the tokens is defined by the MoC.
- Δ The delay process atom implements explicit or implicit timing of the MoC.

The operators are all left associative with the same precedence. I.e. $f \odot \boldsymbol{a} \oplus \boldsymbol{b}\, \Delta\, z_0 = (((f \odot \boldsymbol{a}) \oplus \boldsymbol{b})\, \Delta\, z_0)$.

Signals are sequences of *tokens* defined by the MoC. The ordering of tokens in signals is defined by the MoC. Examples of possible ordering of tokens are: global order and partial order. With *global* ordering, any token in any signal happens strictly before, after or at the same time as some arbitrary other token. With *partial* ordering, tokens are ordered in each signal but has no order when comparing two different signals.

The example in Figure 5.5 in the original definition of ForSyDe the plus-process is replaced by a map and a zip process atom as shown in the second row of Figure 5.6. The resulting figure of the transformation is shown in Figure 5.8. The intermediate signal ($\boldsymbol{s}_3$) from the map to the zip process atom would contain a sequence of partially applied plus operators $\boldsymbol{s}_3 = \langle (1+), (2+), (3+), (4+), (5+) \rangle$. The values of the signal $\boldsymbol{s}_3$ represents functions, e.g. the first value $(+1)$ is a function which adds one to its input. I.e. $\boldsymbol{s}_3 = \langle \mathrm{Add}_1, \mathrm{Add}_2, \mathrm{Add}_3, \mathrm{Add}_4, \mathrm{Add}_5 \rangle$ where $\mathrm{Add}_i = \lambda x \to i + x$.

The map atom (see Figure 5.9) works like a generalised function application.

Figure 5.9: The map process atom. For all tokens in s_i carrying a value, the function f is used to calculate the replacement value for the output s_o. All other tokens are not changed.

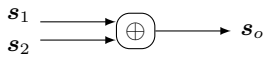


Figure 5.10: The zip process atom. The output s_o is produced by applying values from s_2 to functions in s_1. The tokens of s_1 and s_2 are ordered according to the MoC. The input s_1 is typically an output from another map or zip process atom.

It takes a function f as the first argument, the number of arguments the function f can take defines the maximum of inputs the final process can take. The second argument of the map atom is the first input signal of the final process the map atom is part of. Each token in the input signal, which carries a value, the value is applied to the first argument of the function f. The resulting value is placed in a value carrying token. All other tokens (not carrying values) are copied from input to output. If the function f takes more than one argument, the output signal will carry a sequence of functions taking one less argument than f.

The zip atom is a variant of the map atom. Instead of taking one function, it takes a signal of functions as the first argument. The second argument is carrying the values to be applied to the functions in the same way as for the map atom. Since the zip atom has two input signals it defines the ordering of tokens, e.g. global or partial order.

In order to allow for feedback loops, a delay atom is required. In other words, any loop must contain at least one delay atom. The delay atom must be present in loops for the model to make sense. In other words, a loop must have a finite positive delay, for example described through a synchronisation delay or a time delay. The delay atom (see Figure 5.11) takes an input signal and some notion of delay. The notion of delay depends on the MoC, e.g. timed MoCs use time (and initial value) as delay notion opposed to untimed MoCs which use a number of tokens as delay notion.

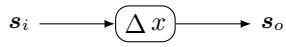


Figure 5.11: The delay process atom. To produce s_o, an initial token is introduced and all tokens in s_i are delayed according to the MoC. The delay notion x may directly specify the initial token, a time delay or something else which best expresses a delay in the given MoC.

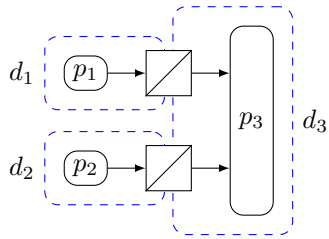


Figure 5.12: Example of domains and domain interfaces.

5.2.3 Domains

In order to combine several MoCs in one model in a formal way, domains are defined. A domain contains a process network implemented in one MoC. In other words, it contains a homogeneous model. A signal from one domain to another is translated using a domain interface. Three domains (d_1, d_2 and d_3) marked by dashed boxes are shown in Figure 5.12.

In Figure 5.12, if domain d_1 and d_2 are implemented in different MoCs, they cannot be merged into one domain. However, if they are expressed in the same MoC, they may be merged into one domain as long as the merged domain still follows the semantics of the MoC. In other words, a merged domain imposes extra restrictions than the two separate domains. In the case of the SY MoC these restrictions are, that the otherwise independent models (p_1 and p_2 in the example) must be fully synchronous if part of the same domain. If p_1 and p_2 are not always intended to be synchronous, they can not be in the same domain. Similarly restrictions apply to other MoCs.

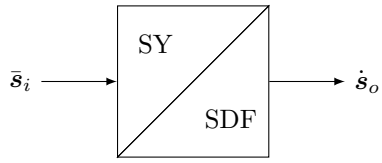


Figure 5.13: Example of a domain interface. The input domain contains the SY MoC and the output domain contains the SDF MoC.

5.2.4 Domain interfaces

A domain interface is a translation of a signal from one domain to another. Each MoC has its own signal type. The four MoCs defined in ForSyDe has the following notation:

- $\bar{\boldsymbol{s}}$ The synchronous (SY) signals are denoted by a *bar*.
- $\dot{\boldsymbol{s}}$ The synchronous data flow (SDF) signals are denoted by a *dot*.
- $\hat{s}$ The discrete event (DE) signals are denoted by a *hat*.
- $\tilde{s}$ The continuous time (CT) signals are denoted by a *tilde*.

The bold face of the notation signifies a sequence of tokens, while the figure above the signal name signifies the MoC it belongs to. The two timed MoCs are not shown in bold, this will be explained in detail later in the individual description of the MoCs.

The generic domain interface is drawn as a rectangle with one diagonal. In the triangles, the source and sink MoC are denoted (see Figure 5.13).

A domain interface must implement the semantics of the two domains it connects. This makes it a challenge to predefine domain interfaces between all MoCs, since adding a new MoC to the framework requires the addition of domain interfaces to all existing MoCs. Another challenge is to describe all possible translations between two MoCs.

A domain interface can only translate one signal. However, some MoCs require extra information (input signals) for the translation. For example explicitly timed MoCs, such as a CT MoC. A domain interface which translates from an untimed MoC to the CT MoC require a time information in addition to the sequence of untimed tokens from the untimed MoC. A domain interface

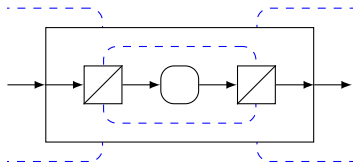


Figure 5.14: Example of a hierarchical domain interface.

which translates in the opposite direction (i.e. from the CT MoC) may require extra time information to sample the continuous signal.

The implementation of a domain interface can be anything from an abstract ideal conversion of signal types to a detailed model. In other words, a domain interface can also be hierarchical of nature. Implemented by more basic domain interfaces and logic connecting them into a network. A hierarchical domain interface is drawn as a rectangle (see the example in Figure 5.14). In essence the difference between a domain interface and a process is: a domain interface has inputs and outputs connected to different domains, where all inputs and outputs of a process are connected to one domain. A reason not to put too much of a model inside domain interfaces, is that it implements multiple domains and their semantics thus may loose the benefit of the formal structure of a single MoC.

An example model with domain interfaces is shown in Figure 5.15. The model specifies a process in the CT MoC which produces a sine signal ($\tilde{s}_i$). The signal is sampled by the analogue to digital converter (ADC) (implemented as a CT/SY domain interface). The time points at which the sampling is done, is specified by $\boldsymbol{s}_t$ which is a chronological sequence of time stamps. This can be implemented by different MoCs. A simple SY model (just a delay) processes the sampled voltage and it is transformed back to the CT MoC with a digital to analogue converter (DAC) (SY/CT domain interface). The DAC combines the values of the samples with the time stamps of $\boldsymbol{s}_t$. The input ($\tilde{s}_i$) and output ($\tilde{s}_o$) CT signals are plotted in the graph.

The example shows how the two domain interfaces implement the same semantics for each of the domains they are connected to. The surrounding domain with the CT MoC is synchronised with the inner domain with the SY MoC on every time point in $\boldsymbol{s}_t$. In other words, every time a synchronisation of

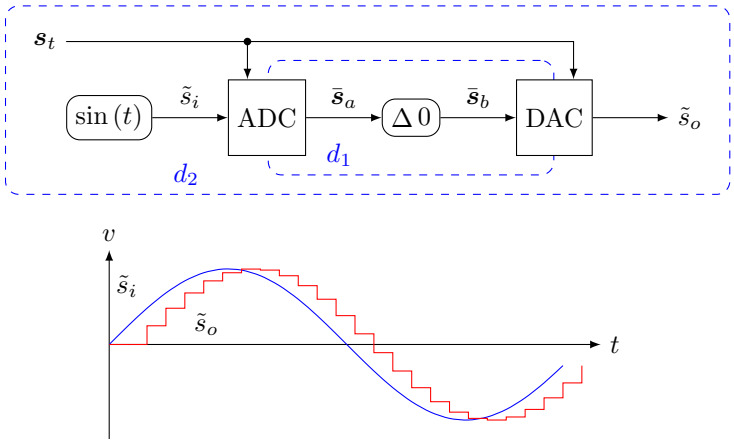


Figure 5.15: Example model with domain interfaces. An input signal $\tilde{s}_i$ is sampled (ADC), processed, and converted back (DAC) to the output signal $\tilde{s}_o$ in the CT MoC. The two analogue signals are plotted in the graph, the blue smooth curve is the input signal ($\tilde{s}_i$) and red step curve is the output signal ($\tilde{s}_o$).

the SY MoC occurs, a value is sampled at the same time as a value is converted to the CT MoC.

5.3 Models of computation

Any MoC can be part of the ForSyDe framework. If domain interfaces can be constructed to other MoCs then a heterogeneous model can be built out of these MoCs. The SYSMODEL project and the company partners believe the four MoCs SY, SDF, DE, and CT to be sufficient for modelling the company use cases. These four MoCs are described and defined as part of the ForSyDe framework in the following sections.

5.3.1 Synchronous MoC

The SY MoC is the simplest to define of the four MoCs in ForSyDe. The SY MoC presented here is derived from the original implementation of ForSyDe [62]

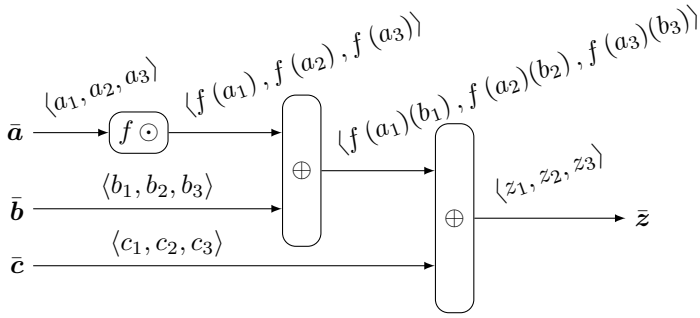


Figure 5.16: Example: How to combine process atoms into models. The mathematical representation of the model is $f \odot \bar{a} \oplus \bar{b} \oplus \bar{c}$. The function f in this example takes precisely three arguments, hence the output signal $\bar{s}_z = \langle z_1, z_2, z_3 \rangle = \langle f(a_1)(b_1)(c_1), f(a_2)(b_2)(c_2), f(a_3)(b_3)(c_3) \rangle$ is the sequence of the fully evaluated functions.

which only contains this single MoC. The main changes from the presentation in [62] are: process atoms have been added, process constructors are derived from process atoms, and the syntax of the MoC is adapted to fit the other MoCs now present in ForSyDe. The Haskell implementation of the SY MoC presented here, is available in Appendix A.1.

The signals in the SY MoC are sequences of tokens with global ordering $\bar{s} = \langle v_1, v_2, \ldots, v_n, \ldots \rangle$. The ordering is defined by the index of the token in the signal (sequence). I.e. tokens with the same index occurs at the same time in all signals. There exists two token types in the SY MoC: the *value token* (v_i) and the *termination token* ($\langle\rangle$).

Since there is a global order of tokens, the processes can synchronise globally by using the index of the tokens of their input signals. Therefore the processes does not need to perform actual synchronisation globally, i.e. there is no communication between processes except the explicit signals. This is illustrated in Figure 5.16, where values with the same index occur at the same time. The first value of each input signal is applied to the function exactly once to produce the output value, after which the second value of each input signal are processed, etc.

The SY MoC provides precisely the three process atoms described in the generic MoC section. The definition of the map atom, shown in (5.2), maps

the function f to each value of the input signal. The definition is divided into two cases, the first (applying value to function and recursing) is chosen if the input signal has at least one value token. The output signal terminates when the input signal terminates. This definition of the map process atom is the same as the one input process constructor of the original ForSyDe.

$$\bar{s}_z = f \odot \bar{s}_x = \begin{cases} f(x) : f \odot \bar{s}'_x & , \bar{s}_x = x : \bar{s}'_x \\ \langle\rangle & , \bar{s}_x = \langle\rangle \end{cases} \tag{5.2}$$

The definition of the zip atom, shown in (5.3), uses the same principle as the map atom to divide the definition into two cases. A recursing and a termination case. The output terminates when either input signal terminates, only if both signals have at least one value token they are combined into an output value. The zip process atom expects the first signal $\bar{s}_f$ to contain functions as values and the second signal $\bar{s}_x$ to contain values matching the type of the functions. This definition of the zip process atom is the same as the two input process constructor of the original ForSyDe fused with the function $\text{apply}(f, x) = f(x)$.

$$\bar{s}_z = \bar{s}_f \oplus \bar{s}_x = \begin{cases} f(x) : \bar{s}'_f \oplus \bar{s}'_x & , \bar{s}_f = f : \bar{s}'_f \wedge \bar{s}_x = x : \bar{s}'_x \\ \langle\rangle & , \bar{s}_f = \langle\rangle \quad \vee \bar{s}_x = \langle\rangle \end{cases} \tag{5.3}$$

The last process atom needed to fully implement the SY MoC in ForSyDe is the delay atom. A delay in the SY MoC is to delay all tokens in a signal by one synchronisation by adding an initial value as the first token. This is simply defined as prefixing the initial token with the input signal, as shown in (5.4). This definition of the delay process atom is the same as the delay process constructor of the original ForSyDe.

$$s_z = \bar{s}_x \,\Delta\, z_0 = z_0 : \bar{s}_x \tag{5.4}$$

Examples of process constructors constructed by process atoms. Four examples are used to illustrate the usage of the process atoms to create other process constructors. The examples are: a simple ALU (plus only), a two input multiplexer, and a Moore and a Mealy state machine.

The ALU is composed of a function (+) and two input signals, put together with one map atom and one zip atom. The composition is graphically represented in Figure 5.17 and as an equation in (5.5). The equation represents a

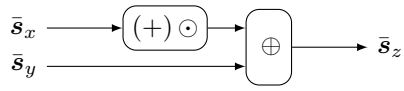


Figure 5.17: Example process constructor. An ALU with a plus operation only.

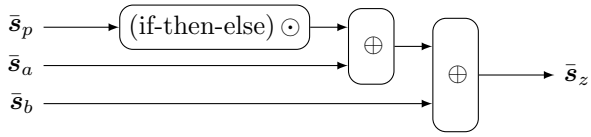


Figure 5.18: A multiplexer process constructor. The function "if-then-else" is defined as: $\lambda p \rightarrow \lambda a \rightarrow \lambda b \rightarrow \text{if}\, p\, \text{then}\, a\, \text{else}\, b$. It selects values from signal $\bar{s}_a$ and $\bar{s}_b$ based on the boolean values of $\bar{s}_p$.

process constructor because the input signals have not been connected/applied yet.

$$P_{+}\left(\bar{s}_x, \bar{s}_y\right) = (+) \odot \bar{s}_x \oplus \bar{s}_y \tag{5.5}$$

The second example implements a multiplexer. It is graphically represented in Figure 5.18 and as an equation in (5.6). The function "if-then-else" is defined as if-then-else $= \lambda p \rightarrow \lambda a \rightarrow \lambda b \rightarrow \text{if}\, p\, \text{then}\, a\, \text{else}\, b$. A process instantiated from this process constructor selects values from signal $\bar{s}_a$ and $\bar{s}_b$ based on the boolean values of $\bar{s}_p$.

$$P_{\text{if-then-else}}\left(\bar{s}_p, \bar{s}_a, \bar{s}_b\right) = (\text{if-then-else}) \odot \bar{s}_p \oplus \bar{s}_a \oplus \bar{s}_b \tag{5.6}$$

The first two examples showed no feedback loops or delay process atoms. A Moore state machine $(S, \Sigma, \Gamma, s_0, f_{\text{next}}, f_{\text{out}})$ is used in the next example. The symbols of the Moore state machine are defined as follows: $\bar{s}_x : [\Sigma]$, $\bar{s} : [S]$, $s_0 : S$, $f_{\text{next}} : S \times \Sigma \rightarrow S$, $f_{\text{out}} : S \rightarrow \Gamma$. The Moore state machine is illustrated in Figure 5.19 and written in ForSyDe notation in (5.7)) and makes use of the delay process atom.

$$P_{\text{Moore}}\left(f_{\text{next}}, f_{\text{out}}, s_0, \bar{s}_x\right) = f_{\text{out}} \odot \bar{s} \quad \text{where} \quad \bar{s} = f_{\text{next}} \odot \bar{s} \oplus \bar{s}_x \,\Delta\, s_0 \tag{5.7}$$

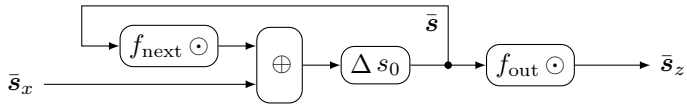


Figure 5.19: A Moore state machine. Arguments of the process constructor are: a next state function f_{next}, an initial state s_0, an output function f_{out}, and the input signal $\bar{s}_x$.

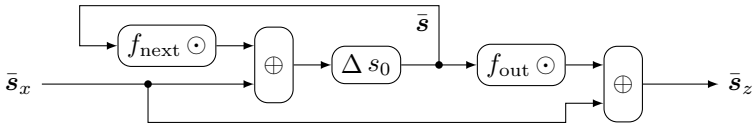


Figure 5.20: A Mealy state machine. Arguments of the process constructor are: a next state function f_{next}, an initial state s_0, an output function f_{out}, and the input signal $\bar{s}_x$.

It is trivial to extend the Moore state machine to a Mealy state machine in the ForSyDe notation (see Figure 5.20 and (5.8)).

$$P_{\text{Mealy}}\left(f_{\text{next}}, f_{\text{out}}, s_0, \bar{s}_x\right) = f_{\text{out}} \odot \bar{s} \oplus \bar{s}_x \quad \text{where} \quad \bar{s} = f_{\text{next}} \odot \bar{s} \oplus \bar{s}_x \,\Delta\, s_0 \tag{5.8}$$

An alternative implementation would be to extend the Moore process constructor with a zip atom at the output (see Figure 5.21). The equation of this state machine is: $P_{\text{Mealy}}\left(f_{\text{next}}, f_{\text{out}}, s_0, \bar{s}_x\right) = P_{\text{Moore}}\left(f_{\text{next}}, f_{\text{out}}, s_0, \bar{s}_x\right) \oplus \bar{s}_x$. Of course the definition of the Mealy state machine based on the Moore state machine requires one to instantiate the Mealy with an output function which takes at least two arguments. Thus the output of the Moore state machine will be a signal of functions.

Example of a simulation of a Moore state machine. The Moore state machine process constructor shown in Figure 5.19 and (5.7) is instantiated with the functions f_{next} and f_{out} and the initial value s_0 as shown below. The state machine instantiation implements a gate, where you have to pay (Coin) to pass

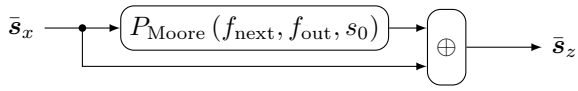


Figure 5.21: Alternate definition of the Mealy state machine.

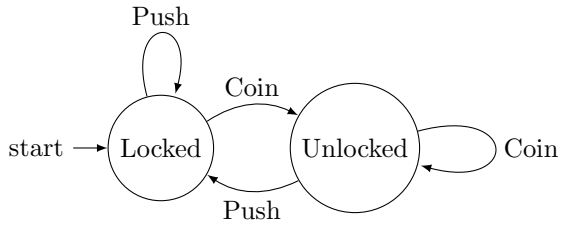


Figure 5.22: A Moore state machine of a gate, where you have to pay (Coin) to pass through (Push).

the gate (Push).

$$f_{\text{next}}(s,i) = \begin{cases} \text{Locked} & , i = \text{Push} \\ \text{Unlocked} & , i = \text{Coin} \end{cases} \tag{5.9}$$

$$f_{\text{out}}(s) = \begin{cases} \text{Enter} & , s = \text{Unlocked} \\ \text{Pay} & , s = \text{Locked} \end{cases} \tag{5.10}$$

$$s_0 = \text{Locked} \tag{5.11}$$

The state diagram is shown in Figure 5.22 and an example simulation is shown below, with both input, state and output signals.

$\bar{s}_x$	$=\langle$Push,	Coin,	Push,	Coin,	Coin,	Push,	Push $\rangle$
$\bar{s}$	$=\langle$Locked,	Locked,	Unlocked,	Locked,	Unlocked,	Unlocked,	Locked $\rangle$
$\bar{s}_z$	$=\langle$Pay,	Pay,	Enter,	Pay,	Enter,	Enter,	Pay $\rangle$

5.3.2 Synchronous data flow MoC

The SDF MoC [39] is the other untimed MoC present in ForSyDe. It is closely related to the SY MoC. The difference between the two MoCs are the number

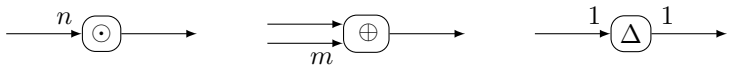


Figure 5.23: Annotation of token consumption rates of process atoms. The map process atom consumes n tokens per execution. The zip process atom consumes 1 token on the first input and m tokens on the second input per execution. The delay process atom consumes 1 token per execution. All three process atoms produce 1 token per execution.

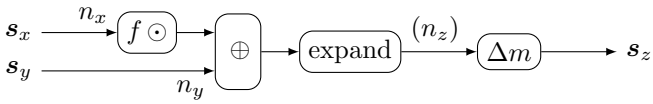


Figure 5.24: A generic two input process constructor.

of tokens consumed and produced per execution of a process, the SDF MoC does not require all processes to fire simultaneously thus there is only a partial order of tokens between signals. The Haskell implementation of the SDF MoC presented here, is available in Appendix A.2.

Both the MoCs consume/produce a static number of tokens per process execution, but the SY MoC requires this to be one for all inputs and outputs. The SDF MoC allows for any positive number. The Figure 5.23 shows the typical annotation of consumption and production of tokens per execution of a process. The map and zip atoms specifies the number of tokens consumed per input, but the number of outputs are not specified with these three process atoms. The delay atom is hard-wired to consume and produce one token per execution.

To be able to specify the number of output tokens produced per execution of a process, a new process atom must be defined for the SDF MoC. It is called *expand*. An example of its use is shown in Figure 5.24 and (5.12) together with all the other process atoms. The example process constructor will only accept functions (f) of precisely two arguments. It is required that a SDF process contains exactly one map atom $n-1$ zip atoms and one expand atom, where n is the number of inputs of the process. Furthermore the function f must have precisely n arguments. This is a restriction of expressiveness compared to the SY MoC.

$$P\left(f, n_x, n_y, m, \dot{\boldsymbol{s}}_x, \dot{\boldsymbol{s}}_y\right) = \text{expand}\left(f \odot \left(n_x, \dot{\boldsymbol{s}}_x\right) \oplus \left(n_y, \dot{\boldsymbol{s}}_y\right)\right) \Delta\, m \tag{5.12}$$

The number of produced tokens (n_z) is defined not by the expand atom itself, but by the result of the function f. The challenge is that the map and zip atoms can only produce one token per execution, and thus the resulting set of tokens that are produced per execution of the final process is packed in one token in the internal signals. The job of the expand atom is to expand these packed tokens into the output signal. This implies that all SDF processes must always have an expand atom after its map and zip atoms.

Comparing the implementation of the map atom of SY and SDF, the map atom takes an extra argument (the number of consumed tokens n_x) in SDF. This argument is used to *split* the input signal into two chunks. A set of consumed tokens and the rest of the input signal, if there where enough tokens to meet the required amount n_x the process fires (and recurses). If the input signal terminates and it is not possible to acquire enough tokens to fire, the output signal terminates. Thus some input tokens may be discarded when the output signal terminates.

$$\dot{\boldsymbol{s}}_z = f \odot \left(n_x, \dot{\boldsymbol{s}}_x\right) = \begin{cases} f\left(x\right) : f \odot \left(n_x, \dot{\boldsymbol{s}}'_x\right) & , \left(x, \dot{\boldsymbol{s}}'_x\right) = \text{split}\left(n_x, \dot{\boldsymbol{s}}_x\right) \\ & \quad \wedge\, \text{length}\left(x\right) = n_x \\ \langle\rangle & \quad , \text{otherwise} \end{cases} \tag{5.13}$$

The zip process atom has, like the map process atom, an extra argument (n_x) to describe the number of consumed tokens per execution on the second input signal. Since any zip process atom is part of the same process as the process atom connected to its first input, the zip process atom must fire once per token in the first input signal. The zip process atom terminates when either of the input signals terminate, otherwise it processes the required number tokens of the input signals and repeats.

$$\dot{\boldsymbol{s}}_z = \dot{\boldsymbol{s}}_f \oplus \dot{\boldsymbol{s}}_x = \begin{cases} f\left(x\right) : \dot{\boldsymbol{s}}'_f \oplus \left(n_x, \dot{\boldsymbol{s}}'_x\right) & , \dot{\boldsymbol{s}}_f = f : \dot{\boldsymbol{s}}'_f \\ & \quad \wedge\, \left(x, \dot{\boldsymbol{s}}'_x\right) = \text{split}\left(n_x, \dot{\boldsymbol{s}}_x\right) \\ & \quad \wedge\, \text{length}\left(x\right) = n_x \\ \langle\rangle & \quad , \text{otherwise} \end{cases} \tag{5.14}$$

The expand process atom is unique to the SDF MoC (of the MoCs currently in ForSyDe). It takes, per execution, one token containing multiple values and

outputs multiple tokens containing one value each. It terminates when the input signal terminates.

$$\dot{\boldsymbol{s}}_z = \text{expand}\left(\dot{\boldsymbol{s}}_x\right) = \begin{cases} \text{foldr}\left((:), \text{expand}\left(\dot{\boldsymbol{s}}'_x\right), x\right) & , \dot{\boldsymbol{s}}_x = x : \dot{\boldsymbol{s}}'_x \\ \langle\rangle & , \dot{\boldsymbol{s}}_x = \langle\rangle \end{cases} \tag{5.15}$$

The delay process atom is extended to take a set of initial values, all prefixed to the input signal. Otherwise it is the same as the delay process atom in the SY MoC.

$$\dot{\boldsymbol{s}}_z = \dot{\boldsymbol{s}}_x \,\Delta\, z_0 = \text{foldr}\left((:), \dot{\boldsymbol{s}}_x, z_0\right) \tag{5.16}$$

5.3.3 Discrete event MoC

The DE MoC is an explicitly timed MoC. Firstly the signals of the DE MoC differs from the two previous MoCs. All tokens in a signal carries a time including when the signal terminates. Furthermore, a signal also carries an initial value, since two different signals may not be defined from the same starting time. A signal in the DE MoC is constructed as a tuple of an initial value and a sequence of tokens carrying timed events and the time of termination of the signal. Each timed event is a tuple of time and value.

$$\hat{s} = (x_0, \boldsymbol{s}) \quad \text{where} \quad \boldsymbol{s} = \langle (t_1, v_1), (t_2, v_2), \ldots, (t_n, v_n), \ldots \rangle_{t_{\text{termination}}} \tag{5.17}$$

A signal must contain a sequence of events with strictly increasing time tags. The termination time $t_{\text{termination}}$ must be greater than the time of the last event in the sequence. If the signal is infinite it does not terminate, thus the termination time is infinite. The Haskell implementation of the DE MoC presented here, is available in Appendix A.3.

The map process atom of the DE MoC looks almost like the map process atom of the SY MoC. However, since the signal is a tuple of an initial value and the sequence of tokens, the initial value is handled separately first in (5.18). The second part in (5.19) resembles the SY MoC and works in the same way. If the token of the input signal is an event with value, the value is applied to the function to produce the output value. All other tokens are copied to the output. Time is not changed in this process atom.

$$\hat{s}_z = f \odot \hat{s}_x = f \odot (x_0, s_x) = (f(x_0), \text{lift}(f, s_x)) \tag{5.18}$$

$$\text{lift}(f, s_x) = \begin{cases} (t, f(x)) : \text{lift}(f, s'_x) & , s_x = x : s'_x \\ \langle\rangle_t & , s_x = \langle\rangle_t \end{cases} \tag{5.19}$$

The equation for the zip process atom looks somewhat more complex than the two previous MoCs, but it still follows the principle that if either of the input signal terminates, so does the output signal. Again the zip process atom first treats the initial value of the signal in (5.20) and then the sequence of tokens in (5.21).

The handling of the tokens is divided into eight cases, covering alignment of the incoming tokens by time and token type. The three first cases are like the the recursion case of the other MoCs in the sense that both signals have an event as the next token. The three cases represents the timed ordering of these events, either the event of the first signal happens first, they happen at the same time, or the event of the second signal happens first.

The first three cases do not cover all cases of recursion. The other two cases of recursion (case 4 and 5) represents cases where one signal terminates *later* than the event of the other signal. Hence the event is handled and the process recurses.

The last three cases cover the termination. The first two termination cases (case 6 and 7) are in essence the opposite of the last two recursion cases. Here the termination of one signal happens *before or at the same time* as the event of the other signal. Thus the process terminates. The last case is when both signal terminates, and the process terminates at the earliest time of the two terminating signals.

$$\hat{s}_z = \hat{s}_f \oplus \hat{s}_x = (f_0, s_f) \oplus (x_0, s_x) = (f_0(x_0), \text{bind}(f_0, s_f, x_0, s_x)) \tag{5.20}$$

$$
\begin{aligned}
&\text{bind}\,(f_0, s_f, x_0, s_x) \\
&= \begin{cases}
(t_f, f\,(x_0)) : \text{bind}\left(f, s'_f, x_0, s_x\right) & , t_f < t_x \wedge s_f = (t_f, f) : s'_f \\
& \qquad \wedge\ s_x = (t_x, x) : s'_x \\
(t_f, f\,(x)) : \text{bind}\left(f, s'_f, x, s'_x\right) & , t_f = t_x \wedge s_f = (t_f, f) : s'_f \\
& \qquad \wedge\ s_x = (t_x, x) : s'_x \\
(t_x, f_0\,(x)) : \text{bind}\,(f_0, s_f, x, s'_x) & , t_f > t_x \wedge s_f = (t_f, f) : s'_f \\
& \qquad \wedge\ s_x = (t_x, x) : s'_x \\
(t_f, f\,(x_0)) : \text{bind}\left(f, s'_f, x_0, s_x\right) & , t_f < t_x \wedge s_f = (t_f, f) : s'_f \\
& \qquad \wedge\ s_x = \langle\rangle_{t_x} \\
(t_x, f_0\,(x)) : \text{bind}\,(f_0, s_f, x, s'_x) & , t_f > t_x \wedge s_f = \langle\rangle_{t_f} \\
& \qquad \wedge\ s_x = (t_x, x) : s'_x \\
\langle\rangle_{t_f} & , t_f \leq t_x \wedge s_f = \langle\rangle_{t_f} \\
& \qquad \wedge\ s_x = (t_x, x) : s'_x \\
\langle\rangle_{t_x} & , t_f \geq t_x \wedge s_f = (t_f, f) : s'_f \\
& \qquad \wedge\ s_x = \langle\rangle_{t_x} \\
\langle\rangle_{\min(t_f, t_x)} & , \quad s_f = \langle\rangle_{t_f} \\
& \qquad \wedge\ s_x = \langle\rangle_{t_x}
\end{cases}
\end{aligned}
\tag{5.21}
$$

The delay process atom is a bit different from the SY and SDF versions. It takes a tuple of an initial value and a time delay. The initial value of the delay process atom overrides the initial value of the input signal.

The initial value of the delay process atom is the initial value of the output signal, furthermore it is used as the value of an initial event at time zero (see (5.22)). The tokens of the input token sequence are all processed by adding the time delay to the token time (see (5.23)). All tokens are delayed, not only the tokens carrying a value. This is of course also the case for the delay process in the SY and SDF MoCs, but here it is done explicitly.

$$
\hat{s}_z = \hat{s}_x\,\Delta(z_0, T) = (x_0, s_x)\,\Delta(z_0, T) = (z_0, (0, z_0) : \text{delay}\,(s_x, T)) \tag{5.22}
$$

$$
\text{delay}\,(s_x, T) = \begin{cases}
(t + T, x) & , s_x = (t, x) : s'_x \\
\langle\rangle_{t+T} & , s_x = \langle\rangle_t
\end{cases} \tag{5.23}
$$

Example simulations of models. First a model of a logic inverter where the output of the inverter is feed back to its input. The inverter is modelled as a not-process and a delay. The model is depicted in Figure 5.25. The model has

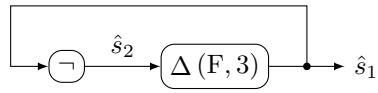


Figure 5.25: Unstable one-inverter loop. The equation of the model is: $\hat{s}_1 = \hat{s}_2\,\Delta(\mathrm{F}, 3)$ where $\hat{s}_2 = (\neg) \odot \hat{s}_1$.

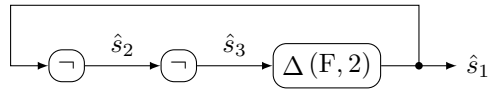


Figure 5.26: Stable two-inverter loop. The equation of the model is: $\hat{s}_1 = \hat{s}_3\,\Delta(\mathrm{F}, 3)$ where $\hat{s}_2 = (\neg) \odot \hat{s}_1$ and $\hat{s}_3 = (\neg) \odot \hat{s}_2$.

no inputs, so the result of a simulation is governed by the initial state of the inverter (the delay process) and its delay. Using an initial value of False (F) and a delay of 3 time units, the following simulation result is obtained:

$$\begin{aligned}
\hat{s}_1 &= \big(\mathrm{F}, \langle(0, \mathrm{F}), \quad (3, \mathrm{T}), \quad (6, \mathrm{F}), \quad (9, \mathrm{T}), \quad \ldots\big) \\
\hat{s}_2 &= \big(\mathrm{T}, \langle(0, \mathrm{T}), \quad (3, \mathrm{F}), \quad (6, \mathrm{T}), \quad (9, \mathrm{F}), \quad \ldots\big)
\end{aligned}$$

The termination (right angular bracket with termination time as index) of the signal sequences is not written, as it never occur.

The second model shows a stable two inverter loop shown in Figure 5.26. The only difference between the first and the second model is an extra not-process in the loop. Only one delay process is present, and defines the total delay of the two inverters. If one delay process was put after each not-process, the model would not exhibit a stable state if the two delay processes had the same initial value.

As in the first model, the resulting simulation trace is determined by the initial value and delay of the delay process. Even though the system is stable, a simulation produces an infinite sequence of events as shown below:

$$\begin{aligned}
\hat{s}_1 &= \big(F, \langle(0, \mathrm{F}), \quad (2, \mathrm{F}), \quad (4, \mathrm{F}), \quad (6, \mathrm{F}), \quad \ldots\big) \\
\hat{s}_2 &= \big(T, \langle(0, \mathrm{T}), \quad (2, \mathrm{T}), \quad (4, \mathrm{T}), \quad (6, \mathrm{T}), \quad \ldots\big) \\
\hat{s}_3 &= \big(F, \langle(0, \mathrm{F}), \quad (2, \mathrm{F}), \quad (4, \mathrm{F}), \quad (6, \mathrm{F}), \quad \ldots\big)
\end{aligned}$$

A more optimal simulation result would only contain one event in each signal and a termination at infinity to illustrate a signal which never changes.

$$\hat{s}_1 = \left(F, \langle (0, \mathrm{F}) \rangle_\infty\right)$$
$$\hat{s}_2 = \left(T, \langle (0, \mathrm{T}) \rangle_\infty\right)$$
$$\hat{s}_3 = \left(F, \langle (0, \mathrm{F}) \rangle_\infty\right)$$

There exists several different theories for optimising DE simulations as exemplified above. One way is to use one big event queue, in which all processes commits produced events to. With such an event queue, it is possible to remove redundant events without any further problems. If the event queue at any point is empty, the simulation will never change the state of any signals again. Such simulators are, however, inherently single threaded. Processing multiple events in parallel, when using one event queue, is one possibility to overcome the single threaded nature of event queues. However, this approach relies on automatic detection of event that can be processed in parallel and corrections if processed events were not yet ready for execution.

In the case of ForSyDe, the processes are defined to only communicate through signals. I.e. they do not synchronise by any other means than signals. Therefore redundant events cannot always be removed without challenges.

There exists two methods of optimising DE simulations with processes defined like in ForSyDe. A conservative and an opportunistic approach [29]. The opportunistic approach tries to perform calculations before a process may have all required inputs ready. If the missing inputs never arrive (i.e. they were removed due to redundancy) the calculation is correct. However, if the missing input arrives later, the calculation must be rolled back (annihilated) and the correct calculation performed instead.

The conservative approach tries to detect when it is safe to remove redundant events. The challenge is that if all events in a feed back loop of the model are removed, the simulations stops (deadlocks). The conservative approach tries to avoid these deadlocks by allowing some redundancy or producing alternative tokens to signify absent events.

The advantage of the conservative and the opportunistic approaches are that it is possible to exploit the concurrency of the model itself for parallel execution of the simulation. I.e. the processes of a model can be arbitrarily chosen to run in parallel, without changing the behaviour/simulation result of the model. The definition of ForSyDe presented in this chapter does, however, not support the opportunistic approach because two consecutive events in a signal can never be processed at the same time in any process.

The research performed as part of this thesis includes conservative optimisations of the DE MoC. These optimisations can produce the same optimal simulation results as presented above of the example model shown in Figure 5.26 and a few other models. The implementation has not been proven to work for all DE models. The implementation of these optimisations can be found on the enclosed disc.

5.3.4 Continuous time MoC

The CT MoC is an explicitly timed MoC like the DE MoC. The definition of the CT MoC presented here is not derived from any other work. The Haskell implementation of the CT MoC presented here, is available in Appendix A.4.

The basic idea of a CT simulator is: a signal is defined as a sequence of time intervals. In each time interval a continuous function describe the analogue signal. The time intervals of signal make up a continuous time sequence from signal start to termination. Thus the time intervals start at the simulation time t_i of the event token (t_i, f_i) and ends just before the simulation time t_{i+1} of the next event (t_{i+1}, f_{i+1}). I.e. the time interval of event i is $[t_i, t_{i+1}[$. The time interval of the last event in the signal will end at the time of the termination instead.

The processes operate on the functions and produce new functions. The processes may change the time intervals, for example by dividing a time interval into two.

This idea can be expressed directly in the DE MoC. However, it requires the definition of functions that can handle functions as values of the events. For example, the simple plus process $(+) \odot \hat{s}_1 \oplus \hat{s}_2$ would become $(\lambda x \to \lambda y \to \lambda u \to x(u) + y(u)) \odot \hat{s}_3 \oplus \hat{s}_4$. The signals $\hat{s}_3$ and $\hat{s}_4$ would then contain functions of time as values, e.g. $\lambda u \to sin(u)$.

The time variable u represents modelled real time, as opposed to the simulation time t. The difference is that the simulation time t must have an integral type, where u could be a real type. An example could be that a simulated time unit is equivalent to one millisecond of modelled real time.

Instead of explicitly handling the time, the modifications of the plus operator shown above can be embedded into the map and zip process atoms, thus creating a dedicated CT MoC. The function which is embedded into the map process atom is $\lambda f \to \lambda u \to f(u)$. The resulting definitions are shown in (5.24) and (5.25).

$$\tilde{s}_z = f \odot \tilde{s}_x = f \odot (x_0, s_x) = (\lambda u \to f\,(x_0\,(u))\,, \mathrm{lift}\,(f, s_x)) \tag{5.24}$$

$$\mathrm{lift}\,(f, s_x) = \begin{cases} (t, \lambda u \to f\,(x\,(u))) : \mathrm{lift}\,(f, s'_x)\ , & s_x = x : s'_x \\ \langle\rangle_t & , \ s_x = \langle\rangle_t \end{cases} \tag{5.25}$$

The zip process atom is adapted similarly from the DE MoC, by embedding the function $\lambda f \to \lambda x \to \lambda u \to f(u)(x(u))$. The resulting definitions are shown in (5.26) and (5.27). No other change has been made to the zip process atom equations of the DE MoC than to embed the above mentioned function. This implies that recursion and termination of the CT zip process is the same as the DE zip process.

$$\begin{aligned} \tilde{s}_z = \tilde{s}_f \oplus \tilde{s}_x &= (f_0, s_f) \oplus (x_0, s_x) \\ &= (\lambda u \to f_0\,(x_0\,(u))\,, \mathrm{bind}\,(f_0, s_f, x_0, s_x)) \end{aligned} \tag{5.26}$$

$$\mathrm{bind}\,(f_0, s_f, x_0, s_x)$$

$$= \begin{cases} (t_f, \lambda u \to f\,(u, x_0\,(u))) : \mathrm{bind}\left(f, s'_f, x_0, s_x\right) & , \ t_f < t_x \\ \qquad\qquad s_f = (t_f, f) : s'_f \wedge s_x = (t_x, x) : s'_x & \\ (t_f, \lambda u \to f\,(u, x\,(u))) : \mathrm{bind}\left(f, s'_f, x, s'_x\right) & , \ t_f = t_x \\ \qquad\qquad s_f = (t_f, f) : s'_f \wedge s_x = (t_x, x) : s'_x & \\ (t_x, \lambda u \to f_0\,(u, x\,(u))) : \mathrm{bind}\,(f_0, s_f, x, s'_x) & , \ t_f > t_x \\ \qquad\qquad s_f = (t_f, f) : s'_f \wedge s_x = (t_x, x) : s'_x & \\ (t_f, \lambda u \to f\,(u, x_0\,(u))) : \mathrm{bind}\left(f, s'_f, x_0, s_x\right) & , \ t_f < t_x \\ \qquad\qquad s_f = (t_f, f) : s'_f \wedge s_x = \langle\rangle_{t_x} & \\ (t_x, \lambda u \to f_0\,(u, x\,(u))) : \mathrm{bind}\,(f_0, s_f, x, s'_x) & , \ t_f > t_x \\ \qquad\qquad s_f = \langle\rangle_{t_f} \wedge s_x = (t_x, x) : s'_x & \\ \langle\rangle_{t_f} & , \ t_f \leq t_x \\ \qquad\qquad s_f = \langle\rangle_{t_f} \wedge s_x = (t_x, x) : s'_x & \\ \langle\rangle_{t_x} & , \ t_f \geq t_x \\ \qquad\qquad s_f = (t_f, f) : s'_f \wedge s_x = \langle\rangle_{t_x} & \\ \langle\rangle_{\min(t_f, t_x)} & , \ s_f = \langle\rangle_{t_f} \\ & \qquad \wedge\ s_x = \langle\rangle_{t_x} \end{cases} \tag{5.27}$$

The delay process atom of the DE MoC can be used without any change in the CT MoC. The equations are repeated (with CT signal notation) in (5.28) and (5.29).

$$\tilde{s}_z = \tilde{s}_x \, \Delta(z_0, T) = (x_0, s_x) \, \Delta(z_0, T) = (z_0, (0, z_0) : \text{delay}\,(s_x, T)) \tag{5.28}$$

$$\text{delay}\,(s_x, T) = \begin{cases} (t+T, x) & , \; s_x = (t, x) : s'_x \\ \langle\rangle_{t+T} & , \; s_x = \langle\rangle_t \end{cases} \tag{5.29}$$

Since the CT MoC is based on the semantics of the DE MoC, it is obvious to assume that the same optimisation techniques also apply. However, there is one challenge with this definition of the CT MoC. The challenge is to determine redundancy when comparing two functions. There exist no natural method to compare two arbitrary functions, to check if they are equal for the time interval where they are to be equal.

5.4 Structured domain interfaces

The domain interfaces have been described in generic terms without any example of implementation. The implementation of domain interfaces depends on how the two domains are meant to interact. It may not be possible to describe all possible domain interfaces, like it is for the processes of each of the presented MoC, that are built from a small set of process atoms. However, in this section a set of generic domain interface atoms are defined, which cover at least the trivial domain interfaces. These domain interface atoms can be combined with the previously described MoCs to form more complex domain interfaces.

The domain interface atoms are formed by taking the domain interface (not the hierarchical domain interface) and splitting it at the diagonal (see Figure 5.27). The idea is that the two halves can be matched with the counterparts of the other MoCs to form a basic domain interface between two different models expressed in different MoCs. The signals $\boldsymbol{o}$ and $\boldsymbol{i}$ have the same semantics as the SY signals in between the two domain interface atoms. Connecting the signals $\boldsymbol{o}$ and $\boldsymbol{i}$ completes the basic domain interface.

5.4.1 Domain interfaces for the untimed MoCs

The domain interface atoms of the SY (see Figure 5.28) and SDF MoCs (see Figure 5.29) have the same semantics but with different signal types. The definition of these atoms is merely a change of the type of the signal. The left atom in each figure is an interface *from* the domain and the right is an interface *to* the domain.

Figure 5.27: Splitting a basic domain interface in two halves.

Figure 5.28: Domain interface atoms of the SY MoC.

Figure 5.29: Domain interface atoms of the SDF MoC.

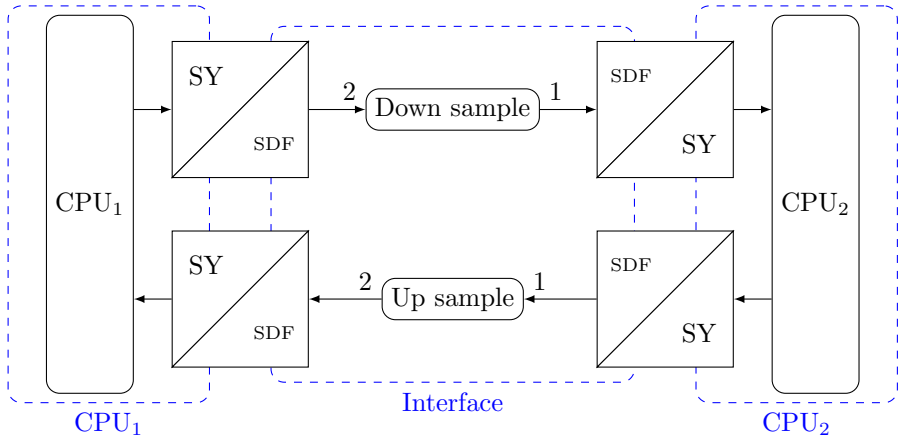


Figure 5.30: Example of two communicating CPUs where CPU_1 runs at double the frequency of CPU_2.

An example of the use of domain interface atoms with SY and SDF MoCs is shown in Figure 5.30. The example illustrates two processor cores (CPUs) that can communicate through an interface. The interface describes a 2:1 relation in clock frequency, i.e. CPU_1 runs twice as fast as CPU_2.

5.4.2 Domain interfaces for the timed MoCs

The domain interface atoms of the DE (see Figure 5.31) and CT MoC (see Figure 5.32) are a bit more complex than the untimed counterparts. Multiple domain interfaces between the same two domains must be synchronised by another means than the token index as is the case for the untimed versions. The timed domain interface atoms must be synchronised by time.

The domain interface atoms for the DE MoC have extra time input ($\boldsymbol{t}_i$) and output ($\boldsymbol{t}_o$) signals. There are two definitions of the domain interface atoms from the DE domain. The one that has no time input (the leftmost of Figure 5.31) takes all tokens with a value and split the time-value tuple in two and puts the time in the output time signal ($\boldsymbol{t}_o$) and the value in $\boldsymbol{o}$. Furthermore the time signal also carries the information of when the input signal ($\hat{s}_i$) terminates. Such splitting can be expressed with a SY model where time is just one value element of each tuple as shown in (5.30).

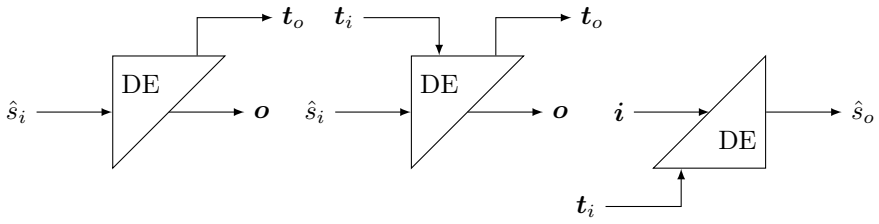


Figure 5.31: Domain interface atoms of the DE MoC.

$$P_{\text{split}}\left(\bar{\boldsymbol{s}}\right) = \left(\left(\lambda(t,v) \to t\right) \odot \bar{\boldsymbol{s}}, \left(\lambda(t,v) \to v\right) \odot \bar{\boldsymbol{s}}\right) \tag{5.30}$$

The domain interface atom to the DE domain (the rightmost) has the opposite operation of the previously described domain interface atom. It takes one time token and one value token from each input ($\boldsymbol{t}_i$ and $\boldsymbol{i}$) and outputs an event with these in $\hat{s}_o$. The merging can also be expressed with a SY model as shown in (5.31).

$$P_{\text{merge}}\left(\bar{\boldsymbol{s}}_t, \bar{\boldsymbol{s}}_v\right) = \left(\lambda t \to \lambda v \to (t,v)\right) \odot \bar{\boldsymbol{s}}_t \oplus \bar{\boldsymbol{s}}_v \tag{5.31}$$

The last domain interface atom (the middle one in Figure 5.31) is an extension of the leftmost domain interface atom. It allows sampling of the DE input signal ($\hat{s}_i$) at time points specified by the input time sequence ($\boldsymbol{t}_i$). The output time sequence ($\boldsymbol{t}_o$) is identical to the input time sequence and the output value sequence (o) contains the sampled values.

The CT domain interface atoms (shown in Figure 5.32) are the same as for DE, but with one extra domain interface atom drawn as a parallelogram. The extra domain interface atom allows for sampling the CT functions such that the values of the functions at the specified time points are obtained. The difference between this domain interface atom unique to the CT MoC and the bottom left one in Figure 5.32 is that the latter returns the functions as values and the former applies time to the function to evaluate it at that time. The behaviour of the sampling domain interface atom can be expressed by the following SY model:

$$P_{\text{sample}}(\bar{\boldsymbol{s}}_t, \bar{\boldsymbol{s}}_v, f) = \left(\lambda t \to \lambda v \to v(f(t))\right) \odot \bar{\boldsymbol{s}}_t \oplus \bar{\boldsymbol{s}}_v \tag{5.32}$$

The purpose of the function f is to transform the simulation time t to the modelled real time u, i.e. $f : t \to u$.

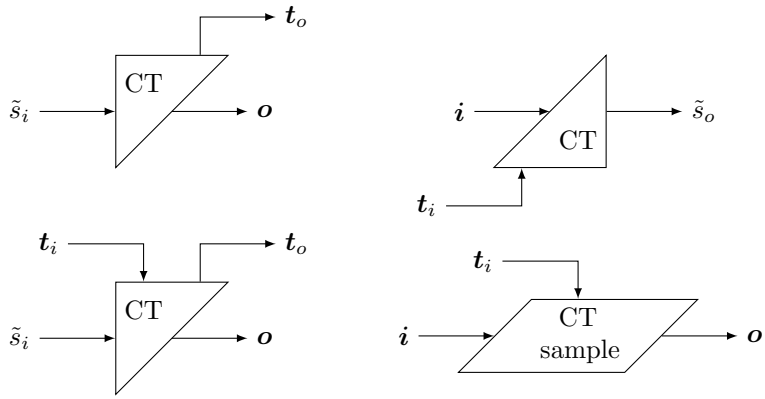


Figure 5.32: Domain interface atoms of the CT MoC.

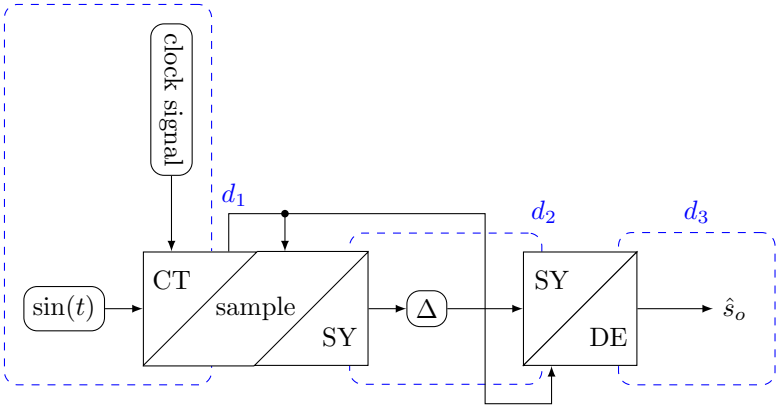


Figure 5.33: Example of both timed and untimed MoCs in the same heterogeneous model.

An example model, which combines both timed and untimed MoCs is shown in Figure 5.33. The model is divided into three domains. The first domain (d_1) contains a model expressed in the CT MoC with a source process that produces a sinusoidal signal. This signal is sampled at time points specified by the clock signal process.

The domain interface from CT to SY represents an ADC. The sampled values are processed by the model in domain d_2 expressed in the SY MoC. Hereafter, a simple model of a DAC is used to convert the signal to the third domain (d_3) which contains a model expressed in the DE MoC. This example is the same as in Figure 5.15 except that the DAC converts to a DE MoC instead of a CT MoC.

5.5 Summary

Two classes of systems are defined, static and dynamic systems. The terms static and dynamic refer to the state space of the system, i.e. whether the system has a static (finite) or dynamic (non-finite) state space. Models of static and dynamic systems are then divided into two classes: homogeneous and heterogeneous, depending on the number of MoCs used to express a system.

The mathematical definition of ForSyDe is presented in generic terms and in details for the four MoCs (SY, SDF, DE, and CT) currently implemented in ForSyDe. The CT MoC is not based on classic CT simulators, whether this is efficient or practical is unknown. It is included to have an implementation of the CT MoC in Haskell-ForSyDe and to illustrate that it is possible to embed other MoCs in the ones presented here.

The idea of embedding MoCs in other MoCs is not limited to the example of the CT MoC. An entire model can be embedded into processes also. In other words, the function of a process can implement a model, e.g. a SY state machine. In real world models, the functions of processes are usually implemented in some programming language (e.g. Haskell, C, C++, etc.). The programming language is itself a MoC which is used to express the model/function.

The concept of formal modelling requires one to express the important aspects of models using the formal structure of MoCs. Embedding a model as the function in a process breaks this formalism.

To help avoiding the use of embedding models inside processes, domain interfaces are defined to formally combine models expressed in different MoCs. A set of structured domain interfaces are presented, to define a formal method to construct domain interfaces. However, these structured domain interfaces

may not be able to express all possible domain interfaces.

ForSyDe as presented in this chapter, does not yet express dynamic systems well with the provided MoCs. However, other MoCs can be added to ForSyDe. Such MoCs does not need to adhere to any special semantics. However, they must provide domain interfaces to the other MoCs, if they are to express heterogeneous models. They must also support simulation of the models.

Chapter 6

Static systems

In this chapter, a system level design methodology is presented, which allows designers to model and analyse their systems from the early stages of the design process until final implementation. The design methodology targets heterogeneous embedded systems and is based on the ForSyDe framework presented in Chapter 5. ForSyDe is available under the open source approach, which allows small and medium enterprise (SME) to get easy access to advanced modelling capabilities and tools. This chapter gives an introduction to the design methodology through the system level modelling of a simple industrial use case.

6.1 Introduction

Industry is facing a crisis in the design of complex hardware/software systems. Due to the increasing complexity, the gap between the generation of a product idea and the realization of a working system is expanding rapidly. To manage complexity and to shorten design cycles, industry is forced to look at system level languages towards specification and design. Such languages allow the designer to capture the system functionality from the very early stages in the design process and to use this system model as a basis for evaluating design decisions and for a stepwise refinement of the system specification into a final implementation.

Figure 6.1 shows the classical design flow. The initial requirement specification, which captures both the functional and non-functional properties of the system, is partitioned into a software and a hardware specification. This partition is usually based on the experience of the designers and on availability of

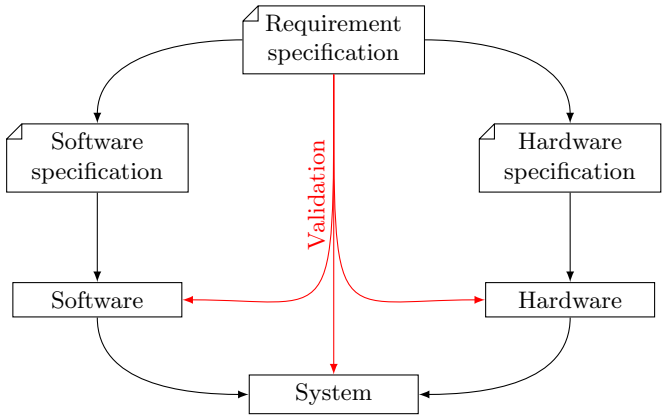


Figure 6.1: Current design methodology.

existing hardware platforms. Although this may be a good starting point, the lack of being able to explore different design alternatives in a systematic way, seriously impacts the quality and competitiveness of the resulting solution.

In this chapter, a system level modelling approach is presented, aimed at capturing the early stages of the design, allowing the designer to explore trade-offs and support design decisions. The proposed modelling approach is captured in a system design framework (system functionality framework (SFF)), which has been developed as part of the SYSMODEL[1] project. The aim of this project has been to support the competitiveness of small and medium enterprises (SMEs). The general availability of the design framework is facilitated by the open source approach, where all tools are made available free of charge. Figure 6.2 shows how the proposed modelling approach extends the classical design flow with system level modelling. The initial requirement specification is partitioned into a functional specification and a non-functional specification. The functional specification is first translated into a suitable model of the application. Relevant non-functional properties, such as latency and power consumption, are used to guide this translation. Likewise, the non-functional properties are used to guide the selection of an appropriate execution platform, expressed as a platform model. Finally, the application model is mapped onto the platform model in order to form a model of the integrated system. As there are many ways to

[1]Funded by ARTEMIS JU

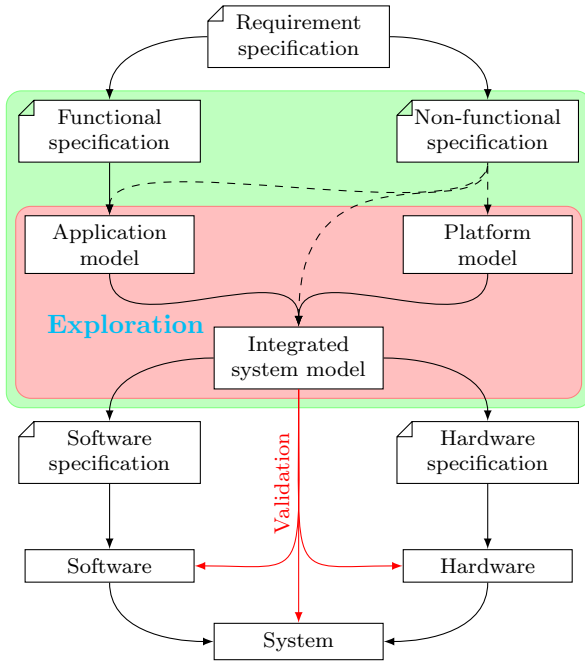


Figure 6.2: Design methodology using system modelling. The green area symbolises the modelling part of the design process. The red area is the design exploration part.

achieve this mapping, there is a need for being able to perform an exploration of the design space.

The proposed SFF is based on ForSyDe (formal system design) [23], a formal design methodology which allows several models of computation to be integrated in a single heterogeneous model, in order to capture and model different types of components, such as analogue, digital and software components, and to describe a system at different stages in the design process. A formal modelling approach with clear semantics, makes it possible to formally reason about properties of the design, such as risks, price, power, and timing, already in the early stages of the design process.

We illustrate the design methodology outlined in Figure 6.2 through a system level design of a simple use case, a hearing aid calibration device. As the

calibration device is a medical device, it has to apply to medical safety regulations [12, 21], which is always a challenge. One of the major advantages of the device as compared to competitors, is its small size and ease of use, which adds to the challenges. In order to work correctly according to safety regulations and medical specifications for hearing aid calibration devices, strict timing requirements are given. Now one of the key challenges in the early stage of the design process, where the complete system is being designed, is to ensure that a given application design when executed on the selected platform will always meet these timing requirements.

In the following, we first describe the use case and how the initial requirement specification may be transformed into an application model, and how this model may be bound to a platform model, in order to form an integrated system model. We explain how the models may be validated through simulation. After the use case, we turn to the modelling framework and outline the basics of the ForSyDe model. Finally, we give a summary and some concluding remarks.

6.2 Industry case

Throughout this chapter, a hearing aid calibration device is used as a use case for our proposed design methodology. The use case is provided by the Danish company Auditdata. Figure 6.3 shows the calibration device in context, i.e., the physical setup, where the calibration device controls sound generation and samples the sound in the ear through two microphones, one in front of the hearing aid device and one inside the ear right behind the hearing aid. This is called the real ear measurement (REM). The sampled sound signals are processed in the calibration device and send to the doctors PC via an USB interface for display. Figure 6.3 only shows the setup for one ear, however, the calibration device is able to handle both ears at the same time, with a total of sampling four microphones (using the same speaker for both ears).

6.2.1 Functional specification

The desired functionality of the product is to produce sound streams that are played in a loud speaker and record by up to four sound streams simultaneously that are then displayed on a computer screen as histograms in real time.

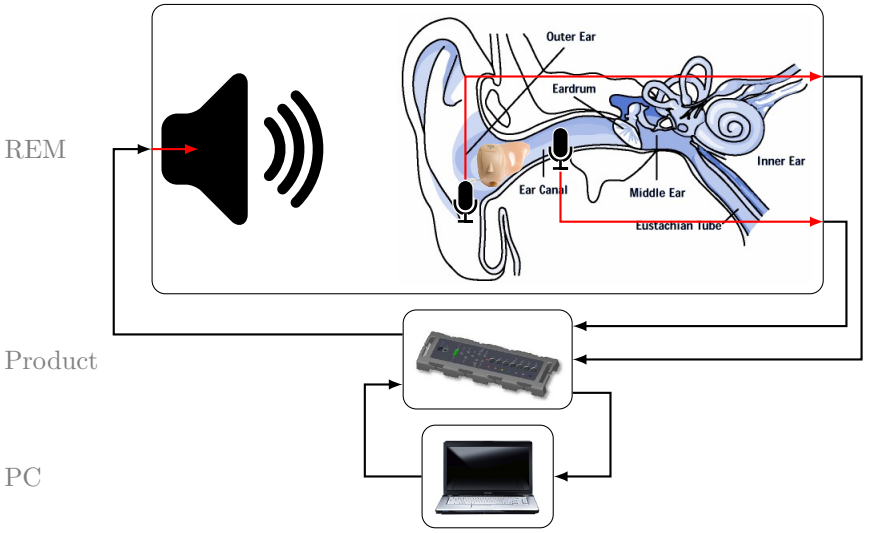


Figure 6.3: The problem that needs to be solved. A patient (the ear) needs to have a new hearing aid adjusted. The doctor places two microphones on the patient, one inside the ear canal and one just outside. In the room there is a loudspeaker. This setup is called a Real Ear Measurement (REM).

6.2.2 Non-functional specification

As the calibration process involves the patient to be able to relate visual and audio input, the calibration device has certain timing requirements. Furthermore, medical safety regulations require the signal processing to be done under real-time requirements and to deliver a certain accuracy. These timing and accuracy requirements challenges the design of the calibration device together with a wish to produce a low cost and low power device that is connected to the PC through a USB interface.

6.3 Application model

The functional part of the requirement specification can be expressed as an application model (block Application model in Figure 6.2). The benefits of such a model are that it can be used to validate the application behaviour, for instance by simulation of the model. Depending on the formalism used to describe the model, formal verification of system properties may also be a possibility.

6.3.1 ForSyDe model

An application model of the use case is shown in Figure 6.4. This particular model implements the behaviour of producing one continuous sound stream for a speaker while continuously sampling two sound streams. Each of these sampled sound streams are transformed into histograms by a fast Fourier transformation (FFT)[2]. This represents one scenario/configuration of the product.

In order to simulate this model, a standard FFT implementation can be used for each of the FFT blocks. This will produce the same functional behaviour as a version which has been optimised with respect to a given platform and non functional requirements.

We use a synchronous data flow (SDF) [40] as the underlining model of computation (MoC). SDF models the data flow between processes and is hence, well suited for modelling streaming applications. Each process consumes a static number of tokens on each input and produces a static number of tokens on each output. A process will execute when sufficient tokens are ready on all inputs. The MoC is without any notion of time. SDF can relatively easy be analysed

[2] A fast Fourier transformation is an efficient algorithm to compute the discrete Fourier transformation.

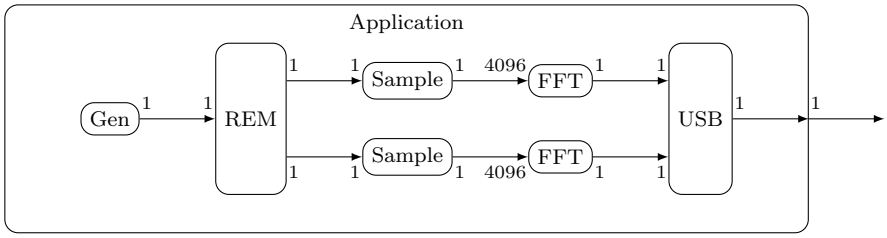


Figure 6.4: Simplified behavioural model of the product. This is a setup where two channels are actively being used.

for required buffer sizes between processes. In the use case in Figure 6.4 the calculation of buffer sizes is not complex (maximum 4096 tokens before each FFT), but in more complex situations, e.g., where interaction would happen between the two streams of sampling followed by FFT, the analysis could quickly become much more complex.

6.3.2 Simulation

The application model can be simulated with very simple definitions of each process. This particular application model is simple enough that the behaviour is obvious. However, in more complex applications, only the behaviour of sub parts may be obvious while the global behaviour might not. Simulation can then be used to validate the behaviour.

An example of a simulation for this use case is to feed the application model with sine waves in the "Gen" process and verify that one frequency is dominating the output. Due to the discrete calculation of the Fourier transformation, the tone might not always be transformed into only one frequency, but will cover a small band. A result from a simulation of the application model is shown in Figure 6.5.

The "REM" process can be used to model different changes to the sound as it travels through the air and ear channel.

6.3.3 Verification

In broad terms the application model can be used to estimate non-functional properties, verify that non-functional requirements are met, and explore be-

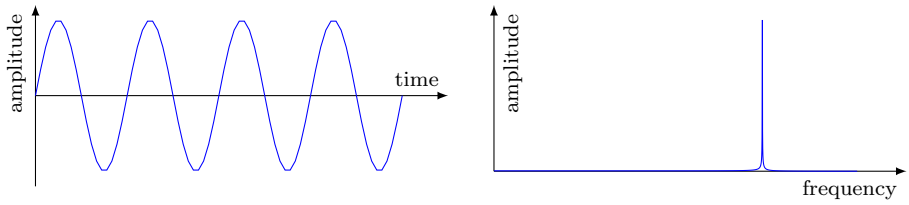


Figure 6.5: Example of simulation input and result.

haviour to achieve certain non-functional properties. Such properties could be buffer sizes needed for communication between processes or a static schedule of the processes. The verification techniques used to determine the properties depends on the MoCs used to model the application. It is possible to combine multiple MoCs in an application model, but some verification techniques will then only apply to certain parts of the application model.

Since the application model of the use case is modelled using only one MoC, specifically the untimed MoC SDF, it is possible to perform verification on the entire structure of the model. One such verification is a buffer analysis, i.e. how much memory is (at most) required to contain all the data which is streamed through the device while operating. As an example, the difference between a 16MB and a 64MB memory solution is approximately a factor 4 in energy consumption (respectively 0.125W and 0.5W). The power used by the memory can therefore be a significant amount of the total power budget of the 2.5W provided by a USB port at maximum.

6.4 Platform model

The SFF can be used to represent details of all components and interconnections of the platform, i.e. a detailed representation of the hardware circuits. However, at the early stages of the design process, we are more interested in having a high level system model, which captures how the application interacts with the platform. In other words, the platform model provides the execution details, such as schedule and duration of the various application processes. In this context, the platform model defines the execution of the application model. This may be done by connecting the application model to the platform through control signals. Control signals from the platform model to the application

model releases a process in the application model, and the opposite returns the control to the platform.

A platform model may start at a relative high level of abstraction with very few details. During the design process, it may be gradually refined with more and more details. In the case of a platform containing a single processor core, the application model may be refined to a sequence of function calls. This can be emulated with the abstract application model by sequencing the control output from the application model to the next control input to the application model. A more complex platform model which includes some model of an operating system, may make more elaborate decisions on which application process to release, effectively modelling the behaviour of a dynamic execution sequence, such as a fixed priority based real-time operation system. The execution semantics of SDF, ensures that the application processes will not execute until both data and control input are ready. Since the application model is without any notion of time and the timing of the execution of application processes are platform dependent, the platform is annotated with execution time of each application process.

In our use case, the platform is given and based on a single processor core. This core executes all digital signal processing and controls the A/D and D/A converters. The interface to the PC is done through a USB chip, which only has a single packet buffer for receiving data. The platform may also have to capture the behaviour of an operating system which in our use case is described as round robin cooperative multitasking. In our case it simplifies into a static series of function calls to each application process.

As the application is presented as four streams (the pairs of sample and FFT processes), it may benefit from a platform supporting multiple cores. Hence, it would be interesting to explore alternative platforms, such as a platform with two processor cores or even four processor cores, each servicing a single stream. Although this may seem obvious, a challenge is that the operating system or the USB process system may be more complex, since synchronisation between the processor cores has to take place when collecting data for transport to the PC. Only a careful exploration of the design space will reveal the best trade-off.

6.5 Integrated system model

The integrated system model is the combination of the application model and the platform model. The integration of the application model is performed by adding extra control dependencies to each process, such that the platform

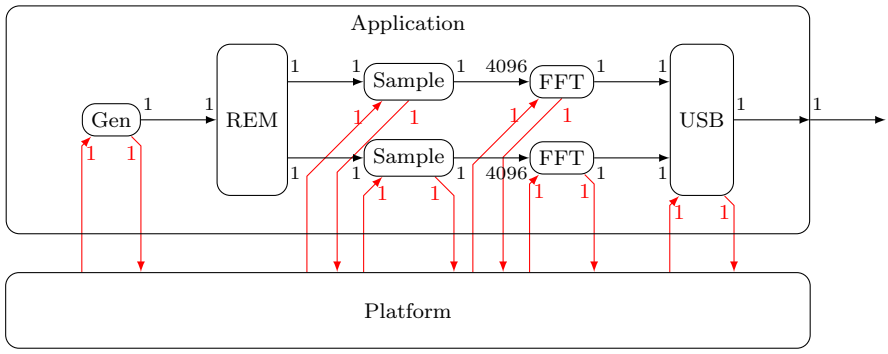


Figure 6.6: Integrated system model, the combination of the application model and the platform model.

activates the process at the appropriate point in time. The estimated execution time of each application process is modelled through these control dependencies. These control dependencies are shown in Figure 6.6 as red arrows.

The platform controls the execution of each process in the application model by sending a release for execution through the input control dependencies. The output control dependencies to an application returns the control to the platform.

An important property of this modelling approach is that the application model with annotated control input/output dependencies, is independent of the actual platform. This means that the application model and the platform model are kept separate in the integrated system model, effectively applying a separation of concerns.

Furthermore, this allows us to perform what-if analysis of possible design choices. We may explore possible changes in the mapping, the platform or even in the application itself. Such explorations could be based on simulation or on an analytical approach which would allow for fast automated design space exploration. Hence, the aim of the exploration is to find the best solution which fulfils all non functional requirements. However, It is also possible to evaluate other relevant parameters of the design. Parameters which are not strictly being expressed as requirements, but are secondary optimization goals, such as the sensitivity of the proposed design. I.e. evaluating the amount of slack in the design which can be used to adjust the final solution in order to compensate for

inaccurate estimates made in the early stages. This may lead to better solutions than those obtained from applying very conservative rules, which often leads to over designed systems.

6.5.1 Simulation

Simulating the integrated system model, can provide important insight into the design. We may be able to decide if the platform supports the application with the given requirements on execution time, power consumption, etc. Another important aspect of simulating a model of the system, is that we can easily get access to information, such as signals, components, variables and software blocks, which may be inaccessible in the final implementation. Further, we may use this information to easily infer start and end times of application processes as well as the pre-emption of these.

6.6 Summary

We have presented a system level design methodology based on the ForSyDe formal modelling framework, which allows designers to model and analyse both functional and non-functional properties of their system at the very early stages of the design process. The methodology has been illustrated through the modelling and analysis of a rather simple industry case of a hearing aid calibration device. Early decision support is a very critical factor in handling the design of complex hardware/software systems and to achieve high quality products in short design cycles. We have illustrated how the separation of application and platform in the integrated system model may provide easy explorations of different platforms or mappings. Finally, having a complete and formal system model, may lead to an easier handling of outsourcing sub-parts of the system, as the requirements of the sub-parts may be extracted directly from the system model.

Chapter 7

Dynamic systems

Formal modelling dynamic systems has not been successful so far. Simulations of dynamic systems have been illustrated in Chapter 2 through the routing algorithm for wireless sensor networks (WSNs). To address the challenges of expressing a dynamic system in the UPPAAL framework as presented in Chapter 4, a formal framework for modelling WSNs is presented in this chapter. The individual node of the WSN is modelled as static subsystems similar to the one presented in Chapter 6. The WSN framework is not directly using the theory of ForSyDe as presented in Chapter 5 for defining the dynamic modelling framework, however, this framework can be implemented in ForSyDe as a separate model of computation (MoC).

7.1 Introduction

A WSN is a distributed network, where a large number of computational components (also referred to as "sensor nodes" or simply "nodes") are deployed in a physical environment. Each component collects information about and offers services to its environment, e.g. environmental monitoring and control, healthcare monitoring and traffic control, to name a few. The collected information is processed either at the component, in the network or at a remote location (e.g. the base station), or in any combination of these. WSNs are typically required to run unattended for very long periods of time, often several years, only powered by standard batteries. This makes energy-awareness a particular important issue when designing WSNs.

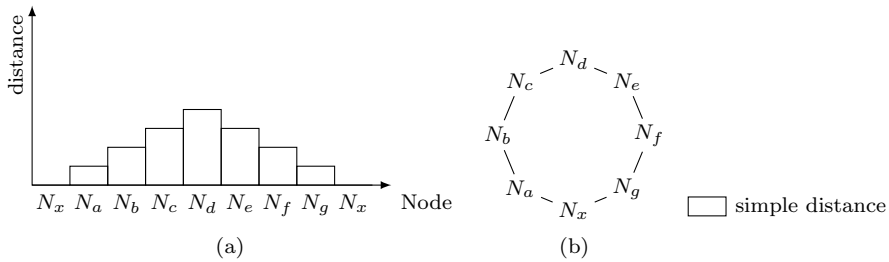


Figure 7.1: An example network displaying the shortest distance to the base station. (a) shows each node's distance to the base station while (b) shows the placement of each node.

In a WSN there are two major sources of energy usage:

- Operation of a node, which includes sampling, storing and possibly processing of sensor data.
- Routing data in the network, which includes sending data sampled by the node or receiving and resending data from other nodes in the network.

Traditionally, WSN nodes have been designed as ultra low-power devices, i.e., low-power design techniques have been applied in order to achieve nodes that use very little power when operated and even less when being inactive or idle. By adjusting the duty-cycle of nodes, it is possible to ensure long periods of idle time, effectively reducing the required energy.

At the network-level nodes are equipped with low-power, low-range radios in order to use little energy, resulting in multi-hop networks in which data has to be carefully routed. A classical technique has been to find the shortest path from any node in the network to the base station and hence, ensuring a minimum amount of energy to route data. The shortest path is illustrated in Figure 7.1. Figure 7.1b shows the circular network layout, where the base station is labelled N_x. Figure 7.1a is a bar-chart showing the distance (y-axis) from a node to the base station, the x-axis is an unfolding of the circular network, placing the base station, with a distance of zero, at both ends.

The routing pattern of a node in this network is based upon the distance from a node (e.g. N_c) and its neighbours (N_b and N_d) to the base station. The node N_c will route to the neighbour with the shortest distance to the base station (in this case N_b). In practice, nodes close to the base station (e.g. N_a and N_g) will

be activated much more frequently than those far away from the base station, resulting in a relative short lifetime of the network. To address this, energy efficient algorithms, such as [16, 22, 71], have been proposed. The aim of these approaches is to increase the lifetime of the network by distributing the data to *several* neighbours in order to minimize the energy consumption of nodes on the shortest path. However, these approaches do not consider the residual energy in the batteries. The energy-aware algorithms, such as [22, 24, 45, 51, 66, 67, 74, 76], are all measuring the residual battery energy and are extending the routing algorithms to take into account the actual available energy, under the assumption that the battery energy is monotonically decreasing.

With the advances in energy harvesting technologies, energy harvesting is an attractive new source of energy to power the individual nodes of a WSN. Not only is it possible to extend the lifetime of the WSN, it may eventually be possible to run them without batteries. However, this will require that the WSN system is carefully designed to effectively use adaptive energy management, and hence, adds to the complexity of the problem. One of the key challenges is that the amount of energy being harvested over a period of time is highly unpredictable. Consider an energy harvester based on solar cells, the amount of energy being harvested, not only depends on the efficiency of the solar cell technology, but also on the time of day, local weather conditions (e.g., clouds), shadows from buildings, trees, etc.. For these conditions, the energy-aware algorithms presented above, cannot be used as they assume residual battery energy to be monotonically decreasing. A few energy harvesting aware algorithms have been proposed to address these issues, such as [28, 38, 44, 72, 73, 75]. They do not make the assumption of monotonically decreasing residual battery energy, and hence, can account for both discharging and charging the battery. Furthermore, they may estimate the future harvested energy in order to improve performance. However, these routing algorithms make certain assumptions that are not valid for multi-hop networks.

The clustering routing approach used in [28, 72] assumes that all nodes are able to reach the base station directly. A partial energy harvesting ability is used in [73], where excess harvested energy can not be stored and the nodes are only battery powered during night. The algorithm in [38] is an offline algorithm, it assumes that the amount of harvest-able energy can be predicted before deployment, which is not a realistic assumption for most networks. The algorithm in [75] requires that each node has knowledge of its geographic position. Global knowledge is assumed in [44].

Techniques for managing harvested energy in WSNs have been proposed, such as [9, 18, 34, 35, 55, 68]. These are focussing on local energy management. In

[35] they also propose a method to synchronise this power management between nodes in the network to reduce latency on routing messages to the base station. They do, however, not consider dynamic routes as such. An interesting energy harvesting aware multi-hop routing algorithm is the REAR algorithm by [24]. It is based on finding two routes from a source to a sink (i.e. the base station), a primary and a backup route. The primary route reserves an amount of energy in each node along the path and the backup route is selected to be as disjunct from the primary route as possible. The backup route does not reserve energy along its path. If the primary route is broken (e.g. due to power loss at some node) the backup route is used until a new primary and backup route has been built from scratch by the algorithm. An attempt to define a mathematical framework for energy aware routing in multi-hop WSNs is proposed by [44]. The framework can handle renewable energy sources of nodes. The advantage of this framework is that WSNs can be analysed analytically, however the algorithm relies on the ideal, but highly unrealistic assumption, that changes in nodal energy levels are broadcasted instantaneously to all other nodes. The problem with this approach is that it assumes global knowledge of the network.

The aim of this chapter is to propose a modelling framework which can be used to study energy harvesting aware routing in WSNs. The capabilities and efficiency of the modelling framework will be illustrated through the modelling and simulation of a distributed energy harvesting aware routing protocol, distributed energy harvesting aware routing (DEHAR) by [31]. In Section 2 a generic modelling framework which can be used to model and analyse a broad range of energy harvesting aware WSNs, is developed. In particular, a conceptual basis as well as an operational basis for such networks are developed. Section 3 shows the adequacy of the modelling framework by giving very natural descriptions and explanations of two energy harvesting based networks: DEHAR [31] and directed diffusion (DD) [27]. The main ideas behind routing in these networks are explained in terms of the simple network in Figure 7.1. Properties of energy harvesting aware networks are analysed in Section 4 using simulation results for DEHAR and DD. These results validate that energy harvesting awareness increases the energy level in nodes, and hence, keeps nodes (which otherwise would die) alive, in the sense that a complete drain of energy in critical nodes can be prevented, or at least postponed. Finally, Section 5 contains a brief summary and concluding remarks.

7.2 A generic modelling framework

The purpose of this section is to present a generic modelling framework which can be used to study energy-aware routing in a WSN, where the nodes of the network have an energy harvesting capability. In the next section instantiations of this generic model will be presented and experimental results through simulations are presented in Section 7.4.

The main idea of establishing a generic framework is to have a conceptual as well as a tool-based foundation for studying a broad range of wireless sensor networks with similar characteristics. In the following we will assume that

- sensor nodes have an energy-harvesting device,
- sensor nodes are using radio-based communication, consisting of a transmitter and a receiver,
- sensor nodes are inexpensive devices with limited computational power, and
- the routing in the network adapts to dynamic changes of the available energy in the individual nodes, i.e. the routing is energy aware.

On the other hand, we will not make any particular assumptions about the kind of sensors which are used to monitor the environment.

These assumptions have consequences concerning the concepts which should be reflected in the modelling framework, in particular, concerning the components of a node. Some consequences are:

- A node may only be able to have a direct communication with a small subset of the other nodes, called its neighbours, due to the range of the radio communication.
- A node needs information about neighbour nodes reflecting their current energy levels in order to support energy-aware routing.
- A node can make immediate changes to its own state; but it can only affect the state of other nodes by use of radio communication.
- The processing in the computational units as well as the sensing, receiving and transmitting of data are energy consuming processes.

These assumptions and consequences fit a broad range of WSNs.

7.2.1 The components of a node

A node consists of five physical components:

- An *energy harvester* which can collect energy from the environment. It could be by the use of a solar panel – but the concrete energy source and harvesting device are not important in the generic setting.
- A *sensor* which is used to monitor the environment. There may be several sensors in a physical node; but we will not be concerned about concrete kinds in the generic setting and will (for simplicity) assume that one generic sensor can capture the main characteristics of a broad range of physical sensors.
- A *receiver* which is used to get messages from the network.
- A *transmitter* which is used to send messages to the network.
- A *computational unit* which is used to treat sensor data, to implement the energy-aware routing algorithm, and to manage the receiving and sending of messages in the network.

The model should capture that use of the sensor, receiver, transmitter and computational unit consume energy and that the only supply of energy comes from the nodes' energy harvesters. It is therefore a delicate matter to design an energy-aware routing algorithm because a risk is that the energy required by executing the algorithm may exceed the gain by using it.

A consequence of this is that exact energy information cannot be maintained between nodes because it requires too much communication in the network as that would imply that too much energy is spent on this administrative issue compared to the harvested energy and the energy used for transmitting sensor-observations from the nodes to the base station.

7.2.2 The identity of a node

We shall assume that each node has a unique identification which is taken from a set Id of identifiers.

7.2.3 The state of a node

The *state* of a node is partitioned into a *computational state* and a *physical state*. The physical state contains a model of the real energy level in the node as well

as a model of the dynamics of energy devices, like, for example, a capacitor. The computational state contains an approximation of the physical energy model, including at least an approximation of the energy level. The computational state also contains routing information and an abstract view of the energy level in neighbour nodes. Furthermore, the computational state could contain information needed in the processing of observations, but we will not go into details about that part of the computational state here, as we will focus on energy harvesting and energy-aware routing.

We shall assume the existence of the following sets (or types):

- PhysicalState – which models the real physical states of the node,
- Energy – which models energy levels,
- ComputationalState – which models the state in the computational unit in a node, including a model of the view of the environment (especially the neighbours) and information about the energy model and the processing of observations, and
- AbstractState – which models the abstract view of a computational state. An abstract state is intended to give a condensed version of a computational state and it can be communicated to neighbour nodes and used for energy-aware routing. It is introduced since it is too energy consuming to communicate complete state information to neighbours when radio communication is used.

The state parts of a node may change during operation. The concrete changes will not be described in the generic framework, where it is just assumed that they can be achieved using the functions specified in Figure 7.2. Notice that a node can change its own state only.

The intuition behind each function is given below. A concrete definition (or implementation) of the functions must be given in an instantiation of the generic model.

- consistent?(cs) is a predicate which is true if the computational state cs is *consistent*. Since neighbour and energy information, which are used to guide the routing, are changing dynamically, a node may end up in a situation where no neighbour seems feasible as the next destination on the route to the base station. Such a situation is called inconsistent, and the predicate consistent?(cs) can test for the occurrences of such situations.

Sets: PhysicalState, ComputationalState, AbstractState, and Energy

Operations:

consistent?	: ComputationalState → {true, false}
abstractView	: ComputationalState → AbstractState
updateEnergyState	: ComputationalState × Energy → ComputationalState
updateNeighbourView	: ComputationalState × Id × AbstractState → ComputationalState
updateRoutingState	: ComputationalState → ComputationalState
transmitChange?	: ComputationalState × ComputationalState → {true, false}
next	: ComputationalState → Id

Figure 7.2: A signature for operations on the computational state

- abstractView(cs) gives the abstract view of the computational state cs. This abstract view constitutes the part of the state which is communicated to neighbours.
- updateEnergyState(cs, e) gives the computational state obtained from cs by incorporation of the actual energy level e. The resulting computational state may be inconsistent.
- updateNeighbourView(cs, id, as) gives the computational state obtained from cs by updating the neighbour knowledge so that as becomes the abstract state of the neighbour node N_{id}. The resulting computational state may be inconsistent.
- updateRoutingState(cs) gives the computational state obtained from cs by updating the routing information on the basis of the energy and neighbour knowledge in cs so that the resulting state is consistent.
- transmitChange?(cs, cs') is a predicate which is true if the difference between the two computational states are so significant that the abstract view of the "new state" should be communicated to the neighbours.
- next(cs) gives, on the basis of the computational state cs, the identifier of the "best" neighbour to which observations should be transmitted.

7.2.4 The computation costs

Each of the above seven functions in Figure 7.2 are executed on the computational unit of a node. Such an execution will consume energy and cause a

change of the physical state. For simplicity, we will assume that the cost of executing the predicates consistent? and transmitChange? can be neglected or rather included in other functions, since they always incur the same energy cost in these functions. These functions are specified in Figure 7.3.

costAbstractView	:	PhysicalState → PhysicalState
costUpdateEnergyState	:	PhysicalState → PhysicalState
costUpdateNeighbourView	:	PhysicalState → PhysicalState
costUpdateRoutingState	:	PhysicalState → PhysicalState
costNext	:	PhysicalState → PhysicalState

The costs of the predicates consistent? and transmitChange? are assumed negligible.

Figure 7.3: A signature for cost operations on the computational state

For simplicity it is assumed that execution of each of the five functions has a constant energy consumption, so that all functions have the type PhysicalState → PhysicalState. It is easy to make this model more fine grained. For example, if the cost of executing abstractView depends on the computational state to which it is applied, then the corresponding cost function should have the type: PhysicalState × ComputationalState → PhysicalState. This level of detail is, however, not necessary to demonstrate the main principles of the framework.

7.2.5 Input events of a node

The computational unit in a node can react to *events* originating from the energy observations on the physical state, e.g. due to the harvesting device, the sensor and the receiver. There are two energy related events, where one is concerned with the change of the physical state while the other is concerned with reading the energy level in the node. The rationale for having two events rather than a "combined" one is that the change of the physical state is a cheap operation which does not involve a reading nor any other kind of computation, whereas a reading of the energy level consumes some energy.

A sensor recording results in an observation o belonging to a set Observation of observations. An observation could be temperature measurement, a traffic

observation or an observation of a bird – but the concrete kind is of no importance in this generic part of the framework.

The events are described as follows:

- readEnergyEvent(e, ps), where $e \in$ Energy and $ps \in$ PhysicalState, which is an event signalling a reading e of the energy level in the node and a resulting physical state ps, which incorporates that the reading actually consumes some energy.
- physicalStateEvent(ps), where $ps \in$ PhysicalState is a new physical state. This event occurs when a change in the physical state is recorded. This change may, for example, be due to energy harvesting, due to a drop in energy level, or due to some other change which could be the elapse of time.
- observationEvent(o, ps), where $o \in$ Observation is a recorded sensor observation and $ps \in$ PhysicalState is a physical state which incorporates the energy consumption due to the activation of the sensor.
- receiveEvent(m, ps), where $ps \in$ PhysicalState and $m \in$ Message, which could be an observation to be transmitted to the base station or a message describing the state of a neighbour node. Further details are given below. The receiver maintains a *queue of messages*. When it records a new message, that message is put into the queue. The event receiveEvent(m, ps) is offered when m is the front element in the queue. Reacting to this event will remove m from the queue and a new receive event will be offered as long as there are messages in the queue. It is unspecified in the generic setting whether there is a bound on the size of the queue.

7.2.6 Input messages

A node has a queue of messages received from the network. There are two kinds of messages:

- *Observation Messages* of the form obsMsg(dst, o), where dst is the identity of the next destination of the observation $o \in$ Observation on the route to the base station.
- *Neighbour Messages* of the form neighbourMsg(src, as), where src is the identity of the source, i.e. the node which has sent this message, and $as \in$ AbstractState is the contents of the message in the form of an abstract state.

Let Message denote the set of all messages, i.e. observation and neighbour messages.

7.2.7 Output messages and communication

A node N_{id} can use the transmitter to broadcast a message $m \in$ Message to the network using the command $\texttt{send}_{id}(m)$. Intuitively, nodes which are within the range of the transmitter will receive this message and this may depend on the strength of the signal, it may depend on geographical positions, or on a variety of other parameters.

A model for sending and receiving messages could include a *global trace* of the messages send by nodes, a *local trace* of messages received by the individual nodes, and a description of a *medium*, that determines which nodes can receive messages sent by a node N_{id} on the basis of the current state of the network and on the basis of the various parameters, for example, concerning geographical positions of the nodes. In instances of the generic model, such a medium must be described. In this chapter we will not be formal about network communication. A formal model of communication along the lines sketched above can be found in [53, 59].

7.2.8 The cost of sending messages

Sending a message consumes energy which is reflected in a change of the physical state of a node. To capture this a function

$$\text{costSend} : \text{PhysicalState} \times \text{Message} \to \text{PhysicalState}$$

can compute a new physical state on the basis of the current one and a broadcasted message.

7.2.9 An operational model of a node

During its lifetime, a node can change between two main *phases*: *idle* and *treat message*.

- The node is basically inactive in the idle phase waiting for some event to happen. It processes an incoming event and makes a phase transition.
- The node treats a single message in the treat message phase and after that it makes a transition to the idle phase.

$\text{Idle}_{id}(cs, ps) =$
  `wait`
    $\text{physicalStateEvent}(ps') \rightarrow \text{Idle}_{id}(cs, ps')$
    $\text{readEnergyEvent}(e, ps') \rightarrow$
      `let` $cs' = \text{updateRoutingState}(\text{updateEnergyState}(cs, e))$
      `let` $ps'' = \text{costUpdateEnergyState}(\text{costUpdateRoutingState}(ps'))$
      `if` $\text{transmitChange?}(cs, cs')$
      `then let` $m = \text{neighbourMsg}(id, \text{abstractView}(cs'))$
        $\texttt{send}_{id}(m); \text{Idle}_{id}(cs', \text{costSend}(\text{costAbstractView}(ps''), m))$
      `else` $\text{Idle}_{id}(cs, ps'')$
    $\text{observationEvent}(o, ps') \rightarrow$
      `let` $dst = \text{next}(cs)$
      `let` $m = \text{obsMsg}(dst, o)$
      $\texttt{send}_{id}(m); \text{Idle}_{id}(cs, \text{costSend}(\text{costNext}(ps'), m))$
    $\text{receiveEvent}(m, ps') \rightarrow \text{TreatMsg}_{id}(m, cs, ps')$

Figure 7.4: The Idle Phase

Each phase is parametrised by the computational state cs and the physical state ps. The state changes and phase transitions for the idle phase are given in Figure 7.4. The node stays inactive in the idle phase until an event occurs.

- A physical-state event leads to a change of physical state while staying in the idle phase.
- A read-energy event leads to an update of the energy and routing parts of the computational state, and the physical state is updated by incorporation of the corresponding costs. If the changes of the computational state are insignificant then these changes are ignored (so that the nodes have a consistent knowledge of each other) and just the physical state is changed. Otherwise, the abstract view of the new computational state is computed and send to the neighbours, and both the computational and the physical states are changed.
- An observation event leads to a computation of the next node (destination) to which the observation should be transmitted on the route to the base station, and a corresponding observation message is sent. The physical

state is changed with the cost of computing the destinations and the cost of sending a message while staying in the idle phase.

- A receive event indicates a pending message in the queue. That message is treated by a transition to the treat message phase.

Notice that all phase transitions from the idle phase preserve the consistency of the computational state. The only non-trivial transition to check is that from $\text{Idle}_{id}(cs, ps)$ to $\text{Idle}_{id}(cs', \text{costSend}(\text{costAbstractView}(ps''), m)$. The consistency of cs' follows since $cs' = \text{updateRoutingState}(\text{updateEnergyState}(cs, e))$ and updateRoutingState is expected to return a consistent computational state, at least under the assumption that cs is consistent.

$$
\begin{array}{l}
\text{TreatMsg}_{id}(m, cs, ps) = \\
\quad \texttt{case}\ m\ \texttt{of} \\
\qquad \text{obsMsg}(dst, o)\ \rightarrow \\
\qquad\quad \texttt{if}\ id = dst \\
\qquad\quad \texttt{then let}\ dst' = \text{next}(cs) \\
\qquad\qquad\quad \texttt{let}\ m' = \text{obsMsg}(dst', o) \\
\qquad\qquad\quad \texttt{send}_{id}(m'); \text{Idle}_{id}(cs, \text{costSend}(\text{costNext}(ps), m) \\
\qquad\quad \texttt{else}\ \text{Idle}_{id}(cs, ps)) \\
\qquad \text{neighbourMsg}(src, as)\ \rightarrow \\
\qquad\quad \texttt{let}\ cs' = \text{updateNeighbourView}(cs, src, as) \\
\qquad\quad \texttt{let}\ cs'' = \text{updateRoutingState}(cs') \\
\qquad\quad \texttt{let}\ ps' = \text{costUpdateNeighbourView}(\text{costUpdateRoutingState}(ps)) \\
\qquad\quad \texttt{if}\ \text{transmitChange?}(cs, cs'') \vee \neg\text{consistent?}(cs') \\
\qquad\quad \texttt{then let}\ as' = \text{abstractView}(cs') \\
\qquad\qquad\quad \texttt{let}\ m = \text{neighbourMsg}(id, as') \\
\qquad\qquad\quad \texttt{send}_{id}(m); \text{Idle}_{id}(cs'', \text{costSend}(\text{costAbstractView}(ps'), m)) \\
\qquad\quad \texttt{else}\ \text{Idle}_{id}(cs', ps')
\end{array}
$$

Figure 7.5: The Treat-Message Phase

The state changes and phase transitions for the treat message phase are given in Figure 7.5. In this phase the node treats a single message. After the message is treated a transition to the idle phase is performed, where it can react to further events including the receiving of another message. A message is treated as follows:

- An observation message is treated by first checking whether this node is the destination for the message. If this is not the case, a direct transition to the idle phase is performed. Otherwise, the next destination is computed, the observation is forwarded to that destination and the physical state is updated taking the computation costs into account. The energy consumed by the test whether to discard or process a message is included in the energy consumption for receiving a message.

- A neighbour message must cause an update of the neighbour view part of the computational state giving a new state cs'. A new routing state cs'' must be computed. If the changes to the computational state is insignificant (in the sense transmitChange?(cs, cs'') is false and cs' is consistent), then a transition to the idle phase is performed with a computational state that is just updated with the new neighbour knowledge, and the physical which is updated by the computation cost. Otherwise, an abstract view of the computational state must be communicated to the neighbours, and the computational and the physical states are updated accordingly.

Notice that all phase transitions from the treat-message phase preserve the consistency of the computational state. The consistency preservation due to observation messages is trivial. The transition from $\text{TreatMsg}_{id}(m, cs, ps)$ to $\text{Idle}_{id}(cs'', \text{costSend}(\text{costAbstractView}(ps'), m))$ preserves consistency since cs'' is constructed by application of updateRoutingState, and this function is expected to return a consistent computational state. The transition from $\text{TreatMsg}_{id}(m, cs, ps)$ to $\text{Idle}_{id}(cs', ps')$ also preserves consistency since that transition can only occur when the `if`-condition transmitChange?$(cs, cs'') \vee \neg$consistent?(cs') is false.

Some of the main features of the operational descriptions in Figure 7.4 and Figure 7.5 are:

- A broad variety of instances of the operational descriptions can be achieved by providing different models for the sets and operations in Figure 7.2 and Figure 7.3. This emphasizes the generic nature of the model.

- The energy and neighbour parts of the model appear explicitly through the occurrence of the associated operations. Hence it is clear that the model reflects energy-aware routing using neighbour knowledge, and it is postponed to instantiations of the model to describe how it works.

- The energy cost model appears explicit in the form of the cost functions including the cost of events.

- A node will send a local view of its state to the neighbours only in the case when a significant change of the computational state has happened, which is determined by the transmitChange? predicate. The adequate definition of this predicate is a prerequisite for achieving a proper routing, as it is not difficult to imagine how it could load the network and drain the energy resources, if minimal changes to the states uncritically are broadcasted.
- The model is not biased towards a particular energy harvester and it is not biased towards and particular kind of sensor observation.

The generic model is based on the existence of a description of the medium through which the nodes communicates. This medium should at least determine which nodes can receive a message send by a given node in a given state. It may depend on the available energy, the geographical position, the distance from the sender, and a variety of other parameters. Furthermore, the medium may be unreliable so that messages may be lost.

The model describes the operational behaviour (including the dynamics of the energy levels in the nodes) for the normal operation of a network. It would be natural to extend the model with an initialization phase where a node through repeated communications with the neighbours are building up the knowledge of the environment needed to start normal operations, i.e. making observations and routing them to the base station. We leave out this initialization part in order to focus on energy harvesting and energy-aware routing.

7.3 Instantiating the modelling framework

In this section it will be demonstrated that the energy-aware routing protocol DEHAR [31] can be considered as an instance of the generic modelling framework presented in the previous section. In order to do so, meaning must be given to the sets and operations collected in Figure 7.2 and Figure 7.3. This will provide a succinct presentation of the main ideas behind DEHAR. Furthermore, we will show that the DD protocol [27] can be considered a special case of DEHAR. Concrete experiments, based on a simulation framework, depends on descriptions of the medium. This will be considered in Section 7.4.

7.3.1 A definition of the states

The abstract state comprises:

- A *simple distance* $d \in \mathcal{R}_{\geq 0}$ to the base station. This is described by a non-negative real number, where larger number means longer distance.
- An *energy-aware adjustment* $a \in \mathcal{R}_{\geq 0}$ of the distance for the route to the base station, where a larger distance means less energy is available.

Hence an abstract state is a pair $(d, a) \in \text{AbstractState}$, where

$$\text{AbstractState} = \mathcal{R}_{\geq 0} \times \mathcal{R}_{\geq 0}$$

For an abstract state (d, a), we call $\text{dist}(d, a) = d + a$ the *energy-adjusted distance*.

The computational state comprises:

- A *simple distance* $d \in \mathcal{R}_{\geq 0}$ to the base station, like the simple distance of an abstract state.
- An *energy level* $e \in \text{Energy}$.
- An *energy-faithful adjustment* $f \in \mathcal{R}_{\geq 0}$ capturing energy deficiencies along the route to the base station.
- A table nt containing entries for the *abstract state of neighbours*. This is modelled by the type: $\text{Id} \to \text{AbstractState}$.

Hence a computational state is a 4-tuple $(d, e, f, nt) \in \text{ComputationalState}$, where

$$\text{ComputationalState} = \mathcal{R}_{\geq 0} \times \text{Energy} \times \mathcal{R}_{\geq 0} \times (\text{Id} \to \text{AbstractState})$$

We shall assume that there is a function $\text{energyToDist} : \text{Energy} \to \mathcal{R}_{\geq 0}$ that converts energy to a distance so that less energy means longer distance.

The value $\text{energyToDist}(e)$ provides a local adjustment of the distance to the base station by just taking the energy level in the node into account. The intention with the energy-faithful adjustment is that the energy deficiencies along the route to the base station is taken into account, and the energy-faithful part is maintained by the use of the neighbour messages.

The *energy adjustment of a computational state* is the sum of the converted energy and the energy-faithful adjustment:

$$\text{adjust}(d, e, f, nt) = \text{energyToDist}(e) + f$$

and the energy-adjusted distance of a computational state is:

$$\text{dist}(d, e, f, nt) = d + \text{adjust}(d, e, f, nt) = d + \text{energyToDist}(e) + f$$

where we overload the dist function to be applied to both abstract and computational states. Furthermore, $\text{dist}(id)$, $id \in \text{Id}$, is the distance of the abstract state of the neighbour node N_{id}.

The function next : ComputationalState $\rightarrow$ Id should give the neighbour with the shortest energy-adjusted distance to the base station, i.e. the "best" neighbour to forward an observation. Hence, $\text{next}(d, e, f, nt)$ is the identity id of the entry $(id, as) \in nt$ with the smallest energy-adjusted distance to the base station, i.e. the smallest $\text{dist}(as)$. If several neighbours have the smallest distance an arbitrary one is chosen.

A computational state cs is consistent if $\text{next}(cs)$ has a smaller energy-adjusted distance than cs, i.e. $\text{dist}(cs) > \text{dist}(\text{next}(cs))$, hence

$$\text{consistent?}(cs) = \text{dist}(cs) > \text{dist}(\text{next}(cs))$$

A node with a consistent computational state has a neighbour to which it can forward an observation. But if the state is inconsistent, then all neighbours have longer energy-adjusted distances to the base station and it does not make sense to forward an observation to any of these neighbours.

We illustrate the intuition behind the adjusted distance using the example network example from Figure 7.1. If the energy level in node N_e of this network is decreased, then the distance of N_e to the base station is increased accordingly (by the amount $\text{energyToDist}(e)$) as shown in Figure 7.6. All nodes are

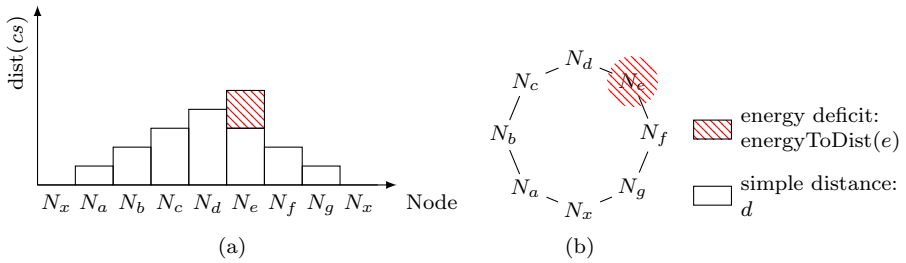


Figure 7.6: The example from Figure 7.1 with an energy adjustment for N_e due to shaded region shown to the right.

still consistent; but in contrast to the situation in Figure 7.1, the node N_d (in

Figure 7.6) has just one neighbour (N_c) with a shorter energy-adjusted distance to the base station.

Consider now the situation shown in Figure 7.7 with energy adjustments for the nodes N_f and N_g. These adjustments make the node N_e inconsistent, since its neighbours N_d and N_f both have energy-adjusted distances which are longer than that of N_e. In the shown situation it would make no sense for N_e to forward observations to its "best" neighbour, which is N_f, since N_f would immediately return that observation to N_e since N_e is the "best" neighbour of N_f.

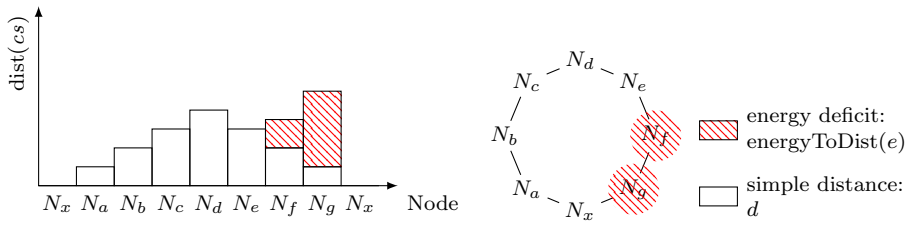


Figure 7.7: Revised example with an inconsistent node: N_e.

Energy-faithful adjustments can be used to cope with inconsistent nodes. By adding such adjustments to the "problematic nodes" inconsistencies may be avoided. This is shown in Figure 7.8, where energy-faithful adjustments (f) have been added to N_e and N_f. Every node is consistent, and there is a natural route from every node to the base station. From N_f there are actually two possible routes.

The physical state comprises:

- The stored energy $e \in$ Energy.
- A model of the energy harvester. In the DEHAR case it is a solar panel, which is modelled by a function $P(t)$ describing the effect of the solar insolation at time t.
- A model of the energy store. In the DEHAR case it is an *ideal capacitor* with a given capacity. It is ideal in the sense that it does not loose energy.
- A model of the computational unit. This model must define the costs of the computational operations by providing definitions for the cost functions in

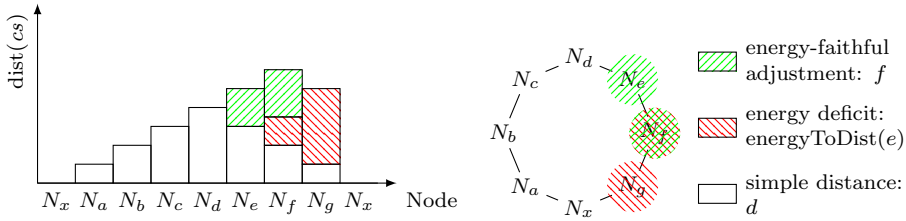


Figure 7.8: A with consistent nodes using energy-faithful adjustments

Figure 7.3. A simple way of doing this is to count the instructions needed for executing the individual functions, and multiply it with the energy needed per instruction. The model can be more fine grained by taking different modes of the processing unit into account.

- A model of the transmitter. This model must give a definition of the cost function: costSend : PhysicalState $\times$ Message $\rightarrow$ PhysicalState. In the DEHAR case the cost of sending is a simple linear function in the size of the message.

- A model of the receiver. This model must explain the cost of a receive event receiveEvent(m, ps). This involves the cost of receiving the message m and it must also take the intervals into account when the receiver is *idle listening*, i.e. it actively listens for incoming messages. Thus ps should reflect the full energy consumption of the receiver since the last receive event.

- A model of the sensor. This model must explain the cost of an observation event observationEvent(o, ps). This involves the cost of sensing o and ps should reflect this energy consumption.

The model should also describe two transitions of the physical state which relate to the two events physicalStateEvent(ps) and readEnergyEvent(e, ps).

The transition related to a physicalStateEvent must take into account at least the dynamics of the energy harvester, the dynamics of the energy store, the time the computational unit spent in the idle phase, and the time elapsed since the last physical state event. For example the new stored energy e' in the

physical state at time t' is given by:

$$e' = e + \int_t^{t'} P(t)dt$$

where t is the time where the old energy e was stored.

The transition related to a readEnergyEvent(e, ps) must take into account at least the cost of reading the energy.

Definition of operations

The function for extracting the abstract view is defined by:

$$\text{abstractView}(d, e, f, nt) = (d, \text{adjust}(d, e, f, nt))$$

Notice that the distance to the base station is preserved by the conversion from a computational state to an abstract one:

$$\text{dist}(d, e, f, nt) = \text{dist}(\text{abstractView}(d, e, f, nt))$$

The definitions of the functions for updating the energy state and the neighbour view are simple:

$$\begin{array}{lcl} \text{updateEnergyState}((d, e, f, nt), e') & = & (d, e', f, nt) \\ \text{updateNeighbourView}((d, e, f, nt), id, as) & = & (d, e, f, \text{update}(nt, id, as)) \end{array}$$

where update(nt, id, as) gives the neighbour table obtained from nt by mapping id to the abstract state as. These two operations may transform a consistent state into an inconsistent one.

The function updateRoutingState(d, e, f, nt) must update the energy adjustment of a computational state in order to arrive at a consistent one. If the state is consistent even when $f = 0$ then no adjustment is necessary. Otherwise, an adjustment is made so that the distance of the computational state becomes K larger than the distance of its "best" neighbour (given by the next function):

$$\begin{array}{l} \text{updateRoutingState}(d, e, f, nt) = \\ \quad \texttt{if}\ \text{consistent?}(d, e, 0, nt) \\ \quad \texttt{then}\ (d, e, 0, nt) \\ \quad \texttt{else let}\ \text{distNext} = \text{dist}(\text{next}(d, e, f, nt)) \\ \qquad (d, e, K + \text{distNext} - (d + \text{energyToDist}(e)), nt) \end{array}$$

where $K > 0$ is a constant used to enforce a consistent computational state.

The energy adjustment in the `else`-branch of this function has the effect that the node becomes less attractive to forward messages to in the case of an energy drop in the node or in the best neighbour.

The function transmitChange?(cs, cs') is a predicate which is true when a change of the computational state from cs to cs' is significant enough to be communicated to the neighbours. This is the case if the change reflects a significant change in distance to base station, where significant in this case means larger than some constant $K_{\text{change}} \in \mathcal{R}_{\geq 0}$.

Hence, the function can be defined as follows:

$$\text{transmitChange?}(cs, cs') = |\text{dist}(cs) - \text{dist}(cs')| > K_{\text{change}}$$

A simple check of the operational descriptions in Figure 7.4 and Figure 7.5 shows that the new computational state used as argument to transmitChange? (cs' in Figure 7.4 and cs'' in Figure 7.5) must be consistent as it is created using updateRoutingState. Hence it is just necessary to define transmitChange? for consistent computational states.

7.3.2 Directed diffusion – another instantiation of the generic framework

It should be noticed that the routing algorithm DD [27] is a simple instance of the generic framework, which can be achieved by simplifying the DEHAR instance so that

- the simple distance is the number of hops to the based station (as for DEHAR) and
- the energy is assumed perfect and hence the adjustments have no effect (are 0).

Hence DD do not support any kind of energy-aware routing.

Actually, it is the algorithm behind DD which is used to initialize the simple distances of nodes in the DEHAR algorithm.

The DD algorithm provides a good model of reference for comparison with energy harvesting aware routing algorithms like DEHAR, since DD incorporates nodes with an energy harvesting capability, but the routing is static in the sense that an observation is always transmitted along the path with the smallest number of hops to the base station. Energy harvesting aware routing algorithms will not necessarily choose this shortest path, since problematic

low-energy nodes should be avoided in order to keep all nodes "alive" as long as possible. Therefore, the total energy consumption in a DD based network should be smaller than the total energy consumption of any energy harvesting aware network (due to longer paths in the latter). On the other hand, energy harvesting awareness can spare low-energy nodes, and there are two important consequences of this:

- A drain of low-energy nodes can be avoided or at least postponed. With regard to this aspect DD should perform worse since these nodes are not spared at all in the routing.

- The total energy stored in a network should exceed that of a corresponding DD based network, since messages are transmitted through nodes with good energy harvesting capability. The reason for this is that low-energy nodes get a chance to recover and that transmissions through high-energy nodes, with a full energy storage, are close to be "free of charge" since there would be almost no storage available for harvested energy in these high-energy nodes.

7.4 Results from simulation of the model

In this section we will study the properties of the energy harvesting aware routing algorithm DEHAR by analysing results [31] of a simulator implementing the DEHAR and DD algorithms. The simulator is a custom-made simulator [30] implemented in the language Java. It can be configured through a comprehensive xml configuration file which includes the network layout, environmental properties (insolation, shadows, etc.) and properties of nodes (such as processor states, radio model, and frequency of observations). The simulator features a classic event driven engine. The simulator produces a trace of observations of the nodes, including energy levels, activity of devices, and environmental properties

The considered network is given in Figure 7.9. The network has one very problematic node, due to a strong shadow, at coordinate $(1, 3)$, and five nodes with potential problems due to light shadows. We will analyse the ability of the routing algorithms to cope with these problematic nodes using simulations.

The medium and the physical setting must be defined for the experiments. It is assumed that a node can communicate with its immediate horizontal and vertical neighbour, i.e. the radio range is 1. Two experiments $S1$ and $S2$ are

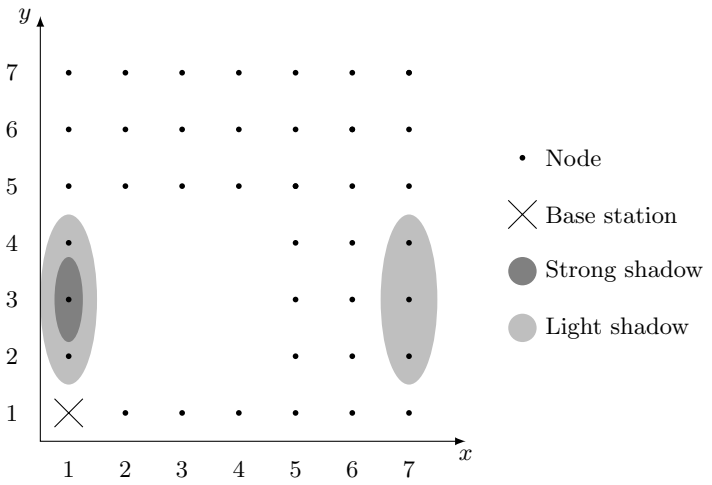


Figure 7.9: A network structure with illustrating problematic nodes

conducted, one with a low and another with a high rate of conducted observations. Table 7.1 shows the parameters that are used in the presented simulations. Only the observation rate is changed between the two simulations.

The energy model is based on real insolation data for a two-weeks period. The data is repeated in simulations over longer periods. To emphasize the effect of the DEHAR algorithm, the insolation pattern have been idealised to either full noon or midnight, i.e. 12 hours of light and 12 hours of darkness. The insolation data is suitably scaled for individual nodes to achieve the shadow effect shown in Figure 7.9.

7.4.1 Energy awareness makes a difference

A 30 day view of the simulations S_1 with the low observation rate is shown in Figure 7.10. The figure shows the energy available in the worst node with minimum energy in the network. The two algorithms cannot be distinguished the first five days. Thereafter, the energy aware routing starts and DEHAR stabilises at a high level where no node is in any danger of being drained for energy. In the DD case, the energy of worst node is steadily drained at a (rather) constant rate and in an foreseeable future it will stop working.

			S_1	S_2	unit
Radio	Range		1	1	
	Transmit power		50	50	mW
	Idle listening power		5.5	5.5	mW
	Bandwidth		45	45	kb/s
Processor	Sleep	Power	1	1	μW
	Active	Frequency	1	1	MHz
		Power	10	10	μW
Battery	Capacity		4	4	kJ
Solar panel	Efficiency		6.25	6.25	%
	Area		12.5	12.5	cm^2
Application parameters	Observation rate		$\frac{1}{900}$	$\frac{1}{60}$	sec^{-1}
Routing parameters	Sense rate		$\frac{1}{1800}$	$\frac{1}{1800}$	sec^{-1}

Table 7.1: Parameters used in simulations.

7.4.2 Energy awareness consumes and stores more energy

The total power consumed and the average energy stored per node in the network are monitored for the same simulations as in Figure 7.10. These results are shown for the first 10 days of simulated time in Figure 7.11.

The day cycle is clearly visible in Figure 7.11a where the nodes recharge during day and discharge during night. The first five days of simulation do not show any significant difference between DEHAR and DD. During the last five days the DEHAR algorithm makes the network able to harvest and store more energy.

The next graph (Figure 7.11b) shows the difference of the two curves from the previous. It shows (in the blow-up) that just before day five ends, the DEHAR algorithm starts to consume significantly more energy than the DD algorithm. By looking at the third graph (Figure 7.11c) which shows the difference in total network energy consumption, it can be confirmed. This extra energy consumption arises from observation packages that travel along longer routes in the network, because the DEHAR algorithm has detected a lower amount of stored energy in some nodes.

Even though the DEHAR consumes more energy due to the longer routes, it can store more energy on average in the nodes. The reason for this is that the extra energy consumption of DEHAR is taken from nodes that are able to

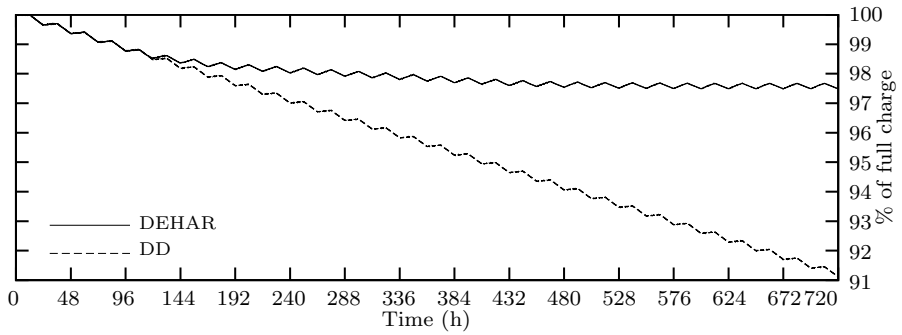


Figure 7.10: Results of simulations S_1 for a 30 day simulation. This graph shows the minimum energy in any node in the network.

recharge fully during daytime. This can be seen in Figure 7.11b (in the blow-up) at the beginning of day 5 (120h), where the graph shows a sudden rise.

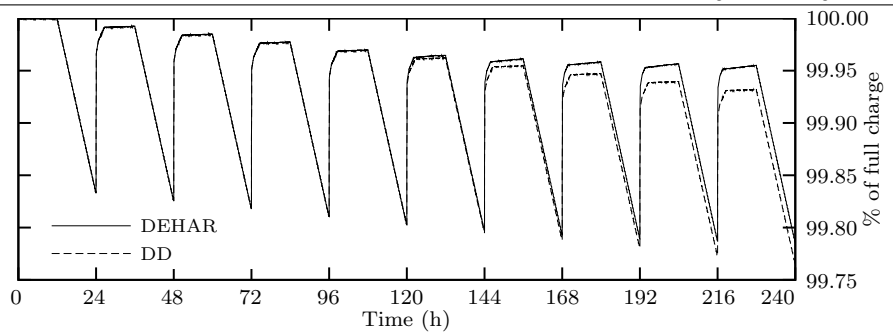
After a short while, the network with the DD algorithm is able to harvest energy at a greater rate than DEHAR. This is due to the fact that the majority of the nodes in the DEHAR network are fully charged. The key point at this time is that the DD algorithm does not allow the network to harvest as much energy as the DEHAR algorithm. This can also be seen through the rest of the daylight during day 5, where the DEHAR network is able to harvest energy at a higher rate than the DD network.

Finally, during night, the DEHAR network again shows a higher energy consumption than the DD network. Hence the graph shows a slow decline.

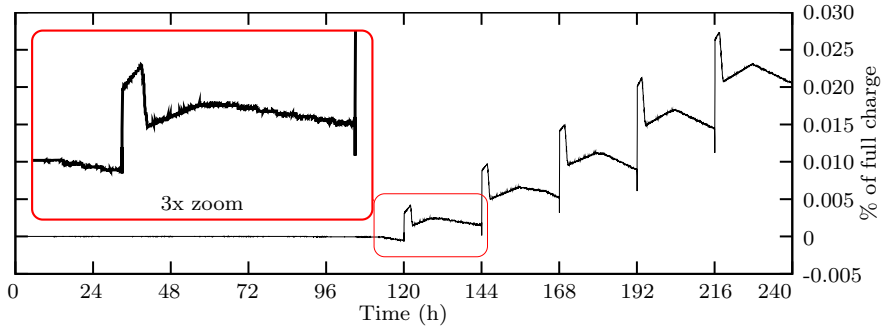
7.4.3 Increasing the rate of observations costs

The next simulations (S_2) have an increased rate of observations and thus an increased radio traffic in the network. The effect of the increased data rate is primarily that the network consumes more power. This extra power consumption speeds up the time from the start of the simulation until the network finds the alternate routing pattern compared to the S_1 simulations.

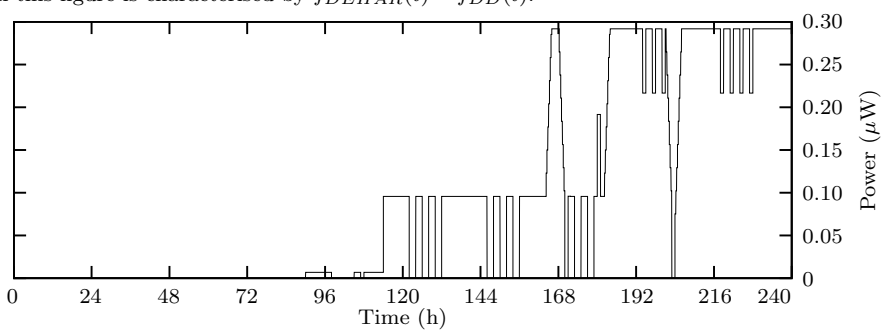
Figure 7.12 shows that the minimum energy in any of the nodes in the network stabilises with the DEHAR algorithm. The level at which it stabilises is lower than in the S_1 simulations, which is expectable. The faster observation rate hurts the DD network and a node will already be drained of energy in about



(a) Average energy in nodes for each simulation of S_1.



(b) Difference in the average energy in nodes for simulations in S_1. Given that the two curves in Figure 7.11a are characterised by the functions $f_{DEHAR}(t)$ and $f_{DD}(t)$, then the curve in this figure is characterised by $f_{DEHAR}(t) - f_{DD}(t)$.



(c) Surplus energy consumption by DEHAR compared with DD for simulations in S_1.

Figure 7.11: Results of simulations S_1 showing the first 10 days. The blow-up in (b) emphasises the first important difference between the DEHAR and DD algorithms.

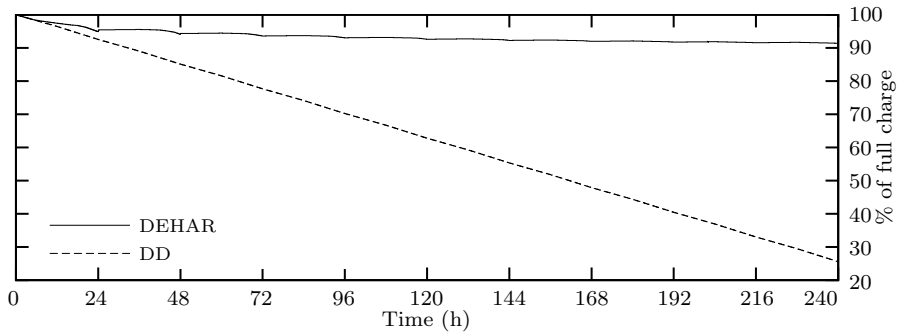


Figure 7.12: Minimum energy in any node of the simulations in S_2. The day cycle is barely visible due to the compressed y-scale, compared to the simulations S_1.

10 days.

The routing trend of the DEHAR algorithm is the same in the simulations S_1 and S_2. The only difference is that the DEHAR algorithm finds this alternative routing pattern faster in S_2 than in S_1.

The energy statistics of the node covered by the strongest shadow (at coordinate (1,3)) can be analysed. A graph of the energy level of this node will look similar to Figure 7.12 and (in this simulation) it stabilises at precisely the same energy level. This shows that the energy it can harvest closely matches the energy it needs to perform routing updates and performing observations (i.e. refraining from routing other nodes observations).

7.5 Summary

We have presented a new modelling framework aimed at describing and analysing wireless sensor networks with energy harvesting capabilities. The framework is comprised of a conceptual basis and an operational basis, which were used to describe and explain two wireless sensor networks with energy harvesting capabilities. One of these network models is based on DD, i.e. it supports energy harvesting; but the routing is not energy aware, as it just forwards observations to the base station along statically defined shortest paths. The other network model is based on the energy harvesting aware routing protocol DEHAR. Both

of these networks were given natural explanations using the concepts of the modelling framework, and this gives a first weak validation of the adequacy of the framework. More experiments are, of course, needed for a thorough validation. Simulation results show that energy awareness of DEHAR-based networks can significantly extend the lifetime of nodes and it significantly improves the energy stored in the network, compared with a network like DD, with no energy aware routing.

There are several natural extensions of this work.

First of all, the modelling framework should be validated by establishing its applicability in a broad collection of energy harvesting aware networks. The framework should be extended to include the deployment phase, where the nodes communicate in order to initialize their states. We do not expect principle difficulties in these extensions, but they are, of course, technical.

The generic framework may be instantiated in ways which will not be beneficial for the energy situation in the network. It is desirable and challenging to establish conditions which instantiations should satisfy in order to define an adequate energy harvesting aware network.

Another natural development would be to implement a platform for the modelling framework. The formalized parts of the framework provide good bases for such an implementation; but further formalization concerning the network communication and the medium should be considered prior to an implementation.

Chapter 8

Conclusion and perspectives

Cyper physical systems (CPSs) are becoming more and more complex. The increased complexity makes it necessary to use computer aided design (CAD) methods. However, it is challenging to express dynamic CPSs (e.g. the wireless sensor network (WSN) routing algorithm presented in Chapter 2) in today's formal frameworks such that they are analysable.

The hearing aid calibration device presented in Chapter 6, is an example of a static CPS which is independent of the platform and the environment as such. It has a static communication protocol with its environment (the PC and the audio measurements). In other words all possible state changes of the system are known, and all possible inputs causing state changes are known. Such systems can be analysed without a complete model of its environment, i.e. how the environment behaves, only a model of the possible inputs from the environment.

8.1 Summary

The goal of this thesis has been to model and analyse the dynamic CPSs. In the process of reaching this goal, the following milestones have been reached.

- The work of the thesis is illustrated by two use cases: A company use case and a WSN use case.
  - The company (`www.auditdata.com`) use case illustrates static systems expressed in ForSyDe.

  - The WSN use case illustrates a dynamic system.

- The distributed energy harvesting aware routing (DEHAR) algorithm for WSNs has been designed.

- The DEHAR algorithm has been analysed in UPPAAL.

  - For small networks the UPPAAL analysis indicates that the DEHAR algorithm works correctly.

- To formally express dynamic WSN systems, a framework is defined.

- The ForSyDe framework has been developed to cope with heterogeneous models.

  - Four models of computation (MoCs) are mathematically defined and implemented in Haskell-ForSyDe: synchronous (SY), synchronous data flow (SDF), discrete event (DE), and continuous time (CT).
  - Domain interfaces for combining MoCs are defined in general.
  - Structured domain interfaces are developed for a formal definition of domain interfaces.

The overall goal has been reached. However, the ForSyDe framework, with the current MoCs, has a static structure which makes it difficult to naturally express dynamic systems. In Chapter 7 a framework for expressing dynamic WSNs is defined. This WSN framework is a challenging case study for modelling and analysing dynamic systems, it could guide future development around ForSyDe.

The work of this thesis has inspired the following projects:

- The UPPAAL team are working on implementing the WSN framework presented in Chapter 7 in the UPPAAL framework.

- The WSN framework is also implemented in the programming language F-Sharp by the Ph.D. student Phan Anh Dung. The focus of this implementation is to model and simulate various routing protocols, one of which is the DEHAR algorithm. This work focuses on studying the network behaviour of the routing algorithms.

- A master student, Nan Wu, has worked on expressing the WSN routing algorithm (described in Chapter 2) in a probabilistic model. The aim is

to perform stochastic verification. The work is promising, as rather than having a static structure where all possible structures must be described, they are described in stochastic terms.

8.2 Perspectives

The work of this thesis is part of the SYSMODEL project in which other work has been created in close cooperation. It has also inspired several current projects and some future work.

- The SYSMODEL project partners have defined wrappers for co-simulating foreign models with ForSyDe [56]. This article also presents a ForSyDe implementation in SystemC. This has been done to increase the availability of this work to the industry.
- A graphical user interface for creating ForSyDe models, called ForSyDe graphical editor (FGE), has been created [1]. It provides a graphical modelling environment and the ability to provide implementations of the leaf-processes. The models created with FGE can be translated into Haskell code, which uses the Haskell-ForSyDe implementation. The perspectives of the FGE are that it can combine the ForSyDe framework with external analysis tools and combine the results from the external tools in the graphical model. The results of the external tools and simulations can be used as constraints, which can be checked for consistency.

Future work that the work of this thesis could inspire is:

- The UPPAAL framework and other static analysis frameworks proved not to be able to express the dynamic WSN system. Maybe the Generic Modeling Environment (GME) could be an interesting framework to test. The models are described through constraints. If the GME framework is able to express dynamic systems, it is possible to statically check the constraints, however, it will not be possible to simulate the model as GME does not facilitate this.
- A dedicated MoC could be defined in ForSyDe to express UPPAAL models. This would give the benefit of simulating heterogeneous systems in ForSyDe, the static analysis from UPPAAL of the sub-model expressed in the UPPAAL-MoC, and the possibility of automatic translation of models between the two frameworks.

- The Ptolemy framework implements a static analysis of a subset of models expressed in the SR and the DE MoCs, this work can be exploited to make static analysis of similar SY and DE MoCs expressed in ForSyDe.
- The four MoCs present in ForSyDe can express both hardware, software and systems in general, however, especially for software and systems, systems are practically limited to streaming applications and synchronous control applications. Other MoCs, e.g. the concurrent sequential processes (CSP) MoC, could be interesting to include in ForSyDe to express more general application and system behaviour. The challenge here is especially to define formal domain interfaces to the existing MoCs.

Appendix A

ForSyDe-Haskell implementation

A.1 Synchronous model of computation

Signal definition

```
1  -----------------------------------------------------------------------------
2  --
3  -- Module      :  ForSyDe.Shallow.SY.Signal.Internal
4  -- Copyright   :
5  -- License     :  AllRightsReserved
6  --
7  -- Maintainer  :  mkoe@imm.dtu.dk
8  -- Stability   :  Experimental
9  -- Portability :
10 --
11 -- |
12 --
13 -----------------------------------------------------------------------------
14
15 module ForSyDe.Shallow.SY.Signal.Internal (
16     Signal (..),
17 ) where
18
19 import Control.DeepSeq (NFData(rnf))
20
21 data Signal a = a :- Signal a | Null
22
23 infixr 4 :-
24
25 instance (Show a) => Show (Signal a) where
26     showsPrec p = showParen (p>1) . showSignal
```

```
27
28 showSignal :: (Show a) => Signal a -> ShowS
29 showSignal (x :- xs) = showChar '{' . showEvent x . showSignal' xs
30 showSignal (Null)    = shows "{}"
31
32 showSignal' :: (Show a) => Signal a -> ShowS
33 showSignal' (x :- xs) = showChar ',' . showEvent x . showSignal' xs
34 showSignal' (Null)    = showChar '}'
35
36 -- Show an event token
37 showEvent :: (Show a) => a -> ShowS
38 showEvent x = shows x
39
40 instance (Read a) => Read (Signal a) where
41         readsPrec d = readParen (d>1) readSignalStart
42
43 readSignalStart :: (Read a) => ReadS (Signal a)
44 readSignalStart = (\ a -> [(xs,c) | ("{",b) <- lex a , (xs,c) <-
       readSignal (',' : b) ++ readNull b])
45
46 readSignal :: (Read a) => ReadS (Signal a)
47 readSignal r = readEvent r ++ readNull r
48
49 -- | Read an event token
50 readEvent :: (Read a) => ReadS (Signal a)
51 readEvent a = [(x :- xs,d) | (",",b) <- lex a , (x,c) <- reads b ,
       (xs,d) <- readSignal c]
52
53 -- | Read a null token
54 readNull :: (Read a) => ReadS (Signal a)
55 readNull a = [(Null,b) | ("}",b) <- lex a]
56
57 instance (NFData a) => NFData (Signal a) where
58     rnf (x :- xs) = rnf x `seq` rnf xs
59     rnf Null      = ()
```

Process atoms

```
1  ------------------------------------------------------------------------------
2  --
3  -- Module      :  ForSyDe.Shallow.SY.Signal
4  -- Copyright   :
5  -- License     :  AllRightsReserved
6  --
7  -- Maintainer  :  mkoe@imm.dtu.dk
8  -- Stability   :  Experimental
9  -- Portability :
10 --
11 -- |
12 --
13 ------------------------------------------------------------------------------
14
15 module ForSyDe.Shallow.SY.Signal (
16     Signal,
17     signal,
18     fromSignal,
19     map,
```

```
20      zip,
21      delay,§
22      (>),§
23      (>>),
24      (+>),⊙
25      (),⊕
26      (),Δ
27      (),
28  ) where
29
30  import Prelude ()
31  import ForSyDe.Shallow.SY.Signal.Internal
32
33  infixl 5 `map`, `zip`, `delay`, §>, §>>, +>, ⊙, ⊕, Δ
34
35  ------------------------------------------------------------------------------
36
37  -- ⊙ : Unicode U+2A00, UTF-8 Hex E2 A8 80⊙
38  (⊙) :: (a -> b) -> Signal a -> Signal b⊙
39  (⊙) = map
40
41  -- ⊕ : Unicode U+2A01, UTF-8 Hex E2 A8 81⊕
42  (⊕) :: Signal (a -> b) -> Signal a -> Signal b⊕
43  (⊕) = zip
44
45  -- Δ : Unicode U+2206, UTF-8 Hex E2 88 86Δ
46  (Δ) :: Signal a -> a -> Signal aΔ
47  (Δ) = delay
48
49  ------------------------------------------------------------------------------§
50
51  (§>) :: (a -> b) -> Signal a -> Signal b§
52  (§>) = map§
53
54  (§>>) :: Signal (a -> b) -> Signal a -> Signal b§
55  (§>>) = zip
56
57  (+>) :: Signal a -> a -> Signal a
58  (+>) = delay
59
60  ------------------------------------------------------------------------------
61
62  map :: (a -> b) -> Signal a -> Signal b
63  map f (x :- xs)   = f x :- f §> xs
64  map _ Null        = Null
65
66  zip :: Signal (a -> b) -> Signal a -> Signal b
67  zip (f :- fs) (x :- xs) = f x :- fs §>> xs
68  zip _         Null      = Null
69  zip Null      _         = Null
70
71  delay :: Signal a -> a -> Signal a
72  delay xs x = x :- xs
73
74  ------------------------------------------------------------------------------
75
76  signal :: [a] -> Signal a
77  signal (x : xs) = x :- signal xs
```

```
78  signal []       = Null
79
80  fromSignal :: Signal a -> [a]
81  fromSignal (x :- xs) = x : fromSignal xs
82  fromSignal Null      = []
```

Process constructors

```
1   -----------------------------------------------------------------------------
2   --
3   -- Module      :  ForSyDe.Shallow.SY.Processes
4   -- Copyright   :
5   -- License     :  AllRightsReserved
6   --
7   -- Maintainer  :  mkoe@imm.dtu.dk
8   -- Stability   :  Experimental
9   -- Portability :
10  --
11  -- |
12  --
13  -----------------------------------------------------------------------------
14
15  module ForSyDe.Shallow.SY.Processes (
16      map,
17      zipWith,
18      zipWith3,
19      zipWith4,
20      delay,
21      delayn,
22      scanl,
23      scanl2,
24      scanl3,
25      scanld,
26      scanld2,
27      scanld3,
28      moore,
29      moore2,
30      moore3,
31      mealy,
32      mealy2,
33      mealy3,
34      source,
35      zip,
36      zip3,
37      zip4,
38      unzip,
39      unzip3,
40      unzip4,
41  ) where
42
43  import Prelude (otherwise, flip, Ord, Num(<=), (-), fst, snd)
44  import ForSyDe.Shallow.SY.Signal (Signal, (⊙), (⊕), (Δ), (), map)
45
46  zipWith :: (a -> b -> c) -> Signal a -> Signal b -> Signal c
47  zipWith f xs ys = f ⊙ xs ⊕ ys
48
```

```
49  zipWith3 :: (a -> b -> c -> d) -> Signal a -> Signal b -> Signal c ->
        Signal d
50  zipWith3 f xs ys zs = f ⊙ xs ⊕ ys ⊕ zs
51
52  zipWith4 :: (a -> b -> c -> d -> e) -> Signal a -> Signal b -> Signal c
        -> Signal d -> Signal e
53  zipWith4 f ws xs ys zs = f ⊙ ws ⊕ xs ⊕ ys ⊕ zs
54
55  delay :: a -> Signal a -> Signal a
56  delay = flip Δ()
57
58  delayn :: (Ord n,Num n) => a -> n -> Signal a -> Signal a
59  delayn x n xs
60      | n <= 0     = xs
61      | otherwise = delayn x (n-1) xs Δ x
62
63  scanl :: (a -> b -> a) -> a -> Signal b -> Signal a
64  scanl f mem xs = s
65      where
66      s = f ⊙ (s Δ mem) ⊕ xs
67
68  scanl2 :: (a -> b -> c -> a) -> a -> Signal b -> Signal c -> Signal a
69  scanl2 f mem xs ys = s
70      where
71      s = f ⊙ (s Δ mem) ⊕ xs ⊕ ys
72
73  scanl3 :: (a -> b -> c -> d -> a) -> a -> Signal b -> Signal c ->
        Signal d -> Signal a
74  scanl3 f mem xs ys zs = s
75      where
76      s = f ⊙ (s Δ mem) ⊕ xs ⊕ ys ⊕ zs
77
78  scanld :: (a -> b -> a) -> a -> Signal b -> Signal a
79  scanld f mem xs = s
80      where
81      s = f ⊙ s ⊕ xs Δ mem
82
83  scanld2 :: (a -> b -> c -> a) -> a -> Signal b -> Signal c -> Signal a
84  scanld2 f mem xs ys = s
85      where
86      s = f ⊙ s ⊕ xs ⊕ ys Δ mem
87
88  scanld3 :: (a -> b -> c -> d -> a) -> a -> Signal b -> Signal c ->
        Signal d -> Signal a
89  scanld3 f mem xs ys zs = s
90      where
91      s = f ⊙ s ⊕ xs ⊕ ys ⊕ zs Δ mem
92
93  moore :: (a -> b -> a) -> (a -> c) -> a -> Signal b -> Signal c
94  moore nextState output mem xs = output ⊙ s
95      where
96      s = nextState ⊙ s ⊕ xs Δ mem
97
98  moore2 :: (a -> b -> c -> a) -> (a -> d) -> a -> Signal b -> Signal c
        -> Signal d
99  moore2 nextState output mem xs ys = output ⊙ s
100     where
101     s = nextState ⊙ s ⊕ xs ⊕ ys Δ mem
```

```
102
103 moore3 :: (a -> b -> c -> d -> a) -> (a -> e) -> a -> Signal b ->
        Signal c -> Signal d -> Signal e
104 moore3 nextState output mem xs ys zs = output ⊙ s
105     where
106     s = nextState ⊙ s ⊕ xs ⊕ ys ⊕ zs Δ mem
107
108 mealy :: (a -> b -> a) -> (a -> b -> c) -> a -> Signal b -> Signal c
109 mealy nextState output mem xs = output ⊙ s ⊕ xs
110     where
111     s = nextState ⊙ s ⊕ xs Δ mem
112
113 mealy2 :: (a -> b -> c -> a) -> (a -> b -> c -> d) -> a -> Signal b ->
        Signal c -> Signal d
114 mealy2 nextState output mem xs ys = output ⊙ s ⊕ xs ⊕ ys
115     where
116     s = nextState ⊙ s ⊕ xs ⊕ ys Δ mem
117
118 mealy3 :: (a -> b -> c -> d -> a) -> (a -> b -> c -> d -> e) -> a ->
        Signal b -> Signal c -> Signal d -> Signal e
119 mealy3 nextState output mem xs ys zs = output ⊙ s ⊕ xs ⊕ ys ⊕ zs
120     where
121     s = nextState ⊙ s ⊕ xs ⊕ ys ⊕ zs Δ mem
122
123 source :: (a -> a) -> a -> Signal a
124 source f mem = o
125     where
126     o = s Δ mem
127     s = f ⊙ o
128
129 zip :: Signal a -> Signal b -> Signal (a,b)
130 zip xs ys = (,) ⊙ xs ⊕ ys
131
132 zip3 :: Signal a -> Signal b -> Signal c -> Signal (a,b,c)
133 zip3 xs ys zs = (,,) ⊙ xs ⊕ ys ⊕ zs
134
135 zip4 :: Signal a -> Signal b -> Signal c -> Signal d -> Signal (a,b,c,d)
136 zip4 ws xs ys zs = (,,,) ⊙ ws ⊕ xs ⊕ ys ⊕ zs
137
138 unzip :: Signal (a,b) -> (Signal a,Signal b)
139 unzip xs = (fst ⊙ xs,snd ⊙ xs)
140
141 unzip3 :: Signal (a,b,c) -> (Signal a,Signal b,Signal c)
142 unzip3 xs = ((\ (x,_,_) -> x) ⊙ xs,(\ (_,x,_) -> x) ⊙ xs,(\ (_,_,x) ->
        x) ⊙ xs)
143
144 unzip4 :: Signal (a,b,c,d) -> (Signal a,Signal b,Signal c,Signal d)
145 unzip4 xs = ((\ (x,_,_,_) -> x) ⊙ xs,(\ (_,x,_,_) -> x) ⊙ xs,(\
        (_,_,x,_) -> x) ⊙ xs,(\ (_,_,_,x) -> x) ⊙ xs)
```

Domain interface tools

```
1 ------------------------------------------------------------------------------
2 --
3 -- Module      :  ForSyDe.Shallow.SY.Interface
4 -- Copyright   :
5 -- License     :  AllRightsReserved
```

```
6  --
7  -- Maintainer  :  mkoe@imm.dtu.dk
8  -- Stability   :  Experimental
9  -- Portability :
10 --
11 -- |
12 --
13 ------------------------------------------------------------------------------
14
15 module ForSyDe.Shallow.SY.Interface (
16     head,
17     tail,
18     take,
19     drop,
20     length,
21 ) where
22
23 import Prelude hiding (head,tail,take,drop,length)
24 import ForSyDe.Shallow.SY.Signal.Internal
25
26 head :: Signal a -> Maybe a
27 head Null      = Nothing
28 head (x :- _) = Just x
29
30 tail :: Signal a -> Maybe (Signal a)
31 tail Null      = Nothing
32 tail (_ :- xs) = Just xs
33
34 take :: Int -> Signal a -> Signal a
35 take n (x :- xs)
36     | n <= 0     = Null
37     | otherwise = x :- take (pred n) xs
38 take _ Null     = Null
39
40 drop :: Int -> Signal a -> Signal a
41 drop n (x :- xs)
42     | n <= 0     = x :- xs
43     | otherwise = drop (pred n) xs
44 drop _ Null     = Null
45
46 length :: Signal a -> Int
47 length (_ :- xs) = 1 + length xs
48 length Null      = 0
```

A.2 Synchronous data flow model of computation

Signal definition

```
1  ------------------------------------------------------------------------------
2  --
3  -- Module      :  ForSyDe.Shallow.SDF.Signal.Internal
4  -- Copyright   :
5  -- License     :  AllRightsReserved
```

```
6  --
7  -- Maintainer  :  mkoe@imm.dtu.dk
8  -- Stability   :  Experimental
9  -- Portability :
10 --
11 -- |
12 --
13 -----------------------------------------------------------------------------

14
15 module ForSyDe.Shallow.SDF.Signal.Internal (
16     Signal (..),
17 ) where
18
19 data Signal a = a :- Signal a | Null
20
21 infixr 4 :-
22
23 instance (Show a) => Show (Signal a) where
24     showsPrec p = showParen (p>1) . showSignal
25
26 showSignal :: (Show a) => Signal a -> ShowS
27 showSignal (x :- xs) = showChar '{' . showEvent x . showSignal' xs
28 showSignal Null      = showString "{}"
29
30 showSignal' :: (Show a) => Signal a -> ShowS
31 showSignal' (x :- xs) = showChar ',' . showEvent x . showSignal' xs
32 showSignal' Null      = showChar '}'
33
34 -- | Show an event token
35 showEvent :: (Show a) => a -> ShowS
36 showEvent x = shows x
37
38 instance (Read a) => Read (Signal a) where
39         readsPrec d = readParen (d>1) readSignalStart
40
41 readSignalStart :: (Read a) => ReadS (Signal a)
42 readSignalStart = (\ a -> [(xs,c) | ("{",b) <- lex a , (xs,c) <-
       readSignal (',' : b) ++ readNull b])
43
44 readSignal :: (Read a) => ReadS (Signal a)
45 readSignal r = readEvent r ++ readNull r
46
47 -- | Read an event token
48 readEvent :: (Read a) => ReadS (Signal a)
49 readEvent a = [(x :- xs,d) | (",",b) <- lex a , (x,c) <- reads b ,
       (xs,d) <- readSignal c]
50
51 -- | Read a null token
52 readNull :: (Read a) => ReadS (Signal a)
53 readNull a = [(Null,b) | ("}",b) <- lex a]
```

Process atoms

```
1  -----------------------------------------------------------------------------
2  --
3  -- Module      :  ForSyDe.Shallow.SDF.Signal
4  -- Copyright   :
```

```
5  -- License     :  AllRightsReserved
6  --
7  -- Maintainer  :  mkoe@imm.dtu.dk
8  -- Stability   :  Experimental
9  -- Portability :
10 --
11 -- |
12 --
13 ------------------------------------------------------------------------------
14
15 module ForSyDe.Shallow.SDF.Signal (
16      Signal,
17      Packed,
18      signal,
19      fromSignal,
20      splitAt,
21      ---------
22      map,
23      zip,
24      expand,
25      delay,§
26      (>),§
27      (>>),§
28      (>),
29      (+>),⊙
30      (),⊕
31      (),⊗
32      (),Δ
33      (),
34 ) where
35
36 import Prelude (Int,(==),(<=),otherwise,(-),($),flip,foldr)
37 import ForSyDe.Shallow.SDF.Signal.Internal
38 import qualified Prelude as P
39
40 infixl 5 §> , §>> , §> , +> , ⊙ , ⊕ , ⊗ , Δ
41
42 ------------------------------------------------------------------------------
43
44 -- ⊙ : Unicode U+2A00 , UTF-8 Hex E2 A8 80⊙
45 () :: ([a] -> b) -> (Int,Signal a) -> Packed (Signal b)⊙
46 () = map
47
48 -- ⊕ : Unicode U+2A01 , UTF-8 Hex E2 A8 81⊕
49 () :: Packed (Signal ([a] -> b)) -> (Int,Signal a) -> Packed (Signal
       b)⊕
50 () = zip
51
52 -- ⊗ : Unicude U+2A02 , UTF-8 Hex ?? ?? ??⊗
53 () :: Packed (Signal [a]) -> () -> Signal a⊗
54 () = expand
55
56 -- Δ : Unicode U+2206 , UTF-8 Hex E2 88 86Δ
57 () :: Signal a -> [a] -> Signal aΔ
58 () = delay
59
60 ------------------------------------------------------------------------------§
61
```

```
62  (>) :: ([a] -> b) -> (Int,Signal a) -> Packed (Signal b)§
63  (>) = map§
64
65  (>>) :: Packed (Signal ([a] -> b)) -> (Int,Signal a) -> Packed (Signal
        b)§
66  (>>) = zip§
67
68  (>) :: Packed (Signal [a]) -> () -> Signal a§
69  (>) = expand
70
71  (+>) :: Signal a -> [a] -> Signal a
72  (+>) = delay
73
74  ------------------------------------------------------------------------
75
76  map :: ([a] -> b) -> (Int,Signal a) -> Packed (Signal b)
77  map f (nx,xs) = let (x,xs') = splitAt' nx xs in
78      if P.length x == nx then
79          Packed $ (f x) :- fromPacked (f `map` (nx,xs'))
80          else
81          Packed Null
82
83  zip :: Packed (Signal ([a] -> b)) -> (Int,Signal a) -> Packed (Signal b)
84  zip (Packed Null)       _          = Packed Null
85  zip _                   (_,Null) = Packed Null
86  zip (Packed (f :- fs)) (nx,xs)  = let (x,xs') = splitAt' nx xs in
87      if P.length x == nx then
88          Packed $ (f x) :- fromPacked (Packed fs `zip` (nx,xs'))
89          else
90          Packed Null
91
92  expand :: Packed (Signal [a]) -> () -> Signal a
93  expand (Packed Null)       () = Null
94  expand (Packed (x :- xs)) () = flip (foldr (:-)) x $ flip expand () $
        Packed xs
95
96  delay :: Signal a -> [a] -> Signal a
97  delay = foldr (:-)
98
99  ------------------------------------------------------------------------
100
101 signal :: [a] -> Signal a
102 signal (x : xs) = x :- signal xs
103 signal []       = Null
104
105 fromSignal :: Signal a -> [a]
106 fromSignal (x :- xs) = x : fromSignal xs
107 fromSignal Null      = []
108
109 ------------------------------------------------------------------------
110
111 splitAt :: Int -> Signal a -> (Signal a,Signal a)
112 splitAt n xs@(x :- xs')
113     | n <= 0    = (Null,xs)
114     | otherwise = let (as,bs) = splitAt (n-1) xs' in (x :- as,bs)
115 splitAt _ Null  = (Null,Null)
116
117 splitAt' :: Int -> Signal a -> ([a],Signal a)
```

```
118 splitAt' n xs@(x :- xs')
119     | n <= 0     = ([],xs)
120     | otherwise = let (as,bs) = splitAt' (n-1) xs' in (x : as,bs)
121 splitAt' _ Null  = ([],Null)
122
123 newtype Packed a = Packed a
124
125 fromPacked :: Packed a -> a
126 fromPacked (Packed x) = x
```

Process constructors

```
1  ------------------------------------------------------------------------------
2  --
3  -- Module      :  ForSyDe.Shallow.SDF.Processes
4  -- Copyright   :
5  -- License     :  AllRightsReserved
6  --
7  -- Maintainer  :  mkoe@imm.dtu.dk
8  -- Stability   :  Experimental
9  -- Portability :
10 --
11 -- |
12 --
13 ------------------------------------------------------------------------------
14
15 module ForSyDe.Shallow.SDF.Processes (
16
17 ) where
18
19 import Prelude ()
20 import ForSyDe.Shallow.SDF.Signal (Signal,(+>)§,(>)§,(>>))
```

Domain interface tools

```
1  ------------------------------------------------------------------------------
2  --
3  -- Module      :  ForSyDe.Shallow.SDF.Interface
4  -- Copyright   :
5  -- License     :  AllRightsReserved
6  --
7  -- Maintainer  :  mkoe@imm.dtu.dk
8  -- Stability   :  Experimental
9  -- Portability :
10 --
11 -- |
12 --
13 ------------------------------------------------------------------------------
14
15 module ForSyDe.Shallow.SDF.Interface (
16     head,
17     tail,
18     take,
19     drop,
20     length,
21 ) where
```

```
22
23 import Prelude (pred,(+),Int,(<=),otherwise,Maybe (..))
24 import ForSyDe.Shallow.SDF.Signal.Internal
25
26 head :: Signal a -> Maybe a
27 head Null      = Nothing
28 head (x :- _) = Just x
29
30 tail :: Signal a -> Maybe (Signal a)
31 tail Null      = Nothing
32 tail (_ :- xs) = Just xs
33
34 take :: Int -> Signal a -> Signal a
35 take n (x :- xs)
36     | n <= 0      = Null
37     | otherwise = x :- take (pred n) xs
38 take _ Null      = Null
39
40 drop :: Int -> Signal a -> Signal a
41 drop n (x :- xs)
42     | n <= 0      = x :- xs
43     | otherwise = drop (pred n) xs
44 drop _ Null      = Null
45
46 length :: Signal a -> Int
47 length (_ :- xs) = 1 + length xs
48 length Null      = 0
```

A.3 Discrete event model of computation

Signal definition

```
1  -----------------------------------------------------------------------------
2  --
3  -- Module      :  ForSyDe.Shallow.DE.Signal.Internal
4  -- Copyright   :
5  -- License     :  AllRightsReserved
6  --
7  -- Maintainer  :  mkoe@imm.dtu.dk
8  -- Stability   :  Experimental
9  -- Portability :
10 --
11 -- |
12 --
13 -----------------------------------------------------------------------------
14
15 module ForSyDe.Shallow.DE.Signal.Internal (
16     Signal (..),
17     SubSignal (..),
18 ) where
19
20 import Control.DeepSeq (NFData(rnf))
21
22 newtype Signal t a = Signal (a,SubSignal t a)
23 data SubSignal t a = (t,a) :- SubSignal t a | Null t
```

```
24
25  infixr 4 :-
26
27  instance (Show t,Show a) => Show (Signal t a) where
28      showsPrec p = showParen (p>1) . showSignal
29
30  showSignal :: (Show t,Show a) => Signal t a -> ShowS
31  showSignal (Signal (x0,xs)) = showChar '(' . shows x0 . showChar ',' .
        showSubSignal xs . showChar ')'
32
33  showSubSignal :: (Show t,Show a) => SubSignal t a -> ShowS
34  showSubSignal (x :- xs) = showChar '{' . showEvent x . showSubSignal' xs
35  showSubSignal (Null t)  = showString "{}" . shows t
36
37  showSubSignal' :: (Show t,Show a) => SubSignal t a -> ShowS
38  showSubSignal' (x :- xs) = showChar ',' . showEvent x . showSubSignal'
        xs
39  showSubSignal' (Null t)  = showChar '}' . shows t
40
41  -- Show an event token
42  showEvent :: (Show t,Show a) => (t,a) -> ShowS
43  showEvent (t,x) = showChar '(' . shows t . showChar ',' . shows x .
        showChar ')'
44
45  instance (Read t,Read a) => Read (Signal t a) where
46          readsPrec d = readParen (d>1) readSignal
47
48  readSignal :: (Read t,Read a) => ReadS (Signal t a)
49  readSignal = (\ a -> [(Signal (x0,xs),f) |
50      ("(",b) <- lex a,
51      (x0,c) <- reads b,
52      (",",d) <- lex c,
53      (xs,e) <- readSubSignalStart d,
54      (")",f) <- lex e
55      ])
56
57  readSubSignalStart :: (Read t,Read a) => ReadS (SubSignal t a)
58  readSubSignalStart = (\ a -> [(xs,c) |
59      ("{",b) <- lex a,
60      (xs,c) <- readSubSignal (',' : b) ++ readNull b
61      ])
62
63  readSubSignal :: (Read t,Read a) => ReadS (SubSignal t a)
64  readSubSignal r = readEvent r ++ readNull r
65
66  -- | Read an event token
67  readEvent :: (Read t,Read a) => ReadS (SubSignal t a)
68  readEvent a = [(x :- xs,e) | (",",b) <- lex a , (x,c) <- reads b ,
        (")",d) <- lex c, (xs,e) <- readSubSignal d]
69
70  -- | Read a null token
71  readNull :: (Read t,Read a) => ReadS (SubSignal t a)
72  readNull a = [(Null t,c) | ("}",b) <- lex a, (t,c) <- reads b]
73
74  instance (NFData t,NFData a) => NFData (Signal t a) where
75      rnf (Signal (x0,xs)) = rnf x0 `seq` rnf xs
76
77  instance (NFData t,NFData a) => NFData (SubSignal t a) where
```

```
78      rnf (x :- xs) = rnf x `seq` rnf xs
79      rnf (Null t)  = rnf t
```

Process atoms

```
1   ------------------------------------------------------------------------------
2   --
3   -- Module      :  ForSyDe.Shallow.DE.Signal
4   -- Copyright   :
5   -- License     :  AllRightsReserved
6   --
7   -- Maintainer  :  mkoe@imm.dtu.dk
8   -- Stability   :  Experimental
9   -- Portability :
10  --
11  -- |
12  --
13  ------------------------------------------------------------------------------
14
15  module ForSyDe.Shallow.DE.Signal (
16      Signal,
17      signal,
18      fromSignal,
19      map,
20      zip,
21      delay,§
22      (>),§
23      (>>),
24      (+>),⊙
25      (),⊕
26      (),Δ
27      (),
28  ) where
29
30  import Prelude ((<),(<=),(==),(+),min,otherwise,Ord,Num)
31  import ForSyDe.Shallow.DE.Signal.Internal
32
33  infixl 5 `map` , `zip` , `delay` , §> , §>> , +> , ⊙ , ⊕ , Δ
34
35  ------------------------------------------------------------------------------
36
37  -- ⊙ : Unicode U+2A00 , UTF-8 Hex E2 A8 80⊙
38  () :: (a -> b) -> Signal t a -> Signal t b⊙
39  () = map
40
41  -- ⊕ : Unicode U+2A01 , UTF-8 Hex E2 A8 81⊕
42  () :: (Ord t) => Signal t (a -> b) -> Signal t a -> Signal t b⊕
43  () = zip
44
45  -- Δ : Unicode U+2206 , UTF-8 Hex E2 88 86Δ
46  () :: (Num t) => Signal t a -> (a,t) -> Signal t aΔ
47  () = delay
48
49  ------------------------------------------------------------------------------§
50
51  (>) :: (a -> b) -> Signal t a -> Signal t b§
52  (>) = map§
```

```
53
54  (>>) :: (Ord t) => Signal t (a -> b) -> Signal t a -> Signal t b§
55  (>>) = zip
56
57  (+>) :: (Num t) => Signal t a -> (a,t) -> Signal t a
58  (+>) = delay
59
60  ------------------------------------------------------------------------
61
62  map :: (a -> b) -> Signal t a -> Signal t b
63  map f (Signal (x0,xs)) = Signal (f x0,map' f xs)
64
65  map' :: (a -> b) -> SubSignal t a -> SubSignal t b
66  map' f ((t,x) :- xs) = (t,f x) :- map' f xs
67  map' _ (Null t)      = Null t
68
69  zip :: (Ord t) => Signal t (a -> b) -> Signal t a -> Signal t b
70  zip (Signal (f0,fs)) (Signal (x0,xs)) = Signal (f0 x0,zip' f0 fs x0 xs)
71
72  zip' :: (Ord t) => (a -> b) -> SubSignal t (a -> b) -> a -> SubSignal t
        a -> SubSignal t b
73  zip' f0 fs@((tf,f) :- fs') x0 xs@((tx,x) :- xs')
74      | tf <  tx  = (tf,f  x0) :- zip' f  fs' x0 xs
75      | tf == tx  = (tf,f  x ) :- zip' f  fs' x  xs'
76      | otherwise = (tx,f0 x ) :- zip' f0 fs  x  xs'
77  zip' _      ((tf,f) :- fs') x0 xs@(Null tx)
78      | tf <  tx  = (tf,f  x0) :- zip' f  fs' x0 xs
79      | otherwise = Null tx
80  zip' f0 fs@(Null tf)      _      ((tx,x) :- xs')
81      | tf <= tx  = Null tf
82      | otherwise = (tx,f0 x ) :- zip' f0 fs  x  xs'
83  zip' _      (Null tf)     _      (Null tx)
84                  = Null (min tf tx)
85
86  delay :: (Num t) => Signal t a -> (a,t) -> Signal t a
87  delay (Signal (_,xs)) (x0,t) = Signal (x0,(0,x0) :- delay' xs t)
88
89  delay' :: (Num t) => SubSignal t a -> t -> SubSignal t a
90  delay' ((t,x) :- xs) dt = (t+dt,x) :- delay' xs t
91  delay' (Null t)      dt = Null (t+dt)
92
93  ------------------------------------------------------------------------
94
95  signal :: (Ord t) => a -> t -> [(t,a)] -> Signal t a
96  signal x0 te xs = Signal (x0,signal' te xs)
97
98  signal' :: (Ord t) => t -> [(t,a)] -> SubSignal t a
99  signal' te ((t,x) : xs)
100     | t <  te   = (t,x) :- signal' te xs
101     | otherwise = Null te
102 signal' te []   = Null te
103
104 fromSignal :: Signal t a -> (a,[(t,a)],t)
105 fromSignal (Signal (x0,xs)) = (x0,xs',te)
106     where
107     (xs',te) = fromSignal' xs
108
109 fromSignal' :: SubSignal t a -> ([(t,a)],t)
```

```
fromSignal' (x :- xs) = let (xs',te) = fromSignal' xs in (x : xs',te)
fromSignal' (Null t)  = ([],t)
```

Process constructors

```
-------------------------------------------------------------------------------
--
-- Module      :  ForSyDe.Shallow.DE.Processes
-- Copyright   :
-- License     :  AllRightsReserved
--
-- Maintainer  :  mkoe@imm.dtu.dk
-- Stability   :  Experimental
-- Portability :
--
-- |
--
-------------------------------------------------------------------------------

module ForSyDe.Shallow.DE.Processes (
--     map,
     zipWith,
     zipWith3,
     zipWith4,
     delay,
     zip,
     zip3,
     zip4,
     unzip,
     unzip3,
     unzip4,
) where

import Prelude (flip,Ord,Num,fst,snd)
import ForSyDe.Shallow.DE.Signal (Signal⊙,()⊕,()Δ,())

zipWith :: (Ord t) => (a -> b -> c) -> Signal t a -> Signal t b ->
     Signal t c
zipWith f xs ys = f ⊙ xs ⊕ ys

zipWith3 :: (Ord t) => (a -> b -> c -> d) -> Signal t a -> Signal t b
     -> Signal t c -> Signal t d
zipWith3 f xs ys zs = f ⊙ xs ⊕ ys ⊕ zs

zipWith4 :: (Ord t) => (a -> b -> c -> d -> e) -> Signal t a -> Signal
     t b -> Signal t c -> Signal t d -> Signal t e
zipWith4 f ws xs ys zs = f ⊙ ws ⊕ xs ⊕ ys ⊕ zs

delay :: (Num t) => (a,t) -> Signal t a -> Signal t a
delay = flip Δ()

zip :: (Ord t) => Signal t a -> Signal t b -> Signal t (a,b)
zip xs ys = (,) ⊙ xs ⊕ ys

zip3 :: (Ord t) => Signal t a -> Signal t b -> Signal t c -> Signal t
     (a,b,c)
zip3 xs ys zs = (,,) ⊙ xs ⊕ ys ⊕ zs
```

```
49
50 zip4 :: (Ord t) => Signal t a -> Signal t b -> Signal t c -> Signal t d
       -> Signal t (a,b,c,d)
51 zip4 ws xs ys zs = (,,,) ⊙ ws ⊕ xs ⊕ ys ⊕ zs
52
53 unzip :: Signal t (a,b) -> (Signal t a,Signal t b)
54 unzip xs = (fst ⊙ xs,snd ⊙ xs)
55
56 unzip3 :: Signal t (a,b,c) -> (Signal t a,Signal t b,Signal t c)
57 unzip3 xs = ((\ (x,_,_) -> x) ⊙ xs,(\ (_,x,_) -> x) ⊙ xs,(\ (_,_,x) ->
       x) ⊙ xs)
58
59 unzip4 :: Signal t (a,b,c,d) -> (Signal t a,Signal t b,Signal t
       c,Signal t d)
60 unzip4 xs = ((\ (x,_,_,_) -> x) ⊙ xs,(\ (_,x,_,_) -> x) ⊙ xs,(\
       (_,_,x,_) -> x) ⊙ xs,(\ (_,_,_,x) -> x) ⊙ xs)
```

Domain interface tools

```
1  ------------------------------------------------------------------------------
2  --
3  -- Module      :  ForSyDe.Shallow.DE.Interface
4  -- Copyright   :
5  -- License     :  AllRightsReserved
6  --
7  -- Maintainer  :  mkoe@imm.dtu.dk
8  -- Stability   :  Experimental
9  -- Portability :
10 --
11 -- |
12 --
13 ------------------------------------------------------------------------------
14
15 module ForSyDe.Shallow.DE.Interface (
16     head,
17     tail,
18     take,
19     drop,
20     length,
21 ) where
22
23 import Prelude hiding (head,tail,take,drop,length)
24 import ForSyDe.Shallow.DE.Signal.Internal
25
26 head :: Signal t a -> Maybe (t,a)
27 head (Signal (_,xs)) = head' xs
28
29 head' :: SubSignal t a -> Maybe (t,a)
30 head' (Null _) = Nothing
31 head' (x :- _) = Just x
32
33 tail :: Signal t a -> Maybe (Signal t a)
34 tail (Signal (x0,xs)) = case tail' xs of
35     Just xs' -> Just $ Signal (x0,xs')
36     Nothing  -> Nothing
37
38 tail' :: SubSignal t a -> Maybe (SubSignal t a)
```

```
39 tail' (Null _)  = Nothing
40 tail' (_ :- xs) = Just xs
41
42 take :: Int -> Signal t a -> Signal t a
43 take n (Signal (x0,xs)) = Signal (x0,take' n xs)
44
45 take' :: Int -> SubSignal t a -> SubSignal t a
46 take' n (x@(t,_) :- xs)
47     | n <= 0    = Null t
48     | otherwise = x :- take' (pred n) xs
49 take' _ (Null t) = Null t
50
51 drop :: Int -> Signal t a -> Signal t a
52 drop n (Signal (x0,xs)) = Signal (x0,drop' n xs)
53
54 drop' :: Int -> SubSignal t a -> SubSignal t a
55 drop' n (x :- xs)
56     | n <= 0    = x :- xs
57     | otherwise = drop' (pred n) xs
58 drop' _ (Null t) = Null t
59
60 length :: Signal t a -> Int
61 length (Signal (_,xs)) = length' xs
62
63 length' :: SubSignal t a -> Int
64 length' (_ :- xs) = 1 + length' xs
65 length' (Null _)  = 0
```

A.4 Continuous time model of computation

Signal definition

```
1  ------------------------------------------------------------------------------
2  --
3  -- Module      :  ForSyDe.Shallow.CT.Signal.Internal
4  -- Copyright   :
5  -- License     :  AllRightsReserved
6  --
7  -- Maintainer  :  mkoe@imm.dtu.dk
8  -- Stability   :  Experimental
9  -- Portability :
10 --
11 -- |
12 --
13 ------------------------------------------------------------------------------
14
15 module ForSyDe.Shallow.CT.Signal.Internal (
16     Signal (..),
17     SubSignal (..),
18 ) where
19
20 --import Control.DeepSeq (NFData(rnf))
21
22 newtype Signal t u a = Signal (u -> a,SubSignal t u a)
23 data SubSignal t u a = (t,u -> a) :- SubSignal t u a | Null t
```

```
24
25  infixr 4 :-
26
27  {-
28  instance (Show t,Show a) => Show (Signal t a) where
29       showsPrec p = showParen (p>1) . showSignal
30
31  showSignal :: (Show t,Show a) => Signal t a -> ShowS
32  showSignal (Signal (x0,xs)) = showChar '(' . shows x0 . showChar ',' .
         showSubSignal xs . showChar ')'
33
34  showSubSignal :: (Show t,Show a) => SubSignal t a -> ShowS
35  showSubSignal (x :- xs) = showChar '{' . showEvent x . showSubSignal' xs
36  showSubSignal (Null t)  = showString "{}" . shows t
37
38  showSubSignal' :: (Show t,Show a) => SubSignal t a -> ShowS
39  showSubSignal' (x :- xs) = showChar ',' . showEvent x . showSubSignal'
         xs
40  showSubSignal' (Null t)  = showChar '}' . shows t
41
42  -- Show an event token
43  showEvent :: (Show t,Show a) => (t,a) -> ShowS
44  showEvent (t,x) = showChar '(' . shows t . showChar ',' . shows x .
         showChar ')'
45  -}
46
47  {-
48  instance (Read t,Read a) => Read (Signal t a) where
49           readsPrec d = readParen (d>1) readSignal
50
51  readSignal :: (Read t,Read a) => ReadS (Signal t a)
52  readSignal = (\ a -> [(Signal (x0,xs),f) |
53       ("(",b) <- lex a,
54       (x0,c) <- reads b,
55       (",",d) <- lex c,
56       (xs,e) <- readSubSignalStart d,
57       (")",f) <- lex e
58       ])
59
60  readSubSignalStart :: (Read t,Read a) => ReadS (SubSignal t a)
61  readSubSignalStart = (\ a -> [(xs,c) |
62       ("{",b) <- lex a,
63       (xs,c) <- readSubSignal (',' : b) ++ readNull b
64       ])
65
66  readSubSignal :: (Read t,Read a) => ReadS (SubSignal t a)
67  readSubSignal r = readEvent r ++ readNull r
68
69  -- | Read an event token
70  readEvent :: (Read t,Read a) => ReadS (SubSignal t a)
71  readEvent a = [(x :- xs,e) | (",(",b) <- lex a , (x,c) <- reads b ,
         (")",d) <- lex c, (xs,e) <- readSubSignal d]
72
73  -- | Read a null token
74  readNull :: (Read t,Read a) => ReadS (SubSignal t a)
75  readNull a = [(Null t,c) | ("}",b) <- lex a, (t,c) <- reads b]
76  -}
77
```

```
{-
instance (NFData t, NFData a) => NFData (Signal t a) where
    rnf (Signal (x0, xs)) = rnf x0 `seq` rnf xs

instance (NFData t, NFData a) => NFData (SubSignal t a) where
    rnf (x :- xs) = rnf x `seq` rnf xs
    rnf (Null t)  = rnf t
-}
```

Process atoms

```
------------------------------------------------------------------------------
--
-- Module      :  ForSyDe.Shallow.CT.Signal
-- Copyright   :
-- License     :  AllRightsReserved
--
-- Maintainer  :  mkoe@imm.dtu.dk
-- Stability   :  Experimental
-- Portability :
--
-- |
--
------------------------------------------------------------------------------

module ForSyDe.Shallow.CT.Signal (
    Signal,
    signal,
    fromSignal,
    map,
    zip,
    delay,§
    (>),§
    (>>),
    (+>),⊙
    (),⊕
    (),Δ
    (),
) where

import Prelude ((<),(<=),(==),(+),min,otherwise,Ord,Num)
import ForSyDe.Shallow.CT.Signal.Internal

infixl 5 `map` , `zip` , `delay` , §> , §>> , +> , ⊙ , ⊕ , Δ

------------------------------------------------------------------------------

-- ⊙ : Unicode U+2A00 , UTF-8 Hex E2 A8 80⊙
() :: (a -> b) -> Signal t u a -> Signal t u b⊙
() = map

-- ⊕ : Unicode U+2A01 , UTF-8 Hex E2 A8 81⊕
() :: (Ord t) => Signal t u (a -> b) -> Signal t u a -> Signal t u b⊕
() = zip

-- Δ : Unicode U+2206 , UTF-8 Hex E2 88 86Δ
() :: (Num t) => Signal t u a -> (u -> a,t) -> Signal t u aΔ
```

```
47  () = delay
48
49  ------------------------------------------------------------------------§
50
51  (>) :: (a -> b) -> Signal t u a -> Signal t u b§
52  (>) = map§
53
54  (>>) :: (Ord t) => Signal t u (a -> b) -> Signal t u a -> Signal t u b§
55  (>>) = zip
56
57  (+>) :: (Num t) => Signal t u a -> (u -> a,t) -> Signal t u a
58  (+>) = delay
59
60  ------------------------------------------------------------------------
61
62  map :: (a -> b) -> Signal t u a -> Signal t u b
63  map f (Signal (x0,xs)) = Signal (\u -> f (x0 u),map' f xs)
64
65  map' :: (a -> b) -> SubSignal t u a -> SubSignal t u b
66  map' f ((t,x) :- xs) = (t,\u -> f (x u)) :- map' f xs
67  map' _ (Null t)      = Null t
68
69  zip :: (Ord t) => Signal t u (a -> b) -> Signal t u a -> Signal t u b
70  zip (Signal (f0,fs)) (Signal (x0,xs)) = Signal (\u -> f0 u (x0 u),zip'
        f0 fs x0 xs)
71
72  zip' :: (Ord t) => (u -> a -> b) -> SubSignal t u (a -> b) -> (u -> a)
        -> SubSignal t u a -> SubSignal t u b
73  zip' f0 fs@((tf,f) :- fs') x0 xs@((tx,x) :- xs')
74      | tf <  tx  = (tf,\u -> f  u (x0 u)) :- zip' f  fs' x0 xs
75      | tf == tx  = (tf,\u -> f  u (x  u)) :- zip' f  fs' x  xs'
76      | otherwise = (tx,\u -> f0 u (x  u)) :- zip' f0 fs  x  xs'
77  zip' _      ((tf,f) :- fs') x0 xs@(Null tx)
78      | tf <  tx  = (tf,\u -> f  u (x0 u)) :- zip' f  fs' x0 xs
79      | otherwise = Null tx
80  zip' f0 fs@(Null tf)      _      ((tx,x) :- xs')
81      | tf <= tx  = Null tf
82      | otherwise = (tx,\u -> f0 u (x  u)) :- zip' f0 fs  x  xs'
83  zip' _      (Null tf)     _      (Null tx)
84                = Null (min tf tx)
85
86  delay :: (Num t) => Signal t u a -> (u -> a,t) -> Signal t u a
87  delay (Signal (_,xs)) (x0,t) = Signal (x0,(0,x0) :- delay' xs t)
88
89  delay' :: (Num t) => SubSignal t u a -> t -> SubSignal t u a
90  delay' ((t,x) :- xs) dt = (t+dt,x) :- delay' xs t
91  delay' (Null t)      dt = Null (t+dt)
92
93  ------------------------------------------------------------------------
94
95  signal :: (Ord t) => (u -> a) -> t -> [(t,u -> a)] -> Signal t u a
96  signal x0 te xs = Signal (x0,signal' te xs)
97
98  signal' :: (Ord t) => t -> [(t,u -> a)] -> SubSignal t u a
99  signal' te ((t,x) : xs)
100     | t <  te   = (t,x) :- signal' te xs
101     | otherwise = Null te
102 signal' te []   = Null te
```

```
103
104 fromSignal :: Signal t u a -> (u -> a,[(t,u -> a)],t)
105 fromSignal (Signal (x0,xs)) = (x0,xs',te)
106     where
107     (xs',te) = fromSignal' xs
108
109 fromSignal' :: SubSignal t u a -> ([(t,u -> a)],t)
110 fromSignal' (x :- xs) = let (xs',te) = fromSignal' xs in (x : xs',te)
111 fromSignal' (Null t)  = ([],t)
```

Process constructors

```
1  ------------------------------------------------------------------------------
2  --
3  -- Module      :  ForSyDe.Shallow.CT.Processes
4  -- Copyright   :
5  -- License     :  AllRightsReserved
6  --
7  -- Maintainer  :  mkoe@imm.dtu.dk
8  -- Stability   :  Experimental
9  -- Portability :
10 --
11 -- |
12 --
13 ------------------------------------------------------------------------------
14
15 module ForSyDe.Shallow.CT.Processes (
16 --     map,
17     zipWith,
18     zipWith3,
19     zipWith4,
20     delay,
21     zip,
22     zip3,
23     zip4,
24     unzip,
25     unzip3,
26     unzip4,
27 ) where
28
29 import Prelude (flip,Ord,Num,fst,snd)
30 import ForSyDe.Shallow.CT.Signal (Signal⊙,()⊕,()Δ,())
31
32 zipWith :: (Ord t) ⇒ (a -> b -> c) -> Signal t u a -> Signal t u b ->
       Signal t u c
33 zipWith f xs ys = f ⊙ xs ⊕ ys
34
35 zipWith3 :: (Ord t) ⇒ (a -> b -> c -> d) -> Signal t u a -> Signal t u
       b -> Signal t u c -> Signal t u d
36 zipWith3 f xs ys zs = f ⊙ xs ⊕ ys ⊕ zs
37
38 zipWith4 :: (Ord t) ⇒ (a -> b -> c -> d -> e) -> Signal t u a ->
       Signal t u b -> Signal t u c -> Signal t u d -> Signal t u e
39 zipWith4 f ws xs ys zs = f ⊙ ws ⊕ xs ⊕ ys ⊕ zs
40
41 delay :: (Num t) ⇒ (u -> a,t) -> Signal t u a -> Signal t u a
42 delay = flip Δ()
```

```
43
44 zip :: (Ord t) => Signal t u a -> Signal t u b -> Signal t u (a,b)
45 zip xs ys = (,) ⊙ xs ⊕ ys
46
47 zip3 :: (Ord t) => Signal t u a -> Signal t u b -> Signal t u c ->
       Signal t u (a,b,c)
48 zip3 xs ys zs = (,,) ⊙ xs ⊕ ys ⊕ zs
49
50 zip4 :: (Ord t) => Signal t u a -> Signal t u b -> Signal t u c ->
       Signal t u d -> Signal t u (a,b,c,d)
51 zip4 ws xs ys zs = (,,,) ⊙ ws ⊕ xs ⊕ ys ⊕ zs
52
53 unzip :: Signal t u (a,b) -> (Signal t u a,Signal t u b)
54 unzip xs = (fst ⊙ xs,snd ⊙ xs)
55
56 unzip3 :: Signal t u (a,b,c) -> (Signal t u a,Signal t u b,Signal t u c)
57 unzip3 xs = ((\ (x,_,_) -> x) ⊙ xs,(\ (_,x,_) -> x) ⊙ xs,(\ (_,_,x) ->
       x) ⊙ xs)
58
59 unzip4 :: Signal t u (a,b,c,d) -> (Signal t u a,Signal t u b,Signal t u
       c,Signal t u d)
60 unzip4 xs = ((\ (x,_,_,_) -> x) ⊙ xs,(\ (_,x,_,_) -> x) ⊙ xs,(\
       (_,_,x,_) -> x) ⊙ xs,(\ (_,_,_,x) -> x) ⊙ xs)
```

Domain interface tools

```
1  ------------------------------------------------------------------------------
2  --
3  -- Module      :  ForSyDe.Shallow.CT.Interface
4  -- Copyright   :
5  -- License     :  AllRightsReserved
6  --
7  -- Maintainer  :  mkoe@imm.dtu.dk
8  -- Stability   :  Experimental
9  -- Portability :
10 --
11 -- |
12 --
13 ------------------------------------------------------------------------------
14
15 module ForSyDe.Shallow.CT.Interface (
16     head,
17     tail,
18     take,
19     drop,
20     length,
21 ) where
22
23 import Prelude hiding (head,tail,take,drop,length)
24 import ForSyDe.Shallow.CT.Signal.Internal
25
26 head :: Signal t u a -> Maybe (t,u -> a)
27 head (Signal (_,xs)) = head' xs
28
29 head' :: SubSignal t u a -> Maybe (t,u -> a)
30 head' (Null _) = Nothing
31 head' (x :- _) = Just x
```

```
32
33 tail :: Signal t u a -> Maybe (Signal t u a)
34 tail (Signal (x0,xs)) = case tail' xs of
35     Just xs' -> Just $ Signal (x0,xs')
36     Nothing  -> Nothing
37
38 tail' :: SubSignal t u a -> Maybe (SubSignal t u a)
39 tail' (Null _)  = Nothing
40 tail' (_ :- xs) = Just xs
41
42 take :: Int -> Signal t u a -> Signal t u a
43 take n (Signal (x0,xs)) = Signal (x0,take' n xs)
44
45 take' :: Int -> SubSignal t u a -> SubSignal t u a
46 take' n (x@(t,_) :- xs)
47     | n <= 0    = Null t
48     | otherwise = x :- take' (pred n) xs
49 take' _ (Null t) = Null t
50
51 drop :: Int -> Signal t u a -> Signal t u a
52 drop n (Signal (x0,xs)) = Signal (x0,drop' n xs)
53
54 drop' :: Int -> SubSignal t u a -> SubSignal t u a
55 drop' n (x :- xs)
56     | n <= 0    = x :- xs
57     | otherwise = drop' (pred n) xs
58 drop' _ (Null t) = Null t
59
60 length :: Signal t u a -> Int
61 length (Signal (_,xs)) = length' xs
62
63 length' :: SubSignal t u a -> Int
64 length' (_ :- xs) = 1 + length' xs
65 length' (Null _)  = 0
```

A.5 UPPAAL model code

Circle of four nodes

```
1 <?xml version="1.0" encoding="utf-8"?><!DOCTYPE nta PUBLIC '-//Uppaal
      Team//DTD Flat System 1.1//EN'
      'http://www.it.uu.se/research/group/darts/uppaal/flat-1_1.dtd'><nta><declaration>
      Place global declarations here.
2
3 clock global_time;
4 const int Separation = 2;
5 const int ApplicationStartDelay = 5;
6
7 const int Nodes = 5;
8 const int Sinks = 1;
9 const int Neighbours = 4;
10 const int Decimals = 10;
11 const int MIN_AUGMENTATION = 1 * Decimals;
12 const int BatteryCapacity = 42 * Decimals;
13 const int MinBatteryCharge = 10;
```

```
14 const int MaxDepletion = 50 * Decimals;
15 const int UpdateThreshold = Decimals / 10;
16 const int MaxAugmentation = MaxDepletion + Nodes * Decimals;
17
18 const int ApplicationInterrupt = 60;
19 const int EnergyUpdateInterrupt = 60;
20 const int RouteDelay = 1;
21
22 typedef int[0,Nodes-1] id_t;
23 typedef int[0,Sinks-1] sinks_t;
24 typedef int[Sinks,Nodes-1] nodes_t;
25 typedef int[0,Neighbours-1] neighbours_t;
26 typedef int[0,Nodes] height_t;
27 typedef int[0,BatteryCapacity] energy_t;
28 typedef int[0,MaxDepletion] depletion_t;
29 typedef int[0,MaxAugmentation] augmentation_t;
30 typedef int[0,Nodes*Decimals+MaxDepletion+MaxAugmentation] total_t;
31
32 energy_t battery[nodes_t];
33 meta bool outOfPower = false;
34
35 meta int[0,Nodes] initialiseNode = 0;
36
37 chan data[Nodes];
38 broadcast chan update[Nodes];
39
40 meta total_t height_update;
41 meta id_t data_source;
42
43 const energy_t ApplicationUsage = 1;
44 const energy_t RouteUsage = 1;
45 const energy_t UpdateEnergyUsage = 1;
46 const energy_t UpdateNeighbourUsage = 1;
47 const energy_t UpdateUsage = 1;
48
49 const int numNeighbours[Nodes] = {
50         2,2,2,2,2
51 };
52
53 const id_t neighbours[Nodes][Neighbours] = {
54         {1,4,0,0},
55         {0,2,0,0},
56         {1,3,0,0},
57         {2,4,0,0},
58         {3,0,0,0}
59 };
60
61 const int hops[Nodes] = {
62         0,1,2,2,1
63 };
64
65 const int ChargeInterval = 1;
66
67 const int insolation_items = 2;
68 const energy_t insolation_value[insolation_items] = {
69         1,
70         0
71 };
```

```
72  const int insolation_time[insolation_items] = {
73          12 * 60/* * 60*/,
74          12 * 60/* * 60*/
75  };
76  int[0,insolation_items] insolation_index;
77
78  const int shadow_items = 1;
79  const int[0,100] shadow_value[Nodes][shadow_items] = {
80          {100},
81          {100},
82          {100},
83          {100},
84          {100}
85  };
86  const int shadow_time[shadow_items] = {
87          60 * 60
88  };
89  int[0,shadow_items] shadow_index;
90
91  void consume(const nodes_t id, const energy_t energy) {
92          if (battery[id] &lt; energy) {
93                  battery[id] = 0;
94          } else {
95                  battery[id] -= energy;
96          }
97  }
98
99  int getSunStep() {
100         return insolation_time[insolation_index];
101 }
102
103 void chargeAll() {
104         for (id : nodes_t) {
105                 const int energy = insolation_value[insolation_index] *
                        shadow_value[id][shadow_index] * ChargeInterval /
                        100 + battery[id];
106                 if (energy &gt; BatteryCapacity) {
107                         battery[id] = BatteryCapacity;
108                 } else {
109                         battery[id] = energy;
110                 }
111         }
112 }
113
114 int getShadowStep() {
115         return shadow_time[shadow_index];
116 }
117
118 void updateSun() {
119         insolation_index++;
120         if (insolation_index == insolation_items) {
121                 insolation_index = 0;
122         }
123 }
124
125 void updateShadow() {
126         shadow_index++;
127         if (shadow_index == shadow_items) {
```

```
128                 shadow_index = 0;
129         }
130 }
131
132 depletion_t getDepletion(const energy_t energy) {
133         if (energy &gt;= (9 * BatteryCapacity) / 10) {
134                 return 0;
135         } else if (energy &gt;= (3 * BatteryCapacity) / 4) {
136                 return MaxDepletion/10 * (energy - (9 *
                        BatteryCapacity) / 10) / ((3 * BatteryCapacity) /
                        4 - (9 * BatteryCapacity) / 10);
137         } else if (energy &gt;= (1 * BatteryCapacity) / 4) {
138                 return (9*MaxDepletion)/10 * (energy - (3 *
                        BatteryCapacity) / 4) / ((1 * BatteryCapacity) / 4
                        - (3 * BatteryCapacity) / 4);
139         } else {
140                 return MaxDepletion;
141         }
142 }
143
144 const int QMAX = Nodes;
145 const int infinity = 500;
146 /**
147         Calculate the global optimal route (neighbour) for the given
            node &lt;id&gt;.
148         Based on battery power of all nodes and shortest path only.
149         Disregard any local node parameters such as augmentation.
150         Currently assuming that id=0 is the base station.
151 */
152 id_t globalOptimalRoute(const id_t id) {
153         /** Optimal height (energy+distance to sink) */
154         int H[Nodes];
155         /** Queue of nodes (values: id_t or NA) to search, with
                pointers for insertion and extraction */
156         int Q[QMAX], Qput, Qget;
157         /** Copy of batteries, adding one for each base station */
158         energy_t E[id_t];
159         /** Initialise */
160         Qput = 0;
161         Qget = 0;
162         for (sink : sinks_t) {
163                 if (id == sink) {
164                         return sink;
165                 }
166                 E[sink] = BatteryCapacity;
167         }
168         for (n : nodes_t) {
169                 E[n] = battery[n];
170         }
171         for (n : id_t) {
172                 /** Initialise all nodes to have a height of "infinity"
                        */
173                 H[n] = infinity;
174         }
175         /** Starting parameters */
176         for (sink : sinks_t) {
177                 H[sink] = 0;
178                 Q[Qput] = sink;
```

```
179                 Qput++;
180         }
181         /** Algorithm */
182         while (Qput != Qget) {
183                 int n;
184                 /** Get next node in queue */
185                 const id_t node = Q[Qget];
186                 Qget++;
187                 if (Qget == QMAX) {
188                         Qget = 0;
189                 }
190                 /** Add neighbours to queue */
191                 for (n = 0; n &lt; numNeighbours[node]; ++n) {
192                         const id_t nid = neighbours[node][n];
193                         const int h0 = H[node];
194                         const int h1 = H[nid];
195                         const int h2 = hops[nid] * Decimals +
                                getDepletion(E[nid]);
196                         if (h0 + MIN_AUGMENTATION &lt; h1 &amp;&amp; h1
                                != h2) {
197                                 /** Add neighbour to queue */
198                                 Q[Qput] = nid;
199                                 Qput++;
200                                 if (Qput == QMAX) {
201                                         Qput = 0;
202                                 }
203                                 /** Calculate new height of added
                                        neighbour */
204                                 if (h2 &lt;= h0) {
205                                         H[nid] = h0 + MIN_AUGMENTATION;
206                                 } else {
207                                         H[nid] = h2;
208                                 }
209                         }
210                 }
211         }
212         /** Find lovest neighbour of given node */
213         {
214                 int i;
215                 /** ID of current candidate */
216                 meta id_t l = neighbours[id][0];
217                 /** Height of current candidate */
218                 meta total_t h = H[l];
219                 /** Find best neighbour of given node */
220                 for (i = 1; i &lt; numNeighbours[id]; ++i) {
221                         const id_t nid = neighbours[id][i];
222                         if (H[nid] &lt; h) {
223                                 l = nid;
224                                 h = H[nid];
225                         }
226                 }
227                 return l;
228         }
229 }
230
231 id_t shortestPathNeighbour(const nodes_t id) {
232         meta id_t x = neighbours[id][0];
233         meta int h = hops[x];
```

```
234         for (n : neighbours_t) if (n &lt; numNeighbours[id]) {
235                 const id_t nid = neighbours[id][n];
236                 if (hops[nid] &lt; h) {
237                         x = nid;
238                         h = hops[nid];
239                 }
240         }
241         return x;
242 }
243
244 </declaration><template><name x="5" y="5">Node</name><parameter>const
        id_t id</parameter><declaration>// Place local declarations here.
245 clock application, protocol, route_delay;
246
247 id_t route;
248 meta augmentation_t augmentation;
249 meta total_t height[Neighbours];
250 meta int buffered_packages;
251 /** Last Distributed Height*/
252 meta total_t ldh;
253
254 void initialise() {
255         for (i : int[0,numNeighbours[id]-1]) {
256                 height[i] = hops[neighbours[id][i]] * Decimals;
257         }
258         battery[id] = BatteryCapacity;
259         buffered_packages = 0;
260 }
261
262 total_t getHeight() {
263         return (hops[id] * Decimals + augmentation +
              getDepletion(battery[id]));
264 }
265
266 id_t getOptimalRoute() {
267         meta int[1,numNeighbours[id]] i;
268         /** ID of current candidate */
269         meta id_t l = neighbours[id][0];
270         /** Find lowest neighbour */
271         meta total_t h = height[0];
272         /** Find lowest neighbour */
273         for (i = 1; i &lt; numNeighbours[id]; ++i) {
274                 if (height[i] &lt; h) {
275                         l = neighbours[id][i];
276                         h = height[i];
277                 }
278         }
279         return l;
280 }
281
282 void updateRoute() {
283         meta int[1,numNeighbours[id]] i;
284         /** Height of current candidate */
285         meta total_t h = height[0];
286         /** ID of current candidate */
287         meta id_t l = neighbours[id][0];
288         /** Find lowest neighbour */
289         for (i = 1; i &lt; numNeighbours[id]; ++i) {
```

```
290                     if (height[i] &lt; h) {
291                             l = neighbours[id][i];
292                             h = height[i];
293                     }
294             }
295             route = l;
296     }
297
298     augmentation_t getNeededAugmentation() {
299             meta int temp;
300             meta int[1,numNeighbours[id]] i;
301             /** Height of current candidate */
302             meta total_t h = height[0];
303             /** Find lowest neighbour */
304             for (i = 1; i &lt; numNeighbours[id]; ++i) {
305                     if (height[i] &lt; h) {
306                             h = height[i];
307                     }
308             }
309             temp = MIN_AUGMENTATION + h - ((hops[id] * Decimals +
                        augmentation + getDepletion(battery[id])) - augmentation);
310             if (temp &lt; 0) {
311                     return 0;
312             } else {
313                     return temp;
314             }
315     }
316
317     void updateAugmentation() {
318             meta int temp;
319             meta int[1,numNeighbours[id]] i;
320             /** Height of current candidate */
321             meta total_t h = height[0];
322             ///** ID of current candidate */
323             //meta id_t l = neighbours[id][0];
324             /** Find lowest neighbour */
325             for (i = 1; i &lt; numNeighbours[id]; ++i) {
326                     if (height[i] &lt; h) {
327                             //l = neighbours[id][i];
328                             h = height[i];
329                     }
330             }
331             /** Calculate augmentation */
332             //temp = MIN_AUGMENTATION + h - (getHeight() - augmentation);
333             temp = MIN_AUGMENTATION + h - ((hops[id] * Decimals +
                        augmentation + getDepletion(battery[id])) - augmentation);
334             if (temp &lt; 0) {
335                     augmentation = 0;
336             } else {
337                     augmentation = temp;
338             }
339     }
340
341     bool hasValidRoute() {
342             //for (i : int[0,numNeighbours[id]-1]) {
343             int i;
344             for (i = 0; i &lt; numNeighbours[id]; ++i) {
345                     //if (height[i] &lt; getHeight()) {
```

```
346                 if (height[i] &lt; (hops[id] * Decimals + augmentation
                        + getDepletion(battery[id]))) {
347                     return true;
348                 }
349         }
350         return false;
351 }
352
353 bool hasOptimalRoute() {
354         meta total_t h = height[0];
355         meta id_t node = neighbours[id][0];
356         for (n : neighbours_t) if (n &lt; numNeighbours[id] &amp;amp;&amp;amp;
                n != 0) {
357             if (height[n] &lt; h) {
358                     node = neighbours[id][n];
359                     h = height[n];
360             }
361         }
362         return (route == node);
363 }
364
365 bool thresholdPassed() {
366         return (ldh - getHeight()) &gt; UpdateThreshold or (getHeight()
                - ldh) &gt; UpdateThreshold;
367 }
368
369 bool isUpdateNeeded() {
370         return (not hasValidRoute()) or (not hasOptimalRoute() and
                thresholdPassed());
371 }
372 </declaration><location id="id0" x="272" y="-200"><name x="184"
    y="-216">RouteSend</name><committed/></location><location id="id1"
    x="272" y="192"><name x="176" y="176">OutOfPower</name><label
    kind="invariant" x="40" y="192">battery[id] &lt;=
    MinBatteryCharge</label></location><location id="id2" x="-360"
    y="24"><name x="-360"
    y="-16">UpdateNeighbour</name><committed/></location><location
    id="id3" x="-640" y="40"><name x="-624"
    y="24">UpdateEnergy</name><committed/></location><location
    id="id4" x="248" y="-168"><name x="144"
    y="-168">RouteReceive</name><committed/></location><location
    id="id5" x="224" y="-120"><name x="144"
    y="-104">Application</name><committed/></location><location
    id="id6" x="-80" y="-64"><name x="-112" y="-144">Idle</name><label
    kind="invariant" x="-352" y="-200">application &lt;=
    ApplicationInterrupt
373 &amp;amp;&amp;amp; protocol &lt;= EnergyUpdateInterrupt
374 &amp;amp;&amp;amp; (
375   buffered_packages == 0
376   || route_delay &lt;= RouteDelay
377 )</label><label kind="comments">Node is
    idle</label></location><location id="id7" x="-640" y="-200"><name
    x="-616" y="-216">Init</name><label kind="invariant" x="-592"
    y="-216">global_time &lt;= id * Separation</label></location><init
    ref="id7"/><transition><source ref="id3"/><target
    ref="id6"/><label kind="guard" x="-640" y="128">not
    isUpdateNeeded()</label><label kind="comments">No broadcast of
    energy update needed, energy has not changed enough since last
```

```
        broadcast</label><nail x="-640" y="200"/><nail x="-80"
        y="200"/></transition><transition><source ref="id1"/><target
        ref="id6"/><label kind="guard" x="48" y="56">battery[id] &gt;
        MinBatteryCharge</label><label kind="synchronisation" x="48"
        y="72">update[id]!</label><label kind="assignment" x="48"
        y="88">updateAugmentation(),
378   ldh = getHeight(),
379   height_update = ldh,
380   outOfPower = false,
381   application = 0,
382   protocol = 0,
383   route_delay = 0</label><label kind="comments">The node is recharged
        enough to reboot</label><nail x="40" y="192"/><nail x="40"
        y="-48"/><nail x="-56" y="-48"/></transition><transition><source
        ref="id2"/><target ref="id2"/><label kind="select" x="-624"
        y="-53">s : neighbours_t</label><label kind="guard" x="-624"
        y="-38">s &lt; numNeighbours[id]</label><label
        kind="synchronisation" x="-624"
        y="-23">update[neighbours[id][s]]?</label><label kind="assignment"
        x="-624" y="-8">height[s] = height_update,
384   consume(id, UpdateNeighbourUsage)</label><label kind="comments">Receive
        an energy update from a neighbour</label><nail x="-632"
        y="24"/><nail x="-632" y="-56"/><nail x="-368" y="-56"/><nail
        x="-368" y="-8"/></transition><transition><source
        ref="id5"/><target ref="id4"/><label kind="assignment" x="112"
        y="-152">data_source = id</label><label kind="comments">Send the
        data to sink which the application
        produces</label></transition><transition><source
        ref="id0"/><target ref="id6"/><label kind="synchronisation"
        x="-56" y="-216">data[route]!</label><label kind="assignment"
        x="24" y="-216">data_source = id,
385   buffered_packages--,
386   route_delay = 0</label><label kind="comments">Send data to the current
        best neighbour towards sink</label><nail x="-72" y="-200"/><nail
        x="-72" y="-96"/></transition><transition><source
        ref="id6"/><target ref="id0"/><label kind="guard" x="72"
        y="-64">route_delay &amp;gt;= RouteDelay
387   &amp;amp;&amp;amp; buffered_packages != 0</label><label kind="assignment"
        x="72" y="-32">consume(id, RouteUsage)</label><label
        kind="comments">Send waiting data towards sink</label><nail
        x="272" y="-64"/></transition><transition><source
        ref="id2"/><target ref="id6"/><label kind="guard" x="-312"
        y="16">isUpdateNeeded()</label><label kind="synchronisation"
        x="-176" y="16">update[id]!</label><label kind="assignment"
        x="-312" y="32">updateAugmentation(),
388   ldh = getHeight(),
389   height_update = ldh,
390   route = getOptimalRoute(),
391   consume(id, UpdateUsage)</label><label kind="comments">Broadcast an
        energy update of this node</label><nail x="-328" y="16"/><nail
        x="-96" y="16"/><nail x="-96"
        y="-40"/></transition><transition><source ref="id6"/><target
        ref="id1"/><label kind="guard" x="48" y="0">battery[id] &lt;=
        MinBatteryCharge</label><label kind="assignment" x="136"
        y="16">outOfPower = true,
392   application = 0</label><label kind="comments">The node runs out of
        power</label><nail x="-48" y="-56"/><nail x="48" y="-56"/><nail
        x="48" y="0"/><nail x="272"
```

```
y="0"/></transition><transition><source ref="id2"/><target
ref="id6"/><label kind="guard" x="-312" y="0">not
isUpdateNeeded()</label><label kind="comments">Accept current best
route as optimal</label><nail x="-344" y="0"/><nail x="-104"
y="0"/><nail x="-104" y="-48"/></transition><transition><source
ref="id6"/><target ref="id2"/><label kind="select" x="-360"
y="-96">s : neighbours_t</label><label kind="guard" x="-360"
y="-80">s &amp;lt; numNeighbours[id]</label><label
kind="synchronisation" x="-360"
y="-64">update[neighbours[id][s]]?</label><label kind="assignment"
x="-360" y="-48">height[s] = height_update,
393 consume(id, UpdateNeighbourUsage)</label><label kind="comments">Receive
an energy update from a neighbour</label><nail x="-96"
y="-88"/><nail x="-104" y="-96"/><nail x="-360"
y="-96"/></transition><transition><source ref="id3"/><target
ref="id6"/><label kind="guard" x="-576"
y="40">isUpdateNeeded()</label><label kind="synchronisation"
x="-448" y="40">update[id]!</label><label kind="assignment"
x="-576" y="56">updateAugmentation(),
394 ldh = getHeight(),
395 height_update = ldh,
396 consume(id, UpdateUsage)</label><label kind="comments">Broadcast an
energy update of this node</label><nail x="-368" y="40"/><nail
x="-368" y="192"/><nail x="-88" y="192"/><nail x="-88"
y="-32"/></transition><transition><source ref="id6"/><target
ref="id3"/><label kind="guard" x="-632" y="-104">protocol ==
EnergyUpdateInterrupt</label><label kind="assignment" x="-632"
y="-88">consume(id, UpdateEnergyUsage),
397 protocol = 0</label><label kind="comments">Run protocol energy update
on interrupt</label><nail x="-88" y="-96"/><nail x="-96"
y="-104"/><nail x="-640"
y="-104"/></transition><transition><source ref="id4"/><target
ref="id6"/><label kind="assignment" x="-56"
y="-168">buffered_packages++,
398 route_delay = 0</label><label kind="comments">Register the received
data for routing towards sink</label><nail x="-64" y="-168"/><nail
x="-64" y="-88"/></transition><transition><source
ref="id6"/><target ref="id4"/><label kind="synchronisation"
x="144" y="-88">data[id]?</label><label kind="assignment" x="-40"
y="-88">consume(id, RouteUsage)</label><label
kind="comments">Receive data from a neighbour</label><nail x="-48"
y="-72"/><nail x="248" y="-72"/></transition><transition><source
ref="id6"/><target ref="id5"/><label kind="guard" x="-56"
y="-136">application == ApplicationInterrupt</label><label
kind="assignment" x="-56" y="-120">consume(id, ApplicationUsage),
399 application = 0</label><label kind="comments">Run the application on
interrupt</label><nail x="-56" y="-80"/><nail x="-48"
y="-88"/><nail x="224" y="-88"/></transition><transition><source
ref="id7"/><target ref="id6"/><label kind="guard" x="-616"
y="-200">global_time == id * Separation
400 &amp;amp;&amp;amp; initialiseNode == id</label><label kind="assignment"
x="-648" y="-176">initialise(),
401 updateRoute(),
402 application = ApplicationStartDelay,
403 protocol = 0,
404 initialiseNode++</label><label kind="comments">Initialise the node
status (route and battery energy)</label><nail x="-80"
y="-200"/></transition></template><template><name>BaseStation</name><parameter>const
```

```
      id_t id</parameter><location id="id8" x="-264" y="40"><name
      x="-272" y="8">Collect</name><committed/></location><location
      id="id9" x="-64" y="40"><name x="-104"
      y="0">Idle</name></location><location id="id10" x="-64"
      y="-80"><name x="-296" y="-104">Init</name><label kind="invariant"
      x="-296" y="-88">global_time &lt;= id *
      Separation</label></location><init ref="id10"/><transition><source
      ref="id8"/><target ref="id9"/><nail x="-224" y="48"/><nail
      x="-104" y="48"/></transition><transition><source
      ref="id9"/><target ref="id8"/><label kind="synchronisation"
      x="-200" y="16">data[id]?</label><nail x="-104" y="32"/><nail
      x="-224" y="32"/></transition><transition><source
      ref="id10"/><target ref="id9"/><label kind="guard" x="-296"
      y="-72">global_time == id * Separation
405 &amp;&amp; initialiseNode == id</label><label kind="assignment"
      x="-296"
      y="-40">initialiseNode++</label></transition></template><template><name>Environme
      sun, shadow, step;
406 </declaration><location id="id11" x="376" y="-120"><name x="400"
      y="-128">ChargeNodes</name><committed/></location><location
      id="id12" x="-360" y="-120"><name x="-376"
      y="-104">Initialise</name><label kind="invariant" x="-376"
      y="-88">global_time &lt;= Nodes *
      Separation</label></location><location id="id13" x="-72"
      y="80"><name x="-56"
      y="72">SwitchShadows</name><committed/></location><location
      id="id14" x="-72" y="-352"><name x="-56"
      y="-360">SwitchSun</name><committed/></location><location
      id="id15" x="-72" y="-120"><name x="-64"
      y="-152">Idle</name><label kind="invariant" x="-32" y="-200">sun
      &lt;= getSunStep() &amp;&amp;
407 shadow &lt;= getShadowStep()
408 &amp;&amp; (step &lt;= ChargeInterval ||
409 insolation_value[insolation_index] == 0)</label></location><init
      ref="id12"/><transition><source ref="id11"/><target
      ref="id15"/><label kind="assignment" x="-8"
      y="-104">chargeAll()</label><nail x="344" y="-104"/><nail x="-40"
      y="-104"/></transition><transition><source ref="id15"/><target
      ref="id11"/><label kind="guard" x="-8" y="-136">step ==
      ChargeInterval &amp;&amp;
410 insolation_value[insolation_index] != 0</label><label kind="assignment"
      x="200" y="-136">step = 0</label></transition><transition><source
      ref="id14"/><target ref="id15"/><label kind="assignment" x="-176"
      y="-248">updateSun()</label><nail x="-88" y="-320"/><nail x="-88"
      y="-152"/></transition><transition><source ref="id12"/><target
      ref="id15"/><label kind="guard" x="-344" y="-136">global_time ==
      Nodes * Separation
411 &amp;&amp; initialiseNode == Nodes</label><label kind="assignment"
      x="-256" y="-176">step = 0,
412 sun = 0,
413 shadow = 0</label></transition><transition><source ref="id13"/><target
      ref="id15"/><label kind="assignment" x="-200"
      y="-24">updateShadow()</label><nail x="-88" y="48"/><nail x="-88"
      y="-88"/></transition><transition><source ref="id15"/><target
      ref="id13"/><label kind="guard" x="-64" y="-64">shadow ==
      getShadowStep()
414 &amp;&amp; sun &lt; getSunStep() &amp;&amp;
415 (step &lt; ChargeInterval ||
```

```
416 insolation_value[insolation_index] == 0)</label><label
        kind="assignment" x="-64" y="-8">shadow = 0, step =
        0</label></transition><transition><source ref="id15"/><target
        ref="id14"/><label kind="guard" x="-64" y="-272">sun ==
        getSunStep() &amp;amp;amp;
417 (step &lt; ChargeInterval ||
418 insolation_value[insolation_index] == 0)</label><label
        kind="assignment" x="-64" y="-232">sun = 0, step =
        0</label></transition></template><system>// Place template
        instantiations here.
419
420 basestation = BaseStation(0);
421 nodes(const int[1,Nodes-1] id) = Node(id);
422 environment = Environment();
423
424 // List one or more processes to be composed into a system.
425 system basestation, nodes, environment;
426
427 </system></nta>
```

Queries for the model of four nodes in a circle

```
1  //This file was generated from (Academic) UPPAAL 4.0.8 (rev. 4276),
       March 2009
2
3  /*
4  When node 1 run the Application, then at a later point base station
       will run Collect with data_source = 1
5  */
6  nodes(1).Application --> (basestation.Collect and data_source == 1)
7
8  /*
9  When node 2 run the Application, then at a later point either node 1 or
       node 2 will run RouteReceive with data_source = 2
10 */
11 nodes(2).Application --> ((nodes(1).RouteReceive or
       nodes(3).RouteReceive) and data_source == 2)
12
13 /*
14 When node 3 run the Application, then at a later point either node 2 or
       node 4 will run RouteReceive with data_source = 3
15 */
16 nodes(3).Application --> ((nodes(2).RouteReceive or
       nodes(4).RouteReceive) and data_source == 3)
17
18 /*
19 When node 4 run the Application, then at a later point base station
       will run Collect with data_source = 4
20 */
21 nodes(4).Application --> (basestation.Collect and data_source == 4)
22
23 /*
24 Verify that node 1 will always send data messages to base station
25 */
26 A[] (nodes(1).route == 0 || nodes(1).Init)
27
28 /*
```

```
29 Verify that node 2 will always send data messages to either node 1 or
       node 3
30 */
31 A[] (nodes(2).route == 1 || nodes(2).route == 3 || nodes(2).Init)
32
33 /*
34 Verify that node 3 will always send data messages to either node 2 or
       node 4
35 */
36 A[] (nodes(3).route == 2 || nodes(3).route == 4 || nodes(3).Init)
37
38 /*
39 Verify that node 4 will always send data messages to base station
40 */
41 A[] (nodes(4).route == 0 || nodes(1).Init)
42
43 /*
44
45 */
46 //NO_QUERY
47
48 /*
49 It is always true that all nodes never runs out of power
50 */
51 A[] forall (n : int[1,Nodes-1]) not nodes(n).OutOfPower
52
53 /*
54 Verify that the system does not deadlock, which implies that no nodes
       in the state of OutOfPower will receive any data messages.
55 */
56 A[] not deadlock
57
58 /*
59 Check if it is possible that node 1 runs out of power
60 */
61 E<>nodes(1).OutOfPower
62
63 /*
64 Check if it is possible that node 2 runs out of power
65 */
66 E<>nodes(2).OutOfPower
67
68 /*
69 Check if it is possible that node 3 runs out of power
70 */
71 E<>nodes(3).OutOfPower
72
73 /*
74 Check if it is possible that node 4 runs out of power
75 */
76 E<>nodes(4).OutOfPower
77
78 /*
79
80 */
81 //NO_QUERY
82
83 /*
```

```
84  Is it possible that node 2 will send data messages to any other node
        than node 1?
85  */
86  E<>(nodes(2).route != 1 and nodes(2).RouteSend)
87
88  /*
89  Is it possible that node 3 will send data messages to any other node
        than node 4?
90  */
91  E<>(nodes(3).route != 4 and nodes(3).RouteSend)
92
93  /*
94
95  */
96  E<> exists (n : int[1,Nodes-1]) nodes(n).route !=
        shortestPathNeighbour(n) and nodes(n).RouteSend
97
98  /*
99
100 */
101 //NO_QUERY
102
103 /*
104 Verify that node 1 will always optain an optimal route when either task
        UpdateEnergy or UpdateNeighbour has been activated.
105 (nodes(1).UpdateEnergy or nodes(1).UpdateNeighbour) -->
        nodes(1).hasValidRoute()
106 */
107 //NO_QUERY
108
109 /*
110 Verify that node 2 will always optain an optimal route when either task
        UpdateEnergy or UpdateNeighbour has been activated.
111 (nodes(2).UpdateEnergy or nodes(2).UpdateNeighbour) -->
        nodes(2).hasValidRoute()
112 */
113 //NO_QUERY
114
115 /*
116 Verify that node 3 will always optain an optimal route when either task
        UpdateEnergy or UpdateNeighbour has been activated.
117 (nodes(3).UpdateEnergy or nodes(3).UpdateNeighbour) -->
        nodes(3).hasValidRoute()
118 */
119 //NO_QUERY
120
121 /*
122 Verify that node 4 will always optain an optimal route when either task
        UpdateEnergy or UpdateNeighbour has been activated.
123 (nodes(4).UpdateEnergy or nodes(4).UpdateNeighbour) -->
        nodes(4).hasValidRoute()
124 */
125 //NO_QUERY
126
127 /*
128
129 */
130 //NO_QUERY
```

```
131
132 /*
133
134 */
135 A[] globalOptimalRoute(0) == 0
136
137 /*
138
139 */
140 A[] globalOptimalRoute(1) == 0
141
142 /*
143
144 */
145 E<> globalOptimalRoute(2) == 1
146
147 /*
148
149 */
150 E<> globalOptimalRoute(2) == 3
151
152 /*
153
154 */
155 E<> globalOptimalRoute(3) == 2
156
157 /*
158
159 */
160 E<> globalOptimalRoute(3) == 4
161
162 /*
163
164 */
165 A[] globalOptimalRoute(4) == 0
166
167 /*
168
169 */
170 //NO_QUERY
171
172 /*
173
174 */
175 E<> globalOptimalRoute(2) == 3 and (nodes(3).UpdateEnergy)
176
177 /*
178
179 */
180 //NO_QUERY
181
182 /*
183 Any node will always have a route in the set of neighbours or be in the
      Init state.
184 */
185 A[] forall (n : nodes_t) exists (s : neighbours_t) ((neighbours[n][s]
      == nodes(n).route and s < numNeighbours[n]) or nodes(n).Init)
186
```

```
/*

*/
//NO_QUERY

/*

*/
E<> exists(n : nodes_t) nodes(n).UpdateEnergy and
    nodes(n).isUpdateNeeded()

/*

*/
E<> exists(n : nodes_t) nodes(n).UpdateEnergy and not
    nodes(n).isUpdateNeeded()

/*

*/
E<> exists(n : nodes_t) nodes(n).UpdateNeighbour and
    nodes(n).isUpdateNeeded()

/*

*/
E<> exists(n : nodes_t) nodes(n).UpdateNeighbour and not
    nodes(n).isUpdateNeeded()

/*
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
*/
//NO_QUERY
```

Bibliography

[1] Forsyde graphical editor. http://code.google.com/p/fge/.

[2] The haskell programming language. http://www.haskell.org/.

[3] MathWorks products. http://www.mathworks.com/.

[4] Modelica and the Modelica Association. http://www.modelica.org/.

[5] The official OMG MARTE web site. http://www.omgmarte.org/.

[6] Open SystemC initiative (OSCI). http://www.systemc.org.

[7] GME: Generic Modeling Environment, September 2009. http://www.isis.vanderbilt.edu/Projects/gme.

[8] UML 2.0 OCL Specification, September 2009. http://www.omg.org/docs/ptc/03-10-14.pdf.

[9] D. Potter A. Kansal and MB Srivastava. Performance Aware Tasking for Environmentally Powered Sensor Networks. In *ACM Joint Intl. Conf. on Measurement and Modeling of Computer Systems*, 2004.

[10] Kemal Akkaya and Mohamed F. Younis. A survey on routing protocols for wireless sensor networks. *Ad Hoc Networks*, 3(3):325 – 349, 2005.

[11] J.N. Al-Karaki and A.E. Kamal. Routing techniques in wireless sensor networks: a survey. *IEEE Wireless Communications Magazine*, 11(6):6–28, 2004.

[12] ANSI S3.46-1997. Methods of measurement of real-ear performance characteristics of hearing aids, 1997.

[13] Gerd Behrmann, Alexandre David, and KimG. Larsen. A tutorial on uppaal. In Marco Bernardo and Flavio Corradini, editors, *Formal Methods for the Design of Real-Time Systems*, volume 3185 of *Lecture Notes in Computer Science*, pages 200–236. Springer Berlin Heidelberg, 2004.

[14] Shuvra S. Bhattacharyya, Joseph T. Buck, Soonhoi Ha, and Edward A. Lee. Generating compact code from dataflow specifications of multirate-signal processing algorithms. *IEEE Transactions on Circuits and Systems I: Fundamental Theory and Applications*, 42(3):138–150, March 1995.

[15] Shuvra S. Bhattacharyya, Praveen. K. Murthy, and Edward A. Lee. Synthesis of embedded software from synchronous dataflow specifications. *Journal of VLSI Signal Processing Systems*, 21(2):151–156, June 1999.

[16] Lawrence A. Bush, Christopher D. Carothers, and Boleslaw K. Szymanski. Algorithm for Optimizing Energy Use and Path Resilience in Sensor Networks. In *Wireless Sensor Networks, 2005. Proc. of the Second European Workshop on*, pages 391 – 396, 2005.

[17] Lukai Cai and Daniel Gajski. Transaction level modeling: an overview. In *Proceedings of the 1st IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 19–24, Newport Beach, CA, USA, 2003. ACM.

[18] Peter Corke, Philip Valencia, Pavan Sikka, Tim Wark, and Les Overs. Long-duration solar-powered wireless sensor networks. In *Proc. of the 4th workshop on Embedded networked sensors*, pages 33–37. ACM, 2007.

[19] Adam Donlin. Transaction level modeling: flows and use models. In *Proceedings of the 2nd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 75–80, Stockholm, Sweden, 2004. ACM.

[20] Johan Eker, Jörn W. Janneck, Edward A. Lee, Jie Liu, Xiaojun Liu, Jozsef Ludvig, Stephen Neuendorffer, Sonia Sachs, and Yuhong Xiong. Taming heterogeneity - the Ptolemy approach. In *Proceedings of the IEEE*, pages 127–144, 2003.

[21] EN 61669. Electroacoustics - equipment for the measurement of real-ear acoustical characteristics of hearing aids, 2001.

[22] Jabed Faruque and Ahmed Helmy. Gradient-based routing in sensor networks. *SIGMOBILE Mob. Comput. Commun. Rev.*, 7(4):50–52, 2003.

[23] ForSyDe: Formal System Design. https://forsyde.ict.kth.se/.

[24] H. Hassanein and Jing Luo. Reliable Energy Aware Routing in Wireless Sensor Networks. In *Dependability and Security in Sensor Networks and Systems, 2006*, pages 54–64. IEEE, 2006.

[25] Fernando Herrera and Eugenio Villar. A framework for heterogeneous specification and design of electronic embedded systems in SystemC. *ACM Trans. Des. Autom. Electron. Syst.*, 12(3):1–31, 2007.

[26] Andreas Herrholz, Frank Oppenheimer, Philipp Andreas Hartmann, Andreas Schallenberg, Wolfgang Nebel, C. Grimm, M. Damm, F. Herrera, E. Villar, I. Sander, A. Jantsch, A.-M. Fouilliart, and Mart. The ANDRES project : Analysis and design of run-time reconfigurable, heterogeneous systems. In *Proceedings of 2007 International Conference on Field Programmable Logic and Applications*. IEEE, aug 2007.

[27] Chalermek Intanagonwiwat, Ramesh Govindan, Deborah Estrin, John Heidemann, and Fabio Silva. Directed Diffusion for Wireless Sensor Networking. *IEEE/ACM Transactions on Networking*, 11(1):2–16, Februar 2002.

[28] Junayed Islam, Muhidul Islam, and Nazrul Islam. A-sLEACH: An Advanced Solar Aware Leach Protocol for Energy Efficient Routing in Wireless Sensor Networks. *International Conference on Networking*, 0:4, 2007.

[29] D. Jagtap, N. Abu-Ghazaleh, and D. Ponomarev. Optimization of parallel discrete event simulator for multi-core systems. In *Parallel Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*, pages 520 –531, may 2012.

[30] Mikkel Koefoed Jakobsen. Energy harvesting aware routing and scheduling in wireless sensor networks. Master's thesis, Technical University of Denmark, Department of Informatics and Mathematical Modeling, September 2008.

[31] Mikkel Koefoed Jakobsen, Jan Madsen, and Michael R. Hansen. DEHAR: A distributed energy harvesting aware routing algorithm for ad-hoc multihop wireless sensor networks. In *World of Wireless Mobile and Multimedia Networks (WoWMoM), 2010 IEEE International Symposium on a*, pages 1 –9, june 2010.

[32] Axel Jantsch. *Modeling Embedded Systems and SoCs*. Morgan Kaufmann, 2004.

[33] Kenneth E. A. Jensen. Schedulability analysis of embedded applications modelled using MARTE. Master's thesis, Technical University of Denmark, 2009.

[34] Xiaofan Jiang, Joseph Polastre, and David Culler. Perpetual environmentally powered sensor networks. In *Proc. of the 4th intl. symposium on Information processing in sensor networks*. IEEE Press, 2005.

[35] Aman Kansal, Jason Hsu, Sadaf Zahedi, and Mani B. Srivastava. Power management in energy harvesting sensor networks. *ACM Trans. Embed. Comput. Syst.*, 6(4):32, 2007.

[36] H. Kleen, T. Schubert, and C. Grabbe. A tutorial for OSSS. Technical report, 2006. http://icodes.offis.de.

[37] Kim G. Larsen, Paul Pettersson, and Wang Yi. Uppaal in a nutshell. *International Journal on Software Tools for Technology Transfer*, 1:134–152, December 1997.

[38] Emanuele Lattanzi, Edoardo Regini, Andrea Acquaviva, and Alessandro Bogliolo. Energetic sustainability of routing algorithms for energy-harvesting wireless sensor networks. *Comput. Commun.*, 30(14-15):2976–2986, 2007.

[39] E.A. Lee and D.G. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235 – 1245, sept. 1987.

[40] E.A. Lee and D.G. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235 – 1245, sept. 1987.

[41] Edward A. Lee and Alberto Sangiovanni-Vincentelli. A framework for comparing models of computation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 17(12):1217–1229, December 1998.

[42] Edward A. Lee and Haiyang Zheng. Leveraging synchronous language principles for heterogeneous modeling and design of embedded systems. In *EMSOFT '07: Proceedings of the 7th ACM & IEEE international conference on Embedded software*, pages 114–123, New York, NY, USA, 2007. ACM.

[43] Edward Ashford Lee and David G. Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Transactions on Computers*, C-36(1):24–35, January 1987.

[44] Longbi Lin, Ness B. Shroff, and R. Srikant. Asymptotically optimal energy-aware routing for multihop wireless networks with renewable energy sources. *IEEE/ACM Transactions on Networking*, 15(5):1021–1034, 2007.

[45] Chi Ma and Yuanyuan Yang. Battery-aware routing for streaming data transmissions in wireless sensor networks. *Mob. Netw. Appl.*, 11(5):757–767, 2006.

[46] J. Madsen, S. Mahadevan, and K. Virk. Network-Centric System-Level Model for Multiprocessor SoC Simulation. In *Interconnect-Centric Design for Advanced SoC and NoC*, pages 341–365. Kluwer Academic, 2004.

[47] J. Madsen, K. Virk, and M. J. Gonzalez. A SystemC-Based Abstract Real-Time Operating System Model for Multiprocessor System-on-Chip. In *Multiprocessor System-on-Chip*, pages 283–312. Morgan Kaufmann, 2004.

[48] Shankar Mahadevan, Michael Storgaard, Jan Madsen, and Kashif Virk. ARTS: A System-Level Framework for Modeling MPSoC Components and Analysis of their Causality. In *Proceedings of the 13th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS'05)*, pages 480–483, Washington, DC, USA, sep 2005. IEEE Computer Society.

[49] Shankar Mahadevan, Kashif Virk, and Jan Madsen. ARTS: A SystemC-Based Framework for Multiprocessor Systems-on-Chip Modelling. *Design Automation for Embedded Systems*, 11(4):285–311, 2007.

[50] S. Mahfoudh and P. Minet. Survey of Energy Efficient Strategies in Wireless Ad Hoc and Sensor Networks. In *Seventh Intl. Conf. on Networking*, pages 1–7, 2008.

[51] Raminder P. Mann, Kamesh R. Namuduri, and Ravi Pendse. Energy-Aware Routing Protocol for Ad Hoc Wireless Sensor Networks. *EURASIP Journal on Wireless Communications and Networking*, 2005(5):635–644, 2005.

[52] Deepak Mathaikutty, Hiren Patel, Sandeep Shukla, and Axel Jantsch. EWD: A metamodeling driven customizable multi-MoC system modeling framework. *ACM Trans. Des. Autom. Electron. Syst.*, 12(3):1–43, 2007.

[53] S. Mørk, J.C. Godskesen, Michael R. Hansen, and R. Sharp. A timed semantics for sdl. In R. Gotzhein and J. Bredereke, editors, *Formal Description Techniques IX: Theory, application and tools*, pages 295–309. Chapman & Hall, 1996.

[54] M. Montoreano. Transaction level modeling using OSCI TLM 2.0. http://www.systemc.org, 2007.

[55] C. Moser, L. Thiele, L. Benini, and D. Brunelli. Real-Time Scheduling with Regenerative Energy. In *Proc. of the 18th Euromicro Conf. on Real-Time Systems*, pages 261–270. IEEE Computer Society, 2006.

[56] S.H.A. Niaki, M.K. Jakobsen, T. Sulonen, and I. Sander. Formal heterogeneous system modeling with systemc. In *Specification and Design Languages (FDL), 2012 Forum on*, pages 160 –167, sept. 2012.

[57] R. Nikhil. Bluespec System Verilog: efficient, correct RTL from high level specifications. In *Formal Methods and Models for Co-Design, 2004. MEMOCODE '04. Proceedings. Second ACM and IEEE International Conference on*, pages 69–70, 2004.

[58] H. D. Patel and S. K. Shukla. Towards a heterogeneous simulation kernel for system level models: A SystemC kernel for synchronous data flow models. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 24(8):1261–1271, 2005.

[59] Henrik Pilegaard, Michael R. Hansen, and Robin Sharp. An approach to analyzing availability properties of security protocols. *Nordic Journal of Computing*, 10:337–373, 2003.

[60] D.I. Rich. The evolution of SystemVerilog. *Design & Test of Computers, IEEE*, 20(4):82–84, 2003.

[61] A. Rose, M. Graphics, S. Swan, J. Pierce, and J. M. Fernandez. Transaction level modeling in SystemC. http://www.systemc.org, 2005.

[62] Ingo Sander. *System modeling and design refinement in ForSyDe*. PhD thesis, LECS/IMIT/Royal Institute of Technology (KTH), Stockholm, 2003.

[63] Ingo Sander and Axel Jantsch. System modeling and transformational design refinement in ForSyDe. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 23(1):17–32, January 2004.

[64] Alberto Sangiovanni-Vincentelli. Quo vadis, SLD? Reasoning about the trends and challenges of system level design. *Proceedings of the IEEE*, 95(3):467–506, March 2007.

[65] A. Schallenberg, W. Nebel, and F. Oppenheimer. OSSS+R: Modelling and simulating self-reconfigurable systems. In *Proceedings of International Conference on Field Programmable Logic and Applications*, pages 28–30, 2006.

[66] Muruganathan S.D., Ma D.C.F., Bhasin R.I., and Fapojuwo A.O. A centralized energy-efficient routing protocol for wireless sensor networks. *IEEE Communications Magazine*, 43(3):S8–13, 2005.

[67] Rahul C. Shah and Jan M. Rabaey. Energy aware routing for low energy ad hoc sensor networks. In *Wireless Communications and Networking Conf., 2002*, volume 1, pages 350 – 355. IEEE, 2002.

[68] Farhan Simjee and Pai H. Chou. Everlast: long-life, supercapacitor-operated wireless sensor node. In *Proc. of the 2006 intl. symposium on Low power electronics and design*, pages 197–202. ACM Press, 2006.

[69] Uppsala University and Aalborg University. UPPAAL. http://www.uppaal.com/.

[70] A. Vachoux, C. Grimm, and K. Einwich. Towards analog and mixed-signal SoC design with SystemC-AMS. In *IEEE International Workshop on Electronic Design, Test and Applications (DELTA' 04)*, Perth, Australia, 2004.

[71] Dimitrios J. Vergados, Nikolaos A. Pantazis, and Dimitrios D. Vergados. Energy-efficient route selection strategies for wireless sensor networks. *Mob. Netw. Appl.*, 13(3-4):285–296, 2008.

[72] T. Voigt, A. Dunkels, J. Alonso, H. Ritter, and J. Schiller. Solar-aware clustering in wireless sensor networks. *IEEE Symp. on Computers and Communications*, 1:238–243, 2004.

[73] Thiemo Voigt, Hartmut Ritter, and Jochen Schiller. Solar-aware Routing in Wireless Sensor Networks. In *Intl. Workshop on Personal Wireless Communications*, pages 847–852. Springer, 2003.

[74] Jinghao Xu, Bojan Peric, and Branimir Vojcic. Performance of energy-aware and link-adaptive routing metrics for ultra wideband sensor networks. *Mob. Netw. Appl.*, 11(4):509–519, 2006.

[75] Kai Zeng, Kui Ren, Wenjing Lou, and Patrick J. Moran. Energy-aware geographic routing in lossy wireless sensor networks with environmental energy supply. In *Proc. of the 3rd intl. conf. on Quality of service in heterogeneous wired/wireless networks*, page 8. ACM Press, 2006.

[76] Baoxian Zhang and Hussein T. Mouftah. Adaptive Energy-Aware Routing Protocols for Wireless Ad Hoc Networks. In *Proc of the First Intl. Conf. on Quality of Service in Heterogeneous Wired/Wireless Networks*, pages 252–259. IEEE Computer Society, 2004.