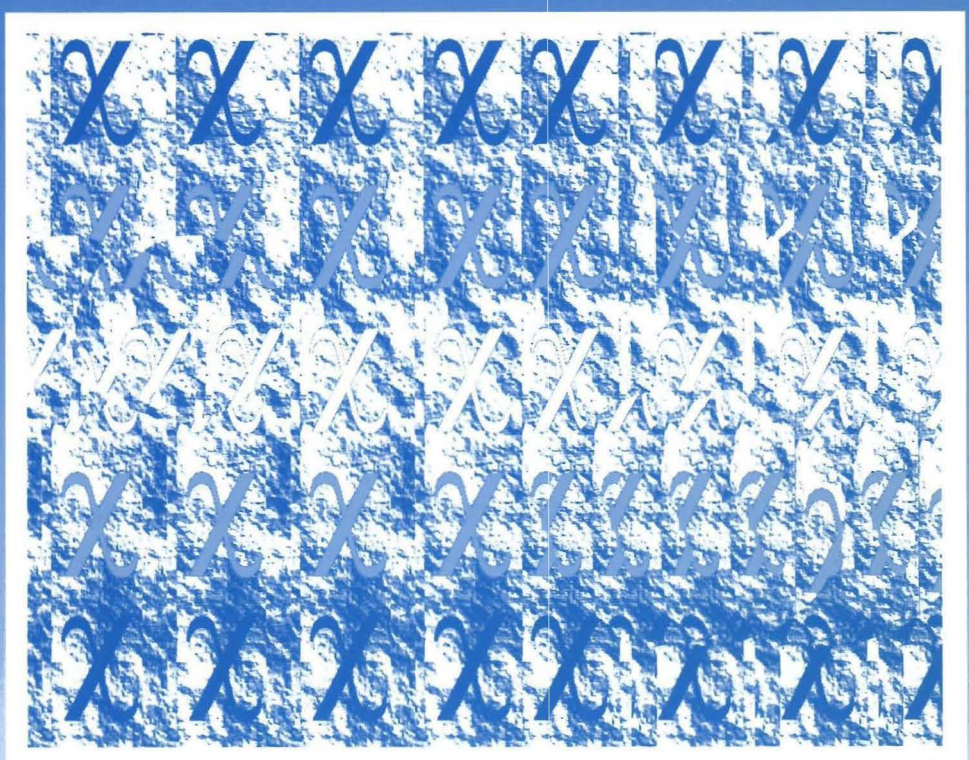
A systems engineering specification formalism

A Systems Engineering Specification Formalism



N.W.A. Arends

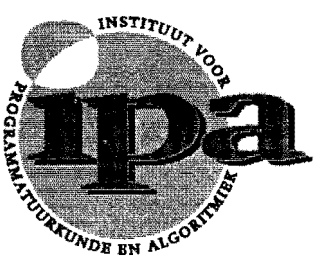
A Systems Engineering Specification Formalism

Cover design: N.W.A. Arends.
Print: Wibro, Helmond.

CIP-DATA KONINKLIJKE BIBLIOTHEEK, DEN HAAG

Arends, Norbertus Wilhelmus Anthonius

A Systems Engineering Specification Formalism /
Norbertus Wilhelmus Anthonius Arends.
- Eindhoven: Eindhoven University of Technology
Thesis Technische Universiteit Eindhoven.
ISBN 90-386-0028-3
Subject headings: discrete continuous systems / systems
engineering / industrial systems.



The work in thesis has been carried out under the auspices of the research school IPA (Institute for Programming and Algorithmics).

A Systems Engineering Specification Formalism

PROEFSCHRIFT

TER VERKRIJGING VAN DE GRAAD VAN DOCTOR AAN DE TECHNISCHE UNIVERSITEIT EINDHOVEN, OP GEZAG VAN DE RECTOR MAGNIFICUS, PROF.DR. J.H. VAN LINT, VOOR EEN COMMISSIE AANGEWEZEN DOOR HET COLLEGE VAN DEKANEN IN HET OPENBAAR TE VERDEDIGEN OP MAANDAG 17 JUNI 1996 OM 16.00 UUR

DOOR

NORBERTUS WILHELMUS ANTHONIUS ARENDS

GEBOREN TE EINDHOVEN

Dit proefschrift is goedgekeurd door de promotoren:

prof.dr.ir. J.E. Rooda

en

prof.dr. M. Rem

to my late uncle
Will Smulders

Preface

Many times, during the research period of this project, I have noticed some confusion about my backgrounds as a mechanical engineer. The development of a specification formalism is not something you would normally expect to be carried out in the field of mechanical engineering. The required mathematics to define a formalism lie beyond the scope of a mechanical engineer. A mathematical engineer, however, can not be expected to have the required knowledge of industrial systems to design a specification formalism. The obvious solution is to join forces.

Our cooperation with the Department of Mathematics and Computing Science at Eindhoven University of Technology was, by all means, pleasing and rewarding. Therefore, I would like to thank M. Rem for his support and refreshing views and for his comments on this thesis. Also, I would like to thank J.E. Rooda and my colleagues J.M. van de Mortel-Fronczak and D.A. van Beek for each being a stimulant and a great resource of ideas. They were, together with W.T.M. Alberts, G. Naumoski, G. Fabian, and P.L. Janson, members of the 'χ-club'. Our weekly meetings provided that extra.

Furthermore, I would like to thank all people that have in some way contributed to this work or have provided an enjoyable working environment. My special thanks go to M.W.J. Gunter-Lousberg, for her coffee and amiable conversations, H.W.A.M. van Rooij, for sharing his PC-related knowledge with me, G.F.W.J. van de Kamp and E.M.M. Voorbraak for their contributions to Chapter 7, and H.A. Preisig and A.A. van Steenhoven for their comments on this thesis and taking part in the committee.

Finally, I want to thank Monique, for sharing her life with me.

Summary

The design process of an industrial system is a process of decision-making, where each decision has its impact on the resulting system. To substantiate design decisions, a formal specification is inevitable. In this research, we introduce the formalism χ as a means to specify the dynamic behaviour of industrial systems. The dynamic behaviour is a key aspect in the design of a system.

The basic concepts of the formalism are derived from the process-interaction approach in which a system is viewed as a collection of interacting processes, operating concurrently. Each process models the behaviour of a system component. Processes can be grouped into a system, where each system can act as a process. It can be grouped with other processes and systems, thereby forming a new system. This hierarchical decomposition of a system into subsystems allows the formalism to be used for the specification of systems of arbitrary complexity.

The behaviour, described by a process, can be discrete, continuous, or a combination of discrete and continuous. In χ, both discrete and continuous behaviours are considered first class citizens. Neither of the two is subordinate to the other. Moreover, it is possible to make a specification with only a continuous or a discrete behaviour. The description of a discrete behaviour has the form of a sequential program where changes in the state of a process are accomplished by performing actions. The continuous behaviour is described by a set of differential and algebraic equations.

The relations between processes are defined by channels. We distinguish between discrete and continuous channels. Discrete channels are used for synchronisation or communication. Both these applications are based on a synchronous interaction mechanism. A continuous channel defines a relation between local variables of different processes. All channels are one-to-one connections.

The language for the description of a discrete behaviour is derived from the guarded command language, extended with an explicit notion of time. The language provides constructs to specify nondeterministic choices between alternative actions, guarded by Boolean expressions. Furthermore, a guarded command enables an orthogonal combination of different events, such as communication, delay, and state-events.

The interaction between a continuous and a discrete behaviour is described in a process. Only two language constructs are needed to define these interactions: state-events and guarded equations. A state-event is a discrete event that is triggered by a Boolean expression becoming true. This expression depends on the continuous state of a process. It defines the influence of the continuous behaviour on the discrete behaviour. The influence of the discrete behaviour on the continuous behaviour is defined by a guarded equation. A guarded equation defines a choice between alternative sets of equations, guarded by Boolean expressions that depend on the discrete state of a process.

The state of a process is defined by local variables in the process. Moreover, the formalism does not allow the use of global variables. The formalism uses strong typing for all data objects. Therefore, all variables and channels are declared with a type, denoting the domain of possible values.

This dissertation deals with the definition of the syntax and semantics of the formalism χ. The application of the formalism is illustrated by examples of industrial systems, including their control systems. Furthermore, it gives a survey of the design decisions that resulted in the presented formalism. This research is meant to initiate the development of a calculus, dedicated to the analysis of the behaviour of industrial systems.

Samenvatting

Het ontwerp-proces van industriële systemen is een proces van het nemen van ontwerp-beslissingen, waarbij elke beslissing zijn invloed heeft op het resulterende ontwerp. Voor het beargumenteren van ontwerp-beslissingen is een formele specificatie onontbeerlijk. Dit proefschrift presenteert het formalisme χ als een manier voor het beschrijven van het dynamische gedrag van industriële systemen. Het dynamisch gedrag vormt een belangrijk aspect van een systeem.

De basisconcepten van het formalisme zijn afgeleid van de proces-interactie benadering waarin een systeem wordt beschouwd als een verzameling parallel werkende interacterende processen. Elk proces modelleert een component van het systeem. Processen kunnen worden gegroepeerd in een systeem, waarbij elk systeem weer als een proces kan worden gezien. Het kan worden gegroepeerd met andere processen en systemen teneinde een nieuw systeem vormend. Dit hiërachisch decomponeren van een systeem in subsystemen maakt het mogelijk het formalisme te gebruiken voor de specificatie van een systeem van een willekeurige complexiteit.

Het gedrag, beschreven door een proces, is discreet, continu of een combinatie van discreet en continu. In χ worden beide soorten gedrag als even belangrijk beschouwd. Geen van beide is ondergeschikt aan de ander. Het is mogelijk om een specificatie op te stellen met uitsluitend processen met een discreet dan wel een continu gedrag. De beschrijving van het discrete gedrag heeft de vorm van een sequentieel programma waarbij veranderingen in de toestand van een proces worden verkregen door het uitvoeren van akties. Het continue gedrag wordt beschreven door een verzameling differentiaal en algebraïsche vergelijkingen.

De relaties tussen processen worden gedefinieerd door kanalen. Er wordt onderscheid gemaakt tussen discrete en continue kanalen. Discrete kanalen worden gebruikt voor synchronisatie en communicatie. Beide toepassingen zijn gebaseerd op een synchroon interactie-mechanisme. Een continu kanaal definieert een relatie tussen lokale variabelen van verschillende processen. Alle kanalen vormen een één-op-één verbinding.

De taal voor het beschrijven van het discrete gedrag is afgeleid van de 'guarded command language', uitgebreid met het begrip tijd. De taal biedt constructies voor het specificeren van een nondeterministische keuze tussen alternatieve akties elk voorzien van een Boolse expressie. Verder maakt een guarded command een orthogonale combinatie van verschillende events mogelijk, zoals communicatie, wachten en state-events.

De interacties tussen een continu en een discreet gedrag wordt beschreven in een proces. Slechts twee taalelementen zijn nodig voor het beschrijven van deze interacties: state-events en guarded equations. Een state-event is een discreet event dat wordt veroorzaakt door het waar worden van een Boolse expressie. Deze expressie hangt af van de continue toestand van een proces. Het definieert de invloed van het continue gedrag op het discrete gedrag. De invloed van het discrete gedrag op het continue gedrag wordt gedefinieerd door een guarded equation. Een guarded equation definieert een keuze tussen een aantal verzamelingen van vergelijkingen die elk zijn voorzien van een Boolse expressie die afhangt van de discrete toestand van een proces.

De toestand van een proces word vastgelegd door lokale variabelen. Het formalisme staat het gebruik van globale variabelen niet toe. Het formalisme gebruikt sterk getypeerde data-objecten. Alle variabelen en kanalen worden daarom gedeclareerd met een type welk het domein aanduidt.

Dit proefschrift handelt over de definitie van de syntaxis en semantiek van het formalisme χ. De toepassing van het formalisme wordt geïllustreerd met voorbeelden van industriële systemen inclusief hun besturingssystemen. Verder geeft het een overzicht van de ontwerp-beslissingen die hebben geresulteerd in het gepresenteerde formalisme. Dit onderzoek beoogd een aanzet te zijn voor de ontwikkeling van een calculus, toegewijd aan de analyse van het gedrag van industriële systemen.

Contents

Chapter 1

Introduction

There seems to be no limit to the demand for products to support the level of prosperity in modern society. The demand for more diversity and larger amounts ever increases. Also, the life-cycle of a product decreases as new technological developments provide new innovative products. As a result of these changes in the consumer's society, the production of these products becomes more complex every day. The production is no longer artisan's work, but complex manufacturing systems must be used to attain the consumer's satisfaction.

1.1 Designing systems

Realisation of such complex systems incorporates the effort of experts in the fields of systems engineering, chemistry, physics, mathematics, and many other disciplines. In the design phase of an industrial system, the experts are working as a team in the project. Designing an industrial system is therefore not only a technical problem but also an organisational one. To ensure that the cooperation of the different disciplines will produce the desired results, a proper specification of intermediate designs is essential. If this specification is a formal one, ambiguity and misinterpretation can be avoided. Formal specifications also enable designers to communicate more easily on their designs and can bring up deficiencies in the design before they are implemented.

The design process of an industrial system can be defined as decision-making concerning the object to be realised [Bra93]. Every decision has its repercussions on the final design of the system and must, therefore, be made well-considered. To help in substantiating a decision, an evaluation of the consequences of the different alternatives should be made. Many evaluation techniques exist to support this decision-making process. Most of these techniques make use of a model of

the object to be designed in which relevant aspects of the object are represented. A formal model enables a formal evaluation and thus leads to more substantiated decisions. The introduction of the computer has increase the possibilities to evaluate formal models. Now that computers become largely available at low costs, these techniques are increasingly gaining the interest of designers.

Summarising, we can state that a formal specification is a valuable document in the design or redesign phase of industrial systems. It can serve as a communication document for experts working on the design and it enables the evaluation of intermediate designs. The evaluation possibilities of a formal specification can help to substantiate the different decisions to be made. Analysis and simulation techniques can evaluate 'what-if' scenarios and can thereby increase the confidence in the resulting design. They can, to some extent, ensure that a design meets the specifications of the system and they can shorten the design process. Furthermore, a formal specification avoids misinterpretations during the realisation phase of the system. Moreover, if the representation of the specification is executable, it can be used in the implementation of the system.

1.2 A new formalism

In this dissertation, we propose a specification formalism for industrial systems. We concentrate on the formal specification of the system's behaviour. The specification of the behaviour is a model of the system that specifies how the system must behave. The behaviour of a system is one of the key aspects in the design of a system. The introduction of a formal specification technique increases the quality of the design and decreases the effort to create the design. Usually, modelling techniques for the specification of a system behaviour distinguish between continuous-time and discrete-event behaviour.

In a continuous-time approach, the system is described by a set of equations and the state of the model changes continuously when time passes. Therefore not only the time changes continuously, but also the model's state. The techniques vary in the kind of equations that are allowed in a model and in the possibilities to describe discontinuities in the otherwise continuous behaviour. Simulation packages, supporting these techniques are often based on the standard formulated by the CSSL committee [CSS67]. One of the drawbacks of this standard is the restriction to ordinary differential equations. During the past decades, other modelling techniques evolved, allowing differential and algebraic equations and even partial differential equations.

A discrete-event specification of the behaviour of a system describes the abrupt changes in the model's state that occur at fixed points in time. Here both the state space and the time are discretised. A number of different approaches can be recognised in the available techniques today [Nan81, Pol89, Tan94]. These

include event-scheduling, activity-scanning, Petri-nets [Pet62], and the process-interaction approach. The use of discrete-event techniques is still not as widespread as is the use of continuous-time modelling techniques. Since the introduction of commercially available simulation packages in the early seventies, the use of discrete-event techniques is slowly gaining acceptance by designers of industrial systems.

The choice for a continuous or a discrete modelling technique becomes more ambiguous when the systems to be modelled become more complex. For instance, chemical processes in the process industry are usually modelled using a continuous-time approach. If, however, the control system is to be included in the model, a discrete-event approach would be more suitable. For this reason, many recent attempts have been made to combine the two approaches in one modelling technique. The developments originated from both the continuous-time and the discrete-event worlds and often consist of adapting an existing technique from the other world of modelling techniques to fit into the technique already used. The results, however, are not always satisfactory because of the large differences in the two approaches. Most problems occur when parts of the model, described by two techniques, interact with each other. Interfacing two modelling techniques involves the addition of new constructs that often do not fit into the concepts of either of the two modelling techniques that were combined.

1.3 Research objectives

From the previous section, we can conclude that there is a need for a theoretical base or framework that enables smooth interactions between discrete and continuous models. A possible approach for combining continuous and discrete modelling techniques is to design a formalism and integrate the concepts for the interactions between the two kinds of behaviours directly from the beginning. This would avoid the problems mentioned earlier. It also allows the decision, whether to model a certain behaviour as a continuous or as a discrete behaviour, to be postponed to a later stage in the modelling process.

The objective of the research in this thesis is to develop a uniform specification formalism for the modelling of the behaviour of industrial systems. With uniform, we mean that the formalism must be capable of modelling both continuous and discrete behaviours. Furthermore, the formalism must be able to specify both the primary and the control processes of a system. The evaluation of a specification can be achieved by either applying analytical or numerical evaluation techniques. Although evaluation techniques are not subject of this research, the formalism must allow the application of such techniques to investigate the system behaviour. As such, the formalism must be independent of any evaluation technique.

1.4 Overview

This dissertation is organised as follows. In Chapter 2, first we briefly review systems theory and give definitions for a system and a model. The life-cycle of a system illustrates in which phases in this life-cycle of a system models are used. As an example of an industrial system, Chapter 3 describes a bottle-filling system. Both the primary and the control processes are defined. With this example, the requirements for the specification of industrial systems are investigated, resulting in the objectives for a new formalism. The new formalism is then presented in Chapter 4. The syntax and semantics of the formalism are defined and some examples of its use are given. With the new formalism, a model of the bottle-filling system is set up in Chapter 5. This chapter is intended to give a brief introduction to the use of the formalism. Chapter 6 deals with some critical notes on the design of the formalism. As in any design process, many compromises must be made. This chapter substantiates the most relevant design decisions. The use of the formalism is illustrated with a real-life example of an industrial system in Chapter 7. The dissertation is completed in Chapter 8 with some concluding remarks and suggestions for future research.

Chapter 2

Systems and Models

Industrial systems become more complex due to the ever increasing demand for more flexibility. As these systems exhibit an increasing complexity, so will the process to design these systems become more complex. The design process is a process of decision-making, where each decision has its impact on the overall design of a system. Thus, each decision has to be made carefully and should be evaluated to investigate its consequences. Models play an important role in the design process of industrial systems. With the increasing complexity of systems, they become an essential means to substantiate design decisions.

To attain the research objectives, mentioned in the previous chapter, we need a suitable set of definitions of systems and models. In this chapter, we will briefly review systems theory and define a vocabulary to be used in the chapters to follow. We start by defining a system and in particular industrial systems. Next, models are defined to investigate the behaviour of systems and some desirable characteristics of model representations will be discussed. The use of models is discussed in the context of the life-cycle of industrial systems. Finally, a brief survey of the modelling process for industrial systems is presented.

2.1 Systems

In this section, we will review systems theory and present a set of definitions to be used in the remainder of this thesis. Furthermore, we will concentrate on the definition of industrial systems and will emphasise some characteristics that are relevant for the modelling of these systems.

Systems theory

Many definitions of a system have been given in all kinds of scientific text books and articles. Each scientific discipline uses its own (set of) definitions that suits best for their purpose. Yet, the word system is not ambiguous. It is merely the wide spread applicability of systems that caused these many definitions. Moreover, because of the wide spread use of systems, it is hard to give a general definition. Gaines has made an attempt in this direction with the following definition [Gai79]:

> "A system is what is distinguished as a system."

Although this definition may seem a trivial one, it indicates an essential characteristic of a system: it has to be distinguished from the rest of the universe. From this definition, it is clear that anything can be defined to be a system. The creation of a system makes only sense if it serves certain objectives. This is reflected by the definition of a system given in the Webster's dictionary [Web86]:

> "A system is a complex unit of many, often diverse, parts subject to a common plan or serving a common purpose."[1]

By separating a part of the universe, the system boundaries are determined. The elements contained in the system are considered to be part of the system because of the existence of relationships between these elements which are relevant for the system's objectives. Here, another characteristic of systems has been encountered: a system contains a collection of interrelated elements. Bertalanffy defines a system as a set of elements standing in interrelation [Ber68]. A similar definition is given by Van Aken [Ake78]:

> "A system is a set of elements with a set of relations between the elements, where the relations have the property that all elements are directly or indirectly related."

Together with the system's boundaries, also the relations between elements inside the system and other elements outside the system are defined. The latter elements are referred to as the system's environment or surroundings. If there are no relations with the environment, and thus the environment contains no elements, the system is called a closed system. Otherwise the system is an open system.

Because of the relations between the system's elements, the elements can be considered to have some kind of ordering. This ordering is reflected by the definition given in [Web90]:

[1] Note that the terms 'complex' and 'many' are subjective and have no relevance to the definition of a system. A system can also be a simple unit of a few parts.

> "A system is an orderly, interconnected, complex arrangement of parts."[2]

In [Coh86], a system is said to have the following characteristics:

- *Systems exhibit behaviour.* Requirements for a system are largely formulated in terms of the observable behaviour of a system. A system's behaviour is observable by the interactions it has with its environment.
- *Systems have internal structure.* The system's behaviour is the result of the behaviour of its parts, which may themselves be systems, and of the interrelationships among these parts.

A subset of the system's elements can again be distinguished as a system. In this case we speak of a subsystem. This can also be applied to the elements of each subsystem. This hierarchical ordering of elements is based on the relations between the elements in the subsystems and thus is considered to be part of the system's structure.

Summarising, we have found the following characteristics that many definitions of a system have in common:

- A system is defined to serve certain objectives.
- A system is a separated part of the universe.
- A system is a collection of hierarchically ordered interrelated elements.
- A system can have relations with its environment.
- A system exhibits a behaviour.

Many related concepts to the definition of a system are mentioned in literature [Ber68, Ake78, Coh86]. Often, the definitions of these concepts are similar and interchangeable. A subset of these system theoretical concepts will be defined here.

In the definitions of a system, the term *element* is mentioned frequently. An element can be defined as the smallest part of a system that is considered to be indivisible. Also the concept of *relations* or *relationships* is used in system definitions. A relation denotes interdependence between elements. A subset of the collection of elements in a system and all the relations between these elements is called a *subsystem.* The collection of all relations is called the *structure* of the system.

[2] Again, the term 'complex' is subjective and has no relevance to the definition of a system. (See also: [1].)

Without defining a system explicitly, we have given the most frequently mentioned characteristics used in definitions of a system. These characteristics will provide an adequate understanding of the concept system. Furthermore, we have defined some related system theoretical concepts. Other concepts from the field of systems theory can be defined with the concepts presented here. This brief review of systems theory is considered to be sufficient for the remainder of this dissertation.

Industrial systems

We will now concentrate on the definition of an industrial system. Industrial systems can be distinguished by the incorporation of three subsystems. The first subsystem, the primary system, involves the flow of material. In a manufacturing environment one can think of products and raw materials. Although it may sound rather crude, also people can be considered as material flow, for instance in the case of a hospital. In general, the material flow is all that is subject to a transformation, carried out by the industrial system. Moreover, this transformation of material is the primary objective of the industrial system.

The secondary system is associated with the flow of information. This information can comprise orders for a factory, signals from a sensor, or sales reports for the management team. The information flow controls the material flow. This subsystem is therefore often referred to as the control system. Although information can be transformed by the system, it is not part of the material flow as it is not the primary objective of the system to transform this information.

The tertiary system or economical system incorporates the flow of values, compensating the flow of material. The value flow is responsible for preserving the system. In a sense, it can be seen as the 'energy' for the system. A value, in this case, is seen in a wider context. It can be a monetary value or a compensation in goods. Again, although the system may transform the value flow, this flow is not part of the material flow. In [Roo83], the value flow has an accompanying information flow, called the invoice flow. This information flow controls the flow of values. In our discussion, however, this invoice flow is considered to be included in the information flow of the secondary system.

The three subsystems mentioned above comprise an industrial system. This matches the definition of a production system given by Brandts in [Bra93]. According to Brandts, an industrial system consists of a collection of products and a production system. We will, however, leave an explicit definition of products aside in our definition of an industrial system and consider the products included in the material flow.

From the three subsystems, it is easily recognised that the relations with the system's environment involve flows of material, information and values. An industrial system receives material, information, and values from its environment,

transforms these flows and supplies them (in a transformed state) back to the environment. Note that an industrial system is considered to be an open system. Its primary objective is to transform material that is exchanged with its environment. It thus has relations with its environment and is therefore an open system.

Many examples of industrial systems can be found around us. Product manufacturing systems and distribution centres are commonly known examples. Also, a refinery and a waste water treatment system are industrial systems. Even a single machine, a person, or a hospital can equally well be considered as an industrial system.

2.2 Models

To investigate the behaviour of an industrial system, we must study the input and output flows of the system and the transformation of these flows within the system. It is thereby desirable to suppress irrelevant aspects and to make an abstract system, representing only the relevant aspects of the system that are responsible for the transformation of input and output flows. Such an abstract system is called a model.

In this section, we will first give a brief introduction to modelling theory. Then, we will address some issues concerning models of industrial systems.

Modelling theory

Models are used to:

- think about the system and try to understand it,
- communicate with others about the system,
- perform experiments on.

The first application of models is common to all scientific and engineering research. Together with the experimentation on models, it is a way of gathering knowledge about a system. It enables us to structure the knowledge and deduct the relations between causes and effects. The definition of science, given in [Web90], illustrates the contribution that models have to science:

> "Science is the knowledge acquired by careful observation, by deduction of the laws which govern changes and conditions, and by testing these deductions by experiment."

Definitions of models are as diverse as definitions of systems. Minsky has given a definition in which the scientific characteristics of models are expressed [Min65]:

> "A model M for a system S and an experiment E is anything to which E can be applied in order to answer questions about S."

From this definition, we can conclude that a model is a system. Furthermore, a model is always related to a system and an experiment. The latter determines what is represented in the model: it defines the model objectives. The model objectives determine which aspects of the system are relevant, and should thus be included in the model, and which aspects are to be left out. Without first stating the objectives for a model, it is impossible to build a model.

Note that the validity of a model is not only determined by the system but also by the experiment for which the model is build. A model is valid if it provides the answers to the posed questions about a system. The answers can be attained by performing an experiment on the model. Clearly, any model is valid for the 'null experiment', while no model is valid for all possible experiments.

Summarising, we can define a model as an abstract system, representing a subset of the aspects of a real system. The relevancy of the aspects, represented by the model, is determined by the objectives of the model. The objectives are derived from the experiment that is applied to the model. An experiment is applied to the model to gain knowledge about the system.

Model representation

Unlike distinguishing some part of the universe as a system, a model has to be build in order to exist. Building a model means that a model has to be represented in some way. Simple models are often only represented in the engineer's mind. Such model representations, called conceptual models, can be used to think about a system but are difficult to share with others. Other model representations can exist on paper, are coded in software, or are shaped in three dimensions. These representations can be used in communication with others or to perform experiments on. The kind of representation used for a model depends on the application of the model and the environment in which the model is used. We will now discuss some characteristics of model representations for industrial systems.

Type of symbols

A model is represented by symbols where each symbol has its meaning. The set of symbols together with the rules of application of the symbols is called

a modelling language. Depending on the kind of symbols, used in a language, modelling languages fall into one of the following categories:

- iconic languages
- mathematical languages

Iconic languages use graphical symbols to express the aspects of a system. Examples of iconic models are construction drawings, electrical circuit diagrams and sheet music. Iconic languages are widely appreciated in systems engineering because of the power to express complex problems in simple abstract drawings. Certain aspects of systems, such as the ordering of and relations between the elements of a system, are more easily expressed in graphical symbols than with an equivalent textual notation.

Mathematical languages use textual symbols that represent numbers, variables or other entities to express relationships or quantities that satisfy particular conditions. Examples of mathematical languages are differential algebra, tensor algebra, and set theory. An advantage of mathematical models is that they can be subject to calculations or symbolic manipulations. The latter are often used to reduce the number of symbols in a model or to emphasise certain relations between the model symbols.

Symbol sets

The applicability of a language depends on the possibilities to properly express the desired aspects of a system that are needed for particular objectives of a model. The language must be rich enough to express what needs to be expressed. If, for instance, a language only provides symbols to represent boxes with their sizes, it will be impossible to distinguish between boxes of different colours. In this case the symbol set of the language does not fit to its application.

On the other hand, the language should not contain too many symbols. A rich symbol set requires a substantial effort to learn the language and, because of the limited use of modelling languages in everyday life, this would be impractical. Natural languages are examples of rich symbol modelling languages and it is clear that it takes much effort to learn such a language. Also, a rich symbol set may introduce the possibility to express the same aspect in more than one way. Moreover, if a language provides too many symbols, the danger of creating ambiguous models increases. Again, the language is not appropriate for its application area.

Another issue, concerning the language symbol set, is that the available set of symbols for representing a model influences the way in which we perceive a system. For example, the experts in the field of dressage have a much more

elaborate vocabulary for describing the performance in a dressage test than has the average spectator. The average spectator simply ignores some of the subtleties of the movements of the horse and its rider because of the lack of symbols to express them.

Finding the proper set of symbols for a certain purpose is an evolution process in which the symbol set is constantly adapted to fit to the application for which it is designed. An example of evolving languages can be found in the field of computer science. Starting with assembly language, programming languages evolved to higher order languages, capable of expressing algebraic calculations and abstract data types. Compared with the assembly languages, the higher order languages provide a symbol set that is more closely related to the perception of the mathematically minded end-user. The language thus fits better to the field of application. Furthermore, the increase in expressive power accompanies a decrease in learning effort.

Structure

Industrial systems tend to be too complex for humans to perceive them as a whole. A modelling language must, however, provide the means to represent a system of arbitrary complexity. The resulting model must be manageable by the human mind. To resolve this problem of complexity, the system can be decomposed into smaller (sub)systems and thereby decrease the complexity of a system.

However, the relations between the subsystems must still be clear to enable, to some extent, the perception of the system as a whole. This implies that a representation of a complex system involves a hierarchy of abstraction levels where the higher-levels hide some of the details of the lower-levels. The relations between the different levels must however be preserved and must be visible in the model. Furthermore, the language should provide some means to keep these relations consistent.

The higher levels of abstraction often represent the structure of a system, where the lower levels represent the (atomic) elements of a system. The structure of a system is best represented in a graphical way and thus a modelling language should provide iconic symbols for this level of abstraction. The elements described in the lower levels of a model are best represented using mathematical symbols. Consequently, a language for the representation of models of industrial systems should contain both iconic and mathematical symbols.

2.3 Modelling industrial systems

In this section, we will describe the role of models in the life-cycle of industrial systems. An emphasis is put on the design process of systems and the application

of models in this process. Finally, we present a method that can be used to create models in the modelling process.

Life-cycle of industrial systems

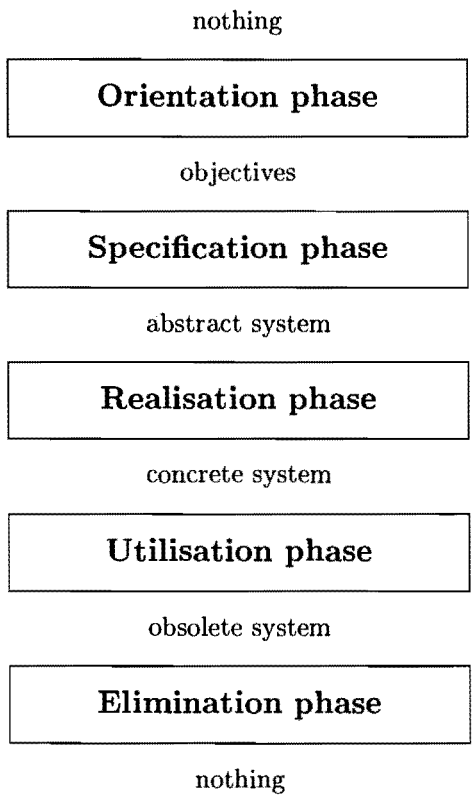


Figure 2.1: The life-cycle of an industrial system [Roo91].

An industrial system has a life-cycle in which five phases can be distinguished [Roo91]. The phases are depicted in Figure 2.1. The life-cycle begins with nothing. In the orientation phase, the objectives are defined for a system that does not yet exist. These objectives are initiated by the need for the transformation of materials, the task of the system to be designed. The orientation phase is characterised by the awareness of this need.

After having defined the objectives, the system is designed in the specification phase. In this specification phase, three subphases can be distinguished:

- The function determination phase.
- The process determination phase.
- The resource determination phase.

The result of the design or specification phase is an abstract system or model. This abstract system is a specification that defines what functions have to be performed by the system, how these functions are performed and what resources are required to perform the functions. A structured design method for industrial systems can be found in [Bra93].

In the realisation phase, the system is built according to the specifications. The result of this phase is a concrete system that serves the objectives, stated in the orientation phase and that meets the specifications as set up in the specification phase.

The concrete system is used in the utilisation phase to a point where either the system does no longer perform according its objectives or to a point where the objectives have changed. Either way, the system has become obsolete and the utilisation phase has ended.

In the elimination phase, the obsolete system is eliminated. Optimally this results in nothing. During the specification phase, it is the responsibility of the designers to allow a safe elimination regarding economic and environmental aspects.

It is obvious from the life-cycle of industrial systems that models are used during the specification phase. Less obvious is the use of models during the realisation phase and utilisation phase. Yet, also in these phases, models can help in understanding and communicating about the system. Even in the elimination phase, we can benefit from the use of models. Nevertheless, we will concentrate on the modelling of an industrial system in its specification phase.

The modelling process

In the specification phase, models are used to evaluate design decisions and to help solving problems that arise during the specification of industrial systems. A well-established method for evaluation and problem-solving is the simulation of the dynamic behaviour of a system. It is not self-evident that modelling and simulation is the only and best method of solving a problem. For now, however, we will concentrate on this method. A number of steps are to be taken for a simulation study to be successful.

Before building a model, we have to define the problem to be investigated. This problem statement is essential. A number of questions can be posed, concerning

the system, that have to be answered. An experiment is defined by which the answers to the problem can be attained. From the experiment definition, the objectives for the model can be derived. Knowing the objectives, we can start the simulation study. This study can be subdivided into a number of steps that are to be taken to allow a simulation study to be completed successfully. Rooda [Roo91] proposes the following steps:

- system identification,
- system specification,
- data collection,
- model implementation,
- model verification,
- model validation,
- experimentation,
- output analysis.

In the first step, we will identify the system by separating it from its environment. The system identification determines the system boundaries and defines the relations with its environment. The second step, the system specification, involves the building of a model of the system. The model represents the relevant aspects of the system for the given objectives. In the next step, data collection results in a statistical representation of the relevant data for the model. This statistical representation reduces the data to a manageable amount. After implementing the model (for instance, by coding it in software), the implementation is verified to operate correctly. In the model validation step, the consistency between the behaviour of the model and the behaviour of the system is checked. Finally, experiments with the model can be performed where-after the output of the experiments can be analysed.

In this dissertation, we will focus on the first two steps, with emphasis on the second step: system specification. This step can, again, be divided into a number of substeps. In the following, we will address the steps to be taken in the modelling process.

The first step in the modelling process is to identify the input and output quantities that are relevant for the model, where the relevancy is determined by the objectives. From these inputs and outputs of the system, we can identify elements in the environment that have relations with the system. These elements will (partly) be included in the model, so that the model is a closed system. The next step is to define the structure of the model. The elements of a model are

related to each other and these relations determine the ordering of the elements. This ordering is a hierarchical ordering which results in the definition of subsystems in the model. The definition of the structure also defines the amount of detail at various levels. Note that the amount of detail at the lowest level depends on the objectives of the model.

The material, information, and values, flowing through the system, define the relations between the system elements. These flows are represented by object flows in the model. The next step in the modelling process is to identify the different objects and to define these objects in terms of the language in which the model is represented. This step is often referred to as defining the data structures of a model. After defining the objects, the relations between the model elements can be defined.

The final step in the modelling process is to complete the model with the description of the behaviour of the model elements. This behaviour involves the transformation of the objects and preserving the flow of objects through the model.

It is unrealistic to think that this process can be completed without iteration. After each step, an evaluation step must take place that may result in repeating previous steps and thereby optimising the model to a point where the model fits to the desired objectives. Building a model is a design process and should be treated as such. The choice for the steps given here is an arbitrary one, yet a practicable one. It will be used in the chapters that present examples of models of industrial systems.

2.4 Summary

The concepts of systems and models have been discussed in the previous sections. A review of systems theory has led to the definition of a system and of an industrial system in particular. An industrial system has been defined to comprise of three subsystems. The primary system involves the flow of material. It is recognised that the transformation of material is the primary objective of an industrial system. The secondary system involves the flow of information which controls the material flow. This second aspect system is identified as the control system. The flow of values is responsible for the preservation of the industrial system, and can be seen and the 'energy' for the system. This value flow is characteristic for the tertiary system.

An introduction to modelling theory illustrated the important role of models in modern science. A definition of a model has been given and the relation between models, systems, and experiments has been discussed. Some requirements for model representation have been summarised with the emphasis on models of industrial systems. A model representation consists of symbols. A

modelling language has been defined as a combination of the set of symbols and their semantics. Iconic and mathematical languages are distinguished. For the representation of the model's structure, iconic languages are considered to be preferable over mathematical languages. For the description of dynamic behaviour, a mathematical language is advised. Finding the proper symbol set of a language for a particular field of application is a process of evolution. Adapting the symbol set to fit to its application is considered to be essential in language development.

The relation between models and industrial systems has been illustrated with the life-cycle of a system. We have focused on the role of models in the specification phase of a system. In this phase, models are used to support decision-making and problem-solving. Simulation studies have been mentioned as a helpful tool for the evaluation of design decisions. A method has been presented for the modelling process in relation with a simulation study. As an introduction to the definition of a formalism for the specification of industrial systems, we will illustrate this method with an example of a bottle filling system in the next chapter.

Chapter 3

Specification Requirements

The modelling process, as presented in the previous chapter, is a design process that results in a model for a given system. A model can help to solve problems concerning the system or it can be used to gain (more) knowledge of the system. The information, represented in a model, is determined by the purpose of the model. It is therefore essential to first state the objectives for a model before starting the modelling process.

The question arises how to determine which aspects of a system should be represented in a model to satisfy the objectives of the model. Answering this question can help us to find the requirements for a formalism in which models are represented. It is, however, not the subject of this dissertation to answer this question. Moreover, finding the answer is all but a trivial matter. Yet, we can find out the main characteristic aspects of modelling a system's behaviour without asserting whether they satisfy given objectives or not. We can compare this problem with designing a programming language. There, we can decide to define a set of language constructs without knowing in advance what programs will be written with the language.

We introduce a bottle-filling system as an example of an industrial system and use this system to illustrate some issues that are encountered in the modelling process. From these issues, we then formulate the requirements for our specification formalism.

3.1 The bottle-filling system

In this section, we present an example of an industrial system: a bottle-filling system [Mel86, Ove87]. Its purpose is to produce filled bottles, of two sizes, and

to deliver them to customers. A customer places an order for the number of bottles required. The bottles are then delivered according to this order. The bottles are filled with a liquid having a certain acidity. The system consists of two bottle-filling lines that operate concurrently. Both lines are fed by a single vessel, containing the primary product to be bottled. Filled bottles are packed before they are delivered to the customer. Figure 3.1 shows the vessel and one of the bottle-filling lines.

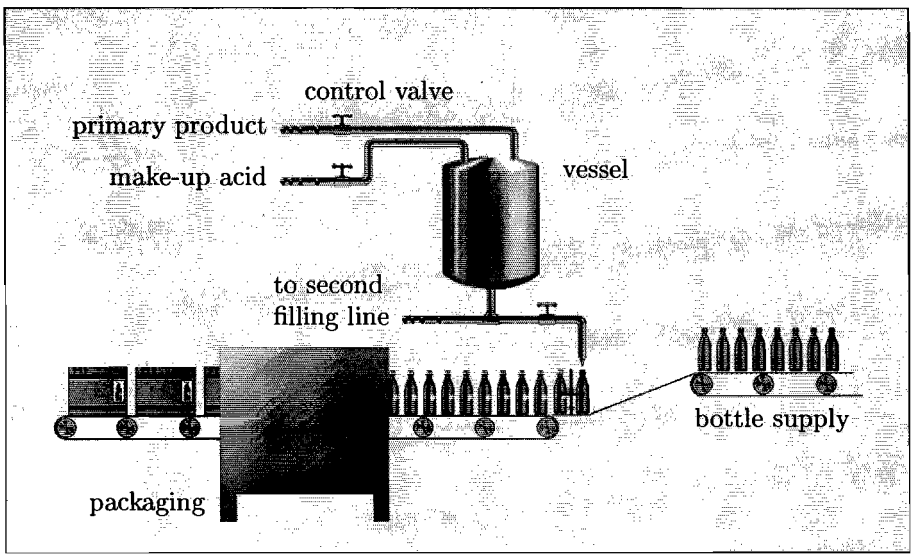


Figure 3.1: The bottle-filling system.

The vessel

The vessel contains an amount of liquid that is kept between a minimum and a maximum value. The maximum capacity of the vessel is about 600 liters and its diameter is 50 cm. Thus the maximum level in the vessel is about 76 cm. If the amount of liquid in the vessel drops below the minimum value, the vessel is filled with a primary product of a predetermined acidity or pH value. The primary product is supplied through an input valve. At the maximum level, the filling is stopped. The product is unstable and changes its pH when it is exposed to air. A make-up stream of acid allows for the necessary adjustment of the product quality in the vessel before it is used to fill the bottles.

The control system of the vessel has two tasks. The first task is to keep the liquid level between its minimum and maximum values by opening and closing the input valve. This input valve has two states: completely open and completely closed. The amount of liquid is measured by a sensor in the vessel. The second

task is to maintain the pH value of the liquid within given tolerance limits. Due to the unstability of the product, this pH value changes over time and can be adjusted by supplying a make-up acid to the vessel. The valve, through which the acid is supplied, has three states: completely open, completely close, and an intermediate state which we call 'dribble'.

The bottle-filling line

Both bottle-filling lines are identical, though different bottles are processed on each line. Two sizes of bottles can be filled: small bottles with a volume of 1-litre and large 5-litre bottles. Because both lines use the same vessel as the source for the liquid, the composition of the liquid used by the two lines is identical at any point in time.

Each line draws bottles from a bottle supply unit that releases bottles on request. The empty bottles slide down a chute from the supply unit to the position where they are filled. The arrival of a new bottle is detected by a sensor. After a bottle has been filled, a label is attached to the bottle that shows the actual pH and volume contained in the bottle. Finally, the bottle is removed from the bottling line and is transported to a packaging machine. Each line has its own control system that controls the filling of bottles on that line. All signals (bottle release and arrival, bottle level sensor, etc.) are handled by the corresponding control system.

Customer orders

The orders received from a customer are split into two separate orders, one for each bottling line. After all bottles have been filled for a given order, the bottles are wrapped and delivered to the customer. There is one packaging machine that wraps up the filled bottles of both lines. The bottles for a single customer order are collected and stacked together on a pallet. Note, that bottles should only be filled if the pH of the product in the vessel is within its tolerance limits. If this is not the case, the filling process is suspended and the pH is first adjusted. Then the filling of the bottles can be resumed.

3.2 Modelling issues

Before setting up a model of the bottle-filling system, the objectives of the model must be defined. Some possible objectives for a model are:

- Determine the number of orders that can be completed in a certain time interval.

- Determine the delivery time of an order. This is the time between the placement of an order and the time of delivery of the bottles.
- Determine the through-put time of an order. This is the time between the start of an order and the time at which the order is completed.
- Determine the degree of capacity utilisation of a bottle-filling line. The subsystem with the highest degree is possibly the bottleneck of the system.
- Try out different control strategies. What happens with the above mentioned values if another control strategy is used?
- Try out different configurations of the system. For example, what happens if not only one vessel is used, but two (one for each line) instead?
- Investigate the behaviour of the system. Try to increase the knowledge about the processes that take place in the system.

In the previous chapter, we have seen that defining an experiment and stating the objectives for a model is an essential step in a modelling study. Different objectives lead to different models, since the objectives determine what is represented in the model. Without restricting ourselves to one of the mentioned objectives, we address some general modelling issues that are to be considered when modelling industrial systems. We do not intend to be complete, but merely try to illustrate what aspects are relevant when making a model to investigate the behaviour of a system.

System identification

The first step in the modelling process is to identify the system. In this step, the system boundaries are determined and the relations between the system and its environment are defined. In the case of the bottle-filling system, the system can be identified to contain the vessel, the two filling lines and the packaging machine. Furthermore, the system contains the necessary control systems. The system's environment contains the customers and the suppliers of the raw materials: empty bottles, pallets, liquid, and so on. The bottle-filling system has now been identified as an open system; it has relations with its environment.

But, did we really identify an open system? Clearly, the bottle-filling system is an open system, but we also distinguished the customers and suppliers of the system. Furthermore, we did not define relations between these customers or suppliers with other entities in the universe. In fact, we identified a closed system, containing the bottle-filling system and some relevant aspects from its environment. Irrelevant aspects, such as relations of customers to other entities are left out. These relations are irrelevant according to the given objectives for the model.

To investigate the behaviour of an industrial system, we study the response of the system to the different stimuli from its environment. To be able to define these stimuli, we must include the entities that provide the inputs for the system in the model. Also, the system may influence its environment. Thus, the entities, affected by the system should also be included in the model. The resulting model represents a closed system, comprising an (open) industrial system and the relevant entities in its surroundings.

System structure

The identified system is considered to consist of three subsystems: the primary, secondary, and tertiary system. The structure of the system is determined by these three subsystems and is essentially a hierarchical structure. Two hierarchies can be distinguished in the specification of a system, the model hierarchy and the system hierarchy.

Model hierarchy

The model hierarchy defines the structure of the model. In this hierarchy, a system is composed of elements: system components or subsystems. A subsystem is again composed of subsystems. Each level in this hierarchy represents a certain level of abstraction.

The level of abstraction determines how much detail is described by a model and it depends on the stated objectives for the model. So far, we have described how bottles are being filled in an operational fashion. We did not describe how sensors function or how the packaging machine is constructed. Also, we did not describe whether the customer orders are handled by humans or by an automated information system. We described the bottle-filling system at a certain level of abstraction which we feel is consistent with the stated objectives. We are, for instance, not interested in how a sensor works but it is sufficient to recognise that a signal is available when a sensor becomes activated.

The level of abstraction is also determined by the existence of parallelism. If two components of a system operate concurrently we prefer to specify these components separately and define their interactions with relations between the components.

The hierarchical ordering of subsystems enables us to gain a clear overview of how the system is constructed. Without falling into details, the model hierarchy offers an abstract representation of the structure of a system at various levels. When we speak of the bottle-filling system, comprising a vessel, two filling lines and a packaging machine, we are not interested in the details of the filling lines. The model hierarchy hides the details of lower-level specifications.

Together with the model hierarchy, also modularity is achieved. This modularity enables the specification of a system component without worrying about other components that have relations with this component. When, for instance, we set up a specification of a filling line, we do not need to have a detailed insight in the behaviour of the vessel. Only the relations with the vessel are of interest.

Modularity also enables the parameterisation of a model. For example, the two filling lines are identical except for the size of bottles that are being filled on a line. A modular model provides the possibility to make one model for both filling lines and adapt each line to its specific characteristics by supplying the proper parameters (the bottle size, for instance).

System hierarchy

An industrial system is characterised by the transformation of material, information, and compensating values. These three flows through the system are reflected by the architecture of the system. We can distinguish three kinds of elements in a system, where each kind is responsible for the transformation of one of the three flows. Rooda [Roo83] defines an architecture for industrial systems in which these three flows are clearly visible (Fig 3.2). The main flows through a system are the material and value flows. The material flow is determined by the main objective for the system, the value flow assures the preservation of the material flow, and thus the entire system. The information flow can be divided into several subflows:

- Order. A request for products, semi-products, or raw material.
- Material tag. Information accompanying the delivered products.
- Invoice. Request for compensating values (money).
- Value tag. Information accompanying the delivered compensating values.

The architecture of a system determines the system hierarchy. In this hierarchy the system components, responsible for the transformation of information flows through the system, form the control systems. The material and value transformation processes are the controlled systems. The relation between these two categories of transformation processes is like the relation between an employer and an employee; a boss-servant relation. This type of relations is characteristic for the system hierarchy.

The system hierarchy and model hierarchy are independent of each other. A system hierarchical relation can be found at several levels in the model hierarchy, as can be seen in for instance job-shop manufacturing environments. Also, a model hierarchy can be used in the specification of any system. For instance,

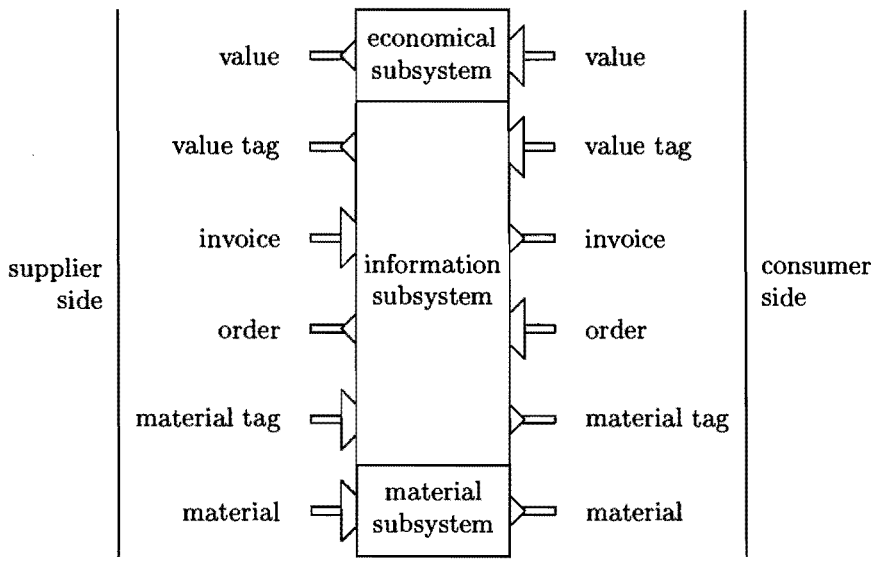


Figure 3.2: The system's architecture model [Roo83].

a control system can be decomposed in different (concurrent) control tasks. Furthermore, the controlled system can, in its turn, be composed of different parts.

Data structure

The input and output of an industrial system consist of the material, information, and compensating value flows. In our example system, we can identify the material flow to contain bottles, pallets, liquid, etc. The information flow consists of customer orders and probably delivery information, accompanying the delivered pallets. The economical system is not considered and is therefore missing in the model.

A specification of the bottle-filling system must include the specification of the various material and information objects. These can be represented by data structures that contain the relevant information about the object they represent. The data objects in a specification are hierarchically ordered. For instance the object pallet contains boxes, and these boxes contain bottles. This hierar-

chy, called the data hierarchy, must be made clear in the specification of these elements.

Behaviour

The behaviour of a system is described by the concurrent behaviours of the system components. A component can have a discrete behaviour, a continuous behaviour, or both. Whether a behaviour is modelled as a discrete or a continuous behaviour depends on the desired abstraction or granularity for a component and on the kind of material, information, or compensating value that has to be transformed.

In the bottle-filling system, the vessel's main function is to store and mix the incoming liquids. The material flow for this component is a continuous flow (of liquids) and its behaviour is therefore best described as a continuous behaviour. The packaging machine, however, receives bottles and 'transforms' them into boxes. Both input and output of this component are discrete objects. Therefore, the behaviour of the packaging machine is considered to have a discrete behaviour. A mixed continuous and discrete behaviour can be found in the filling lines, where liquid is coming in as a continuous flow, and the bottles that enter and leave (empty and filled respectively) are discrete objects. A specification of a filling line, therefore, exhibits both a continuous and a discrete behaviour.

The above mentioned examples of component behaviours are determined by looking at the type of input and output of the component. However, the choice for a certain behaviour can also depend on the granularity used to describe the component. For example, the opening and closing of a valve can be modelled as to occur at a certain point in time, but it can also be modelled as a gradual change of the aperture of the valve. In the first case, the behaviour is discrete, in the latter case it is continuous.

The kind of relations between system components is determined by the kind of behaviours of the components. If two components have a discrete behaviour, it is obvious that a relation between these components has discrete characteristics. A similar reasoning can be applied to a relation between continuous behaviours. Less obvious is a relation between a continuous and a discrete behaviour. Such relations are known as discontinuities and state-events. A discontinuity is a discrete change in an otherwise continuous behaviour. A state-event is an event in the discrete behaviour of a system, induced by the continuous behaviour of the system.

A specification of a system's behaviour is an idealised representation of the actual or required behaviour of a system. The actual behaviour is normally not a deterministic but a stochastic behaviour. For example, machines can fail, bottles

can break, and the number of bottles ordered by a customer fluctuates. This stochastic behaviour is the cause of many difficulties in analysing a behaviour of a system. A specification, unable to describe stochastic behaviours, would be highly impracticable, if not worthless.

3.3 Requirements for a specification formalism

In the previous sections, we have illustrated some modelling issues using an example of a bottle-filling system. In this section we set-up the requirements for a specification formalism that enables the creation of specifications with consideration of the mentioned aspects. The requirements can be categorised as:

- general requirements,
- domain-specific requirements,
- implementation requirements.

General requirements

The general requirements consider the language for the representation of models. Some aspects of model representation have been discussed in Chapter 2. These aspects can be summarised by:

- *Type of symbols.* The language symbols can be iconic or mathematical.
- *Symbol set.* The choice for a symbol set influences the expressive power of the language, the effort needed to learn the language, and the perception we develop when using the language.
- *Structure.* The language must enable the specification of hierarchical structures like the model, system, and data hierarchies.

Although the hierarchical structures, mentioned in the previous section, are possibly best described using iconic symbols, we prefer the use of mathematical symbols only. Mathematical symbols have the advantage of leading to more compact and unambiguous specifications than do iconic symbols. We do, however, allow the use of iconic symbols in addition to mathematical symbols for illustration purposes, especially for representing the model structure. A translation from mathematical symbols to iconic symbols is easier and more accurate than a translation in the opposite direction.

Another issue that has to be concerned, regarding the choice for mathematical or iconic symbols, is the acceptance of the formalism. There are many disciplines involved in the design process of industrial systems, where each discipline has its own established iconic languages. If the formalism is to be used in as many disciplines as possible, it should allow the use of these existing iconic representation methods. We believe that a mathematical language will experience a higher level of acceptance than an iconic language. Most systems engineers are familiar with mathematics and are expected to have little difficulty in learning a new set of mathematical symbols. Replacing an established iconic language with a new one, however, will encounter considerable resistance.

Finally, although it may seem of minor importance, mathematical symbols are easier to produce using today's word processors than iconic symbols, which would require more dedicated tools.

The symbols of a language represent the concepts that can be expressed by the language. A concept (and its representational symbol) is called primitive if the concept cannot be expressed using other symbols of the language. We strive for a minimal number of primitives where the primitives form an orthogonal set of concepts. All primitives are essential language constructs and should together cover the domain of application of the formalism.

Additional (non-primitive) symbols can be defined to increase the expressive power of the language. Such symbols are often referred to as 'syntactic sugar'. The definition of such symbols should, however, be well considered to avoid proliferation. The latter would decrease the acceptance and thus the use of the formalism because of the increasing learning effort to get familiar with the additional symbols.

Domain-specific requirements

The domain-specific requirements focus on the set of concepts that can be expressed by the formalism. A number of these concepts have been mentioned in the previous section. We will now give a summary of these concepts in the form of requirements for the formalism.

The structure of a system and the objects that exist in it must be reflected in the system's specification. Both system and data structures are essentially hierarchical structures. Thus, the specification formalism must provide language constructs for a hierarchical decomposition of the system and the objects contained in it.

The decomposition of a system leads to a (hierarchically ordered) collection of system components. These components operate concurrently and together they determine the behaviour of the system. The specification of this concurrent behaviour of system components demands a concept of parallelism in the

formalism. We refrain from defining parallelism within a single component. Such parallelism can be modelled by further decomposition in (sub)components. Thus, the formalism should provide a parallel composition of system components where each component does not contain any parallelism.

Modularity allows a piece-wise development of specifications. In setting up a specification of a subsystem, only the relations with other subsystems have to be considered. Furthermore, a module can be made generic with the use of parameters. These generic specifications tend to shorten a modelling process by the re-use of existing specifications. Summarising, we require the formalism to enable the definition of parameterised specifications in a modular way.

Implementation specific requirements

This research focuses on the development of a formalism and not on possible assisting software tools. Yet, we take these future developments into account by stating some requirements that should contribute to the implementation of such tools. The most important (and most obvious) requirement is the definition of the operational semantics of the formalism. The semantics may serve as an operational specification for a simulation engine that executes specifications set up by the formalism. Recent developments on this subject can be found in [Mor95b].

A less important requirement deals with the implementation of data structures. Most programming languages use a concept of strong typing that allows a syntax and type check before compilation or execution is started. Even if an implementation does not require strong typing, we still require strong typing for the formalism. This provides a means to check the consistency of models.

3.4 Summary

The bottle-filling system is presented as an example of an industrial system. Some modelling issues are discussed using this example. The definition of the objectives for a model are considered to be essential when building a model. The modelling process consists of a number of phases. In the system identification phase, the system and its environment are determined as well as the interactions between them.

Two hierarchical structures are recognised: the model hierarchy and the system hierarchy. The model hierarchy determines the abstraction levels in the model. The system hierarchy involves the relationships between control systems and controlled systems. The architecture model is presented to clarify these relationships. The third hierarchy that can be distinguished is the hierarchy of the

data structures in a model. The data structures represent the various objects that comprise the three flows of material, information, and compensating values. The last step in the modelling process is the specification of the behaviours of the system components.

From the discussed modelling issues, the requirements for a specification formalism are derived. The general requirements concern the specification language. A language is defined by its symbol set and their associated semantics. The domain-specific requirements involve the choice for certain concepts that the formalism must enable to represent. Finally, some requirements are presented that may contribute to the implementation of a supporting software tool for the formalism.

Chapter 4

The Formalism χ

In this chapter, we introduce the formalism χ for the specification of (industrial) systems. The formalism offers a language in which the behaviour of systems can be described. The language provides constructs according to the requirements mentioned in the previous chapter. We define a minimal language that forms the basis of our formalism χ. Other language constructs are defined using these primitive language constructs. The first section presents some basic concepts of the formalism, followed by the definition of the minimal language. The remaining sections describe additional language constructs that, together, result in the definition of the formalism χ.

In this chapter, we restrict the definition of the formalism to an informal description of the syntax and semantics of the language. We refer to Appendix A for a formal definition of the syntax of χ.

4.1 Basic concepts

The approach to the modelling of systems with the formalism χ is derived from the process-interaction approach [Roo82b, Ove87] and real-time concurrent programming [Bur93]. In our approach, as in both mentioned approaches, a system is considered to consist of a set of processes. Each process models a system component and the processes operate concurrently. Although this approach was originally developed in the field of discrete-event modelling, we will use it as the basis of our formalism χ, to specify both the discrete and continuous behaviours of a system.

The relations between the processes are modelled by fixed channels that connect the processes. Different kinds of channels are distinguished for discrete and continuous relations. All channels are one-to-one connections between processes.

A hierarchical structure is achieved by grouping processes into an entity called a system. Such a system can act as a process. It can be combined with other processes and systems to form a new system. The resulting top-level system forms a specification of the modelled system. Although, here a bottom-up approach has been used to illustrate the relationships between processes and systems in a specification, the formalism does not prescribe any approach for the modelling of systems. It allows for both a top-down and bottom-up approach as well as for a centre-out approach or for any combination of these.

A specification forms a description of the behaviour of a system. The state of the system is determined by the states of the processes that comprise the system. This state changes when time passes. The state of a process is recorded in process variables. All variables in a specification are local variables, defined in processes. Global variables do not exist.

The behavioural description of a process can be discrete or continuous or a combination of the two. The description of a discrete behaviour has the form of a sequential program where changes in the state of a process are accomplished by performing actions. The continuous behaviour of a process is defined by a set of differential and algebraic equations, expressed in the process' state variables.

To visualise the structure of a model, a graphical representation can be used. Here, a process (or system) is represented by a circle with the process name centred in this circle. A channel is visualised by a (curved) line between the processes that are connected by the channel. The line can be supplied with an arrow head or with one or two open dots to indicate the type of channel. Somewhere along the line, the channel's name can be given. The graphical representation is not an essential part of the formalism but it is a helpful means in structuring system specifications.

4.2 Minimal language

The minimal language presented here forms the basis for the formalism χ. It is the minimal subset of χ by which systems can be specified. Based on this minimal language, the formalism χ is described in the sections that follow. The syntax used to define the minimal language is a simplified version of the syntax of χ.

In the sequel, we use the following identifiers to denote variables, channels and other entities that occur in the definition of the formalism:

$$\begin{aligned} \vartheta &\in \mathcal{V} \text{ (domain of values)} \\ \tau &\in \mathcal{T} \text{ (time)}, \mathcal{T} \subseteq \mathcal{V} \\ i, j, k, m, n, N &\in \mathbb{N} \text{ (natural numbers)}, \mathbb{N} \subseteq \mathcal{V} \end{aligned}$$

$$
\begin{aligned}
c, c_1, \ldots, c_n &\in \mathcal{C} \text{ (channel names)} \\
x, x_1, \ldots, x_n &\in \mathcal{X} \text{ (discrete variables)} \\
u, u_1, \ldots, u_n &\in \mathcal{U} \text{ (continuous variables)} \\
u', u'_1, \ldots, u'_n &\in \mathcal{U}' \text{ (derivative variables)} \\
b, b_1, \ldots, b_n &\in \mathcal{B} \text{ (Boolean values)}
\end{aligned}
$$

We use the following conventions in the definition of our minimal language. These conventions do not hold for the formalism χ.

- Although all variables are local, we use unique global identifiers for the variables of processes. No two processes, however, may refer to the same variable.
- Channels are defined globally and if two processes refer to a channel identifier $c \in \mathcal{C}$ they refer to the same channel.

Systems and processes

The formalism is based on the parallel composition of sequential processes. Each process specifies the behaviour of a system component. A system is defined with:

$$Y \quad ::= \quad [\![P_1 \parallel \cdots \parallel P_n]\!]$$

where $P_1, \ldots, P_n$ are the processes of the system. A process can have two kinds of behaviour: discrete and continuous. We formally define a process as:

$$P \quad ::= \quad [\![E \text{ '|' } S]\!]$$

where E denotes the continuous behaviour of the process and consists of a set of differential and algebraic equations. S is a sequential program that describes the discrete behaviour.

Note that in our minimal language, a model of a system consists of a single system. All system components are modelled by the processes of this system. Thus, the minimal language does not provide hierarchical modelling.

Variables and values

The description of a process' behaviour forms a formal definition of how the state of a process changes in time. The state of a process is stored in the process' memory. This memory is constructed with variables. Because of the differences between a discrete and a continuous behaviour, different variables must be used

to memorise the state of a process. In a discrete behaviour, the process' state changes abruptly at certain points in time. Therefore, the values of variables will change abruptly too. A continuous behaviour, however, defines a continuously changing process state. To record these state changes, the values of variables must be able to change continuously. We therefore distinguish between discrete and continuous variables.

In our minimal language, discrete variables can have numerical values only, and not, for instance, strings or characters. In the sequel, we use the identifiers $x, x_1, \ldots, x_n \in \mathcal{X}$ to denote discrete variables. Since a continuous variable can be used in differential equations, it must also be able to keep track of the value of at least its first time derivative. Furthermore, continuous variables are restricted to have a numerical value. With the identifiers $u, u_1, \ldots, u_n \in \mathcal{U}$ we denote continuous variables and the identifiers $u', u'_1, \ldots, u'_n \in \mathcal{U}'$ represent their first time derivatives. For continuous variables and their first time derivatives we require:

$$u \in \mathcal{U} \Leftrightarrow u' \in \mathcal{U}' \wedge u' = \frac{\mathrm{d}u}{\mathrm{d}\tau}$$

We introduce a special variable τ that represents the process' time. It has a value $\tau \in \mathcal{T}$, where $\mathcal{T} = \{\, \tau \in \mathbb{R} \mid \tau \geqslant 0 \,\}$ is a dense time domain.

The possible numerical values of a variable (either discrete or continuous) are given by the set $\mathcal{V}$, where $\mathcal{V}$ is the domain of all (numerical) values. Furthermore, we define $\mathcal{T} \subseteq \mathcal{V}$ to allow calculations in the time domain.

Expressions

In both continuous and discrete behaviour specifications we use expressions to describe operations on variables and values. We define the following expressions for our minimal language:

$$\begin{array}{lcl} e & ::= & \vartheta \mid x \mid u \mid u' \mid e_1 + e_2 \mid e_1 - e_2 \mid e_1 \times e_2 \\ b & ::= & e_1 = e_2 \mid e_1 < e_2 \mid b_1 \vee b_2 \mid \neg b \end{array}$$

where $e, e_1, \ldots, e_n$ denote numerical expressions and $b, b_1, \ldots, b_n \in \mathcal{B}$ denote Boolean expressions. Furthermore, $\vartheta \in \mathcal{V}$.

Henceforth, we use the following abbreviations for expressions:

$$\begin{array}{lcl} \mathrm{true} & \equiv & 0 = 0 \\ \mathrm{false} & \equiv & \neg\, \mathrm{true} \\ b_1 \wedge b_2 & \equiv & \neg(\neg\, b_1 \vee \neg\, b_2) \end{array}$$

where $\equiv$ denotes syntactic equivalence.

Continuous behaviour and relations

The continuous behaviour of a process is described by a set of equations. An equation has the general form:

$$e_1 = e_2$$

where e_1 and e_2 are expressions as described earlier. If a process contains more than one equation, the equations are separated by commas:

$$Q \quad ::= \quad e_1 = e_2 \mid Q_1,\ Q_2$$

The continuous behaviour of a process is formally defined as:

$$E \quad ::= \quad \epsilon \mid E_1,\ E_2 \mid Q \mid H \mid L$$

where H and L denote guarded equations and variable links respectively. Both language constructs are described later. Furthermore, ϵ denotes an empty equation set with:

$$\epsilon,\ E \equiv E,\ \epsilon \equiv E$$

We allow the following abbreviation for processes that have no continuous behaviour:

$$[\![\ \epsilon\ \text{`|'}\ S\ ]\!] \equiv [\![\ S\ ]\!]$$

Guarded equations

Sometimes, the set of equations for a process is not fixed, but depends on the values of discrete or continuous variables. For example, a stationary liquid flow through a pipe is a laminar flow if Reynolds' Constant has a value $\mathrm{Re} \leqslant 2320$. Otherwise the flow is a turbulent flow. In these two cases, the behaviour of the liquid flow is described by two totally different sets of equations. Which set must be used depends on the value of Re, and this value, in its turn, depends on the values of the variables representing the flow. For this kind of situations we introduce the language construct of guarded equations.

A guarded equation is an equation (or a set of equations) that takes part in the behaviour description of a process if (and only if) a certain condition is fulfilled. The condition is represented by a Boolean expression. Formally, we define a guarded equation as:

$$H \quad ::= \quad [\,[\!]_{i=1}^{n}\, b_i \longrightarrow Q_i\,]$$

For the guards in a guarded equation, we demand that at least one guard is open:

$$b_1 \vee \ldots \vee b_n = \text{true}$$

If more than one guard is open, one will be chosen non-deterministically. In this case we require, however, that the behaviour is the same for all alternatives with an open guard.

As an example, the stationary flow through a pipe can be described by a guarded equation of the form:

$$\begin{array}{l}[\quad \mathrm{Re} < 2320 \longrightarrow Q_1 \\ [\!]\, \neg \mathrm{Re} < 2320 \longrightarrow Q_2 \\ ]\end{array}$$

where Q_1 is the set of equations describing the stationary laminar flow and Q_2 describes the turbulent flow through the pipe.

The construct of the guarded equation is derived from guarded commands, introduced by E.W. Dijkstra in [Dij74]. The guarded commands are also used in χ in the discrete behaviour of processes (see Section 4.2, Page 40).

Variable linking

A continuous relation between two processes is a relation between the states of the two processes. Or, more specific, a relation between the continuous (local) variables of the two processes. To define such a relation, the two variables are linked to a continuous channel. This channel then defines that the variables represent the same physical quantity and are said to be equal to each other.

We distinguish two kinds of physical quantities: vector and scalar quantities. A vector quantity has an associated direction. Examples of vector quantities are mass flow, force, and speed. Scalar quantities, like pressure and temperature, have no direction. A possible direction of a quantity is defined by the channel to which a variable, representing this quantity, is linked. Thus, we have channels with a direction (for vector quantities) and without a direction (for scalar quantities). For more details on the semantics of linking variables to channels, we refer to Section 6.2.

Linking a continuous variable $u \in \mathcal{U}$ to a channel $c \in \mathcal{C}$ is formally defined with:

$$L \quad ::= \quad c\updownarrow \multimap u \;\mid\; c\downarrow \multimap u \;\mid\; c\uparrow \multimap u$$

The expression $c\updownarrow \multimap u$ defines that the variable u is linked to channel c. Here, the channel c does not define a direction, which is denoted by the symbol $\updownarrow$. Thus, the variable u represents a scalar quantity. The expressions $c\downarrow \multimap u$ and $c\uparrow \multimap u$ define that the variable u is linked to an incoming ($c\downarrow$) and an outgoing ($c\uparrow$) channel respectively. Linking a variable to an incoming channel means that, if the value of the variable is positive, the vector quantity enters the process.

As an illustration of the use of variable linking, consider the following system S in which a liquid flows from a vessel A to a vessel B:

$$S = [\![\, A \parallel B \,]\!]$$

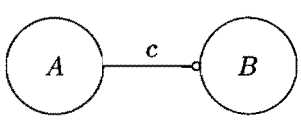
$$A = [\![\, c\uparrow \multimap \varphi_1,\ V_1' = -\varphi_1 \mid S_A \,]\!]$$

$$B = [\![\, c\downarrow \multimap \varphi_2,\ V_2' = \varphi_2 \mid S_B \,]\!]$$

where $\varphi_1 \in \mathcal{U}$ represents the outgoing flow of vessel A and $\varphi_2 \in \mathcal{U}$ represents the incoming flow of vessel B. Both variables represent the same flow. To define this relation between φ_1 and φ_2, both variables are linked to channel c. The relation between the variables φ_1 and φ_2, defined by the channel c, is $\varphi_1 = \varphi_2$. The current contents of the vessels are represented by $V_1 \in \mathcal{U}$ and $V_2 \in \mathcal{U}$ respectively. Observe that this implies that $V_1' \in \mathcal{U}'$ and $V_2' \in \mathcal{U}'$ and thus:

$$V_1' = \frac{\mathrm{d}V_1}{\mathrm{d}\tau} \qquad \text{and} \qquad V_2' = \frac{\mathrm{d}V_2}{\mathrm{d}\tau}$$

The system can be represented graphically as:



The graphical representation of the continuous channel can be seen as an arrow, except that the arrow head is replaced by an open dot. The direction of this 'arrow' indicates the direction of the channel. In our example, a positive value of the variables φ_1 and φ_2 means that the liquid flows from A to B. For relations involving a quantity without a direction, an open dot is placed at each end of the channel.

Discrete behaviour and relations

The discrete behaviour of a process is described by a sequential program. The statements that comprise such a program are defined by:

$$S \quad ::= \quad \varepsilon \mid x := e \mid S_1 ; S_2 \mid C \mid G \mid *G$$

The statement ε denotes the empty statement, with:

$$\varepsilon ; S \equiv S ; \varepsilon \equiv S$$

Execution of the empty statement terminates immediately without influencing the state of the process. In analogy with the absence of a continuous behaviour, we allow the following abbreviation for processes without a discrete behaviour:

$$[\![E \text{ '|' } \varepsilon]\!] \equiv [\![E]\!]$$

Furthermore, in the sequel, we also use the keyword skip to denote the empty statement:

$$\varepsilon \equiv \text{skip}$$

The statement $x := e$ denotes an assignment statement that changes the value of a variable x to the value of the expression e. Although we have used $x \in \mathcal{X}$ to define the assignment statement, both discrete and continuous variables are allowed as the destination for an assignment. Thus, also $u := e$ is a valid statement. Furthermore, execution of an assignment statement takes no time, i.e. the value of τ does not change.

The sequential composition of statements is defined by $S_1 ; S_2$, which means that statement S_2 is executed after the execution of statement S_1 has been terminated.

The statements C and G (with variant $*G$) denote event statements and guarded commands respectively. These statements are described in the following sections.

Event statements

The event statements comprise synchronisation, communication, time-passing, and state-events. Formally, these statements are defined by:

$$C \quad ::= \quad c^{\sim} \mid c?x \mid c!e \mid \Delta e \mid \nabla b$$

We introduce the name interaction to denote a communication or synchronisation statement.

Synchronisation A synchronisation action is performed on a channel c by execution of the statement $c^{\sim}$. If two processes perform a synchronisation action on the same channel, but at different times, the discrete behaviour of the process that first started the synchronisation statement is suspended until the other process performs its synchronisation action. If both synchronisation actions are performed at the same time, nothing happens, and the execution of both processes is continued.

Note that a synchronisation channel has no associated direction.

Communication Communication between two processes is similar to synchronisation, except that while processes are synchronised, a value is passed from one process to the other. The process that provides a value for a communication along a channel c executes the statement $c!e$, where the value of expression e is the value used in the communication. This process is referred to as the sending process. The other process, the receiving process, executes a statement $c?x$, where x is the variable whose value is changed to the value received in the communication. As with synchronisation, a process can be suspended if its communication partner is not yet ready to participate in the communication.

Communication can be seen as a distributed assignment. The value of expression e is assigned to variable x. A communication channel is unidirectional.

Time-passing Execution of the statement $\Delta\, e$ denotes time-passing . We also use the name delay for this statement. The discrete behaviour of a process that executes this statement is suspended for a number of time units equal to the value of expression e. The value of e is restricted to $e \geqslant 0$, where $\Delta\, 0 \equiv \varepsilon$.

State-event An event, caused by the continuous behaviour of a process, is called a state-event. The occurrence of such an event depends on a condition expressed in the continuous variables of a process. The discrete behaviour of a process can be suspended until a state-event occurs with the statement $\nabla\, b$, where b is a Boolean expression denoting the condition for a state-event.

Observe that the Boolean expression b must depend on continuous variables. Discrete variables do not change their values if a process is suspended and would therefore suspend the process forever. The continuous variables, however, change while time advances and can thereby trigger the condition for a state-event. Observe further that the state-event statement is similar to a time-passing statement, except that the time for which a process is suspended is normally not known at the time the execution of the state-event statement is started.

Guarded commands

Guarded commands were first introduced by E.W. Dijkstra [Dij74] and adapted to be used with communicating processes by J. Hooman in [Hoo91]. The guarded command we define in our formalism exist in two varieties. The first defines a selection statement, the second is used for selective waiting. For both types, we define a variant that makes the command repetitive.

Selection The selection statement has the general form:

$$G \quad ::= \quad [[]_{i=1}^{n} b_i \longrightarrow S_i \,]$$

where $b_1, \ldots, b_n \in \mathcal{B}$ are Boolean expressions and $S_1, \ldots, S_n$ are statements. The execution of a selection statement starts by evaluating the Boolean expressions, called the guards. A guard is called open if the Boolean expression evaluates to true. In a selection statement, at least one guard must be open. One of the open guards b_i is chosen non-deterministically and the associated statement S_i is executed.

Selective waiting A selective waiting statement has the form:

$$G \quad ::= \quad [[]_{i=1}^{n} b_i ; C_i \longrightarrow S_i \,]$$

where C_i are event statements as described earlier. The other elements are the same as in a selection statement. A guard is the part to the left to the $\longrightarrow$ and is called open if the Boolean part b_i evaluates to true. The statement is invalid if none of the guards is open. The selection for an open guard is determined by the earliest possible event statement that can be executed. If more than one event statement can be executed at the same time, the following priorities are applied:

1. *Interactions.* If one of the possible events is a synchronisation or communication, execute the alternative containing an interaction.
2. *State-events.* If no interactions are possible, then chose a state-event.
3. *Delays.* If no state-events occur, then select a delay statement.

If in one of these situations more than one alternative can be chosen, each having the same priority (for instance, two state-events occurring at the same time), select one non-deterministically. Note that including a time-passing statement in a guarded command, defines a time-out for the interaction statements and state-events.

Repetitive guarded command The guarded commands, both selection and selective waiting, can be made repetitive by preceding it with an asterisk:

$$*G$$

The repetition terminates if none of the guards in G is open.

We allow the following abbreviations to be used in guarded commands:

$$\begin{array}{ll} \text{true}; C & \equiv C \\ *[\ \text{true} \longrightarrow S\ ] & \equiv *[\ S\ ] \\ b; C \longrightarrow \varepsilon & \equiv b; C \end{array}$$

Observe that with the guarded commands, conventional programming constructs can be defined:

$$\begin{array}{ll} \textbf{if } b \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ fi} & \equiv [\ b \longrightarrow S_1 \ [\!]\ \neg b \longrightarrow S_2\ ] \\ \textbf{while } b \textbf{ do } S \textbf{ od} & \equiv *[\ b \longrightarrow S\ ] \end{array}$$

Also, a **case**-like statement can be constructed in a similar way.

Syntactic constraints

We assume the following restrictions for our minimal language. These restrictions also hold for the formalism χ.

- Communication and vector-type continuous channels are uni-directional.
- All channels are one-to-one connections.
- A continuous channel can only be linked once in a process.

4.3 Keywords

The first extension to the syntax of our minimal language involves the use of keywords to introduce the definition of specification elements. A specification consists of a number of such elements, such as systems and processes. The elements can be given in any order that seems appropriate. This gives a specification a declarative character rather than an imperative one.

In the sequel, keywords are printed in a sans-serif font style. This is not a prescribed rule but merely an aesthetical preference.. For the specification elements *system* and *process* we introduce the keywords syst and proc:

$$\textsf{syst } Y = [\![P_1 \parallel \cdots \parallel P_n]\!]$$

$$\textsf{proc } P = [\![\dot{E} \mid S]\!]$$

where Y and P are the identifiers of the system and process respectively.

Other keywords are presented in the sections to follow.

4.4 Variables and types

The specifications of objects, occurring in a model, must reflect the relevant aspects of their real-life counterparts. These objects are represented by one or more variables and the restriction, that a variable can only have a numerical value, may conflict with this requirement. In this section, we will introduce types for defining a domain of a variable. These domains no longer restrict values to be numerical but allow for more diverse values that increase the similarity between modelled objects and the variables that represent them.

Also, the limitation to simple variables prevents the data from representing hierarchical structures that occur in objects. To overcome this deficiency, compound data types are introduced that allow variables to be grouped into structures, thereby enabling the creation of hierarchical data structures.

Furthermore, we will allow new data types to be defined in terms of existing data types. A new type declaration starts with the keyword **type** and is followed by the definition of the type. A type declaration is of the form:

$$\textsf{type } t_1 = type_1, \ldots, t_n = type_n$$

where, for $i \in \{1, \ldots, n\}$, t_i is an identifier for the new type and $type_i$ is an expression that denotes how the type is constructed.

The type of a variable is defined with the declaration of the variable. A variable x of type T is declared with:

$$x : T$$

Since all variables are local variables, the declaration of variables occurs in a process definition. We extend the syntax of a process definition with a declaration section as follows:

$$\textsf{proc } P = [\![V \mid E \mid S]\!]$$

where V denotes the declaration of variables. If no variables are to be declared, the declaration section may be omitted.

Primitive types

A number of primitive types exist in χ where a primitive type defines the domain of a simple variable. In our minimal language, we defined the set $\mathcal{V}$ to denote all numerical values. The following primitive types represent subsets of $\mathcal{V}$:

Type	Denotation	Associated set of values
Natural	nat	$\{0, 1, 2, \ldots\}$
Integer	int	$\{\ldots, -3, -2, -1, 0, 1, 2, 3, \ldots\}$
Real	real	all real numbers

Variables of different numerical types can occur in the same expression. The value of the expression is coerced to the destination type required. The coercion rules we apply are similar to those found in many programming languages and are not addressed here in detail.

In addition to the arithmetical operations $+$, $-$, and $\times$, we allow all commonly used mathematical operations on numerical values. Thus the following expression is a valid expression in χ:

$$\frac{\sqrt[3]{x^2 - 2 \times x + y^2}}{x/2 + y}$$

Furthermore, we propose the operators // and \\ for Integer division and Modulo calculation respectively.

To define a variable that represents a Boolean value, we introduce the type bool. A variable of type bool has either a value true or false. In addition to the $<$ operation we define:

$$\begin{aligned} e_1 > e_2 &\equiv e_2 < e_1 \\ e_1 \leqslant e_2 &\equiv \neg(e_2 < e_1) \\ e_1 \geqslant e_2 &\equiv \neg(e_1 < e_2) \\ e_1 \neq e_2 &\equiv \neg(e_1 = e_2) \end{aligned}$$

Text values are represented by the types char and string, where char denotes the type Character whose values represent single characters. A value of type string represents a concatenation of characters. Examples of Characters and Strings are:

```
'a', 'B', '3'
"abc", "This is a string"
```

We allow the following abbreviations for Characters and Strings if no ambiguity arises because of their use:

'A' $\equiv$ A
"ThisIsAString" $\equiv$ ThisIsAString

A String is a special kind of list: a list of Characters. In the next section, we introduce a type list that can hold any kind of data type.

Lists

The first compound data type we define is the list. The list type is similar to the list type of Lisp and defines an ordered sequence of elements of some base type. A list may contain any number of elements. Also, duplicate elements are allowed. The type representing a list of elements of type T is denoted by T^*. The literal expression for a list is of the form $[e_1, \ldots, e_n]$. For example:

$[1,2,3]$	is of type nat*
$[1,-2,3,-2]$	is of type int*
$[[1,2],[3]]$	is of type nat**
$[]$	(empty list) is of type T^*, for every type T

For the list type, we define the following predicates and functions[1] as:

- $=$ — equality: two lists are equal if they contain the same number of elements and if their elements have equal values in each position.
- $+\!\!+$ — concatenation: joins two lists of the same base type, for example: $[1,2,3] +\!\!+ [3,2,1] = [1,2,3,3,2,1]$
- len — length: delivers the number of elements contained in the list: $\text{len}([1,2,3,3,2,1]) = 6$
- hd — head: delivers the first element of the list: $\text{hd}([1,2,3,3,2,1]) = 1$
- tl — tail: delivers the remainder of the list when the first element is removed: $\text{tl}([1,2,3,3,2,1]) = [2,3,3,2,1]$

[1] see Section 4.5 for an explanation of functions.

Tuples

A tuple is a structured data type for the aggregation of a fixed number of variables of possibly different types. The literal expression for a tuple is of the form $\langle e_1, \ldots, e_n \rangle$. For example:

$\langle 1, 0.3 \rangle$	is of type $\langle \text{nat} \times \text{real} \rangle$
$\langle 1, 2, 3 \rangle$	is of type $\langle \text{nat} \times \text{nat} \times \text{nat} \rangle = \langle \text{nat}^3 \rangle$
$\langle \langle -0.3, 1.6 \rangle, [1, 2], 3 \rangle$	is of type $\langle \langle \text{real} \times \text{real} \rangle \times \text{nat}^* \times \text{nat} \rangle$
$\langle -0.3, 1.6, [1, 2], 3 \rangle$	is of type $\langle \text{real} \times \text{real} \times \text{nat}^* \times \text{nat} \rangle$

Note that a tuple can contain zero or more elements. For example, an empty tuple is of type $\langle\rangle$. The abbreviated form in the second example may be used as a replacement for a number of successive occurrences of the same type. So, for instance the third example can also be defined by $\langle \langle \text{real}^2 \rangle \times \text{nat}^* \times \text{nat} \rangle$ and the fourth example by $\langle \text{real}^2 \times \text{nat}^* \times \text{nat} \rangle$. The constant, denoting the number of occurrences, must be greater then 1.

An individual element of a tuple is accessed by its index in the tuple. The index of an element in a tuple is zero based. For example:

let $tt = \langle 2, 3 \rangle$, then $tt.0 = 2$, and $tt.1 = 3$

This construct can also be used in an assignment statement to change the value of an element in a tuple, as in:

$tt.0 := 6$

For the tuple, the following functions are defined:

- lft — left: delivers the leftmost element of the tuple, for example: $\text{lft}(\langle 1, 2, 3 \rangle) = 1$
- rgt — right: delivers the rightmost element of the tuple, for example: $\text{rgt}(\langle 1, 2, 3 \rangle) = 3$

Units

The unit type is a numerical type similar to type real. It defines not just a single value, but also the values of associated time derivatives. The first time derivative of a variable u is denoted by u', the second time derivative by u'', and so on.

$$u' = \frac{\mathrm{d}u}{\mathrm{d}\tau}\,, \qquad u'' = \frac{\mathrm{d}^2u}{\mathrm{d}\tau^2}\,, \qquad u''' = \frac{\mathrm{d}^3u}{\mathrm{d}\tau^3}$$

The unit type is used for the declaration of continuous variables and channels. A unit type represents a physical quantity and is created by defining the unit of measurement in which the physical quantity is expressed. For example:

```
type  temp  = [K],          -- temperature in Kelvin
      area  = [m^2],         -- area in square meters
      press = [N·m^-2]       -- pressure in Newton per square meters
```

Dimension	Name	Unit
length	meter	m
mass	kilogram	kg
time	second	s
electric current	ampere	A
thermodynamic temperature	kelvin	K
amount of matter	mole	mol
amount of light	candela	cd

Table 4.1: Fundamental SI units.

The units of measurement that can be used in the definition of unit types are the fundamental SI units (Table 4.1). These units are predefined in χ.

For example, to denote a concentration of some soluble component in a liquid, the following type can be defined:

```
type  conc  = [kg·kg^-1]      -- concentration in mass fractions
```

Although the unit of this type has no dimension, it is clear that the concentration is expressed in mass fractions (compare with for instance: $[\mathrm{m}^3\cdot\mathrm{m}^{-3}]$).

For values that cannot be expressed with the SI umits, for example the pH value of a liquid, we use the unit type $[-]$.

The keyword **unit** introduces a unit declaration by which new units can be defined. A unit is used as an abbreviation for a commonly used unit as the ones listed in Table 4.2. As an illustration of a declaration of derived units we define:

```
unit  N  = kg·m·s^-2,
      Ω  = m^2·kg·s^-3·A^-2
```

Dimension	Name	Symbol	Unit	SI units
frequency	hertz	Hz	s^{-1}	s^{-1}
force	newton	N	$kg \cdot m \cdot s^{-2}$	$m \cdot kg \cdot s^{-2}$
energy	joule	J	$N \cdot m$	$m^2 \cdot kg \cdot s^{-2}$
power	watt	W	$J \cdot s^{-1}$	$m^2 \cdot kg \cdot s^{-3}$
electrical charge	coulomb	C	$A \cdot s$	$s \cdot A$
electrical potential	volt	V	$W \cdot A^{-1}$	$m^2 \cdot kg \cdot s^{-3} \cdot A^{-1}$
electrical resistance	ohm	Ω	$V \cdot A^{-1}$	$m^2 \cdot kg \cdot s^{-3} \cdot A^{-2}$
electrical capacitance	farad	F	$C \cdot V^{-1}$	$m^{-2} \cdot kg^{-1} \cdot s^4 \cdot A^2$
pressure or stress	pascal	Pa	$N \cdot m^{-2}$	$m^{-1} \cdot kg \cdot s^{-2}$

Table 4.2: Some common derived SI units.

Constants

It is sometimes convenient to define an identifier to represent a predetermined value. Such constants are defined with a constant declaration. A constant declaration is introduced by the keyword const and is followed by an expression that defines the constant:

$$\mathsf{const}\ a_1 = e_1, \ldots, a_n = e_n$$

The identifier a_i, for $i \in \{1, \ldots, n\}$, is (globally) defined to have a value denoted by the expression e_i. The type of a constant is equal to the type of the expression that is used to define the constant.

We allow all kinds of operations in the expression of a constant, but we require that the expression can be evaluated statically. It may contain literal expressions and other constant identifiers.

As an example of a constant declaration, we define the acceleration of gravity g as:

$$\mathsf{const}\ g = 9.81\,[m \cdot s^{-2}]$$

where the unit of measurement $[m \cdot s^{-2}]$ is optional to allow consistency checks on expressions in equations. Note that the constant g is not a continuous variable.

Observe that the following constant declaration is invalid:

$$\begin{aligned}&\mathsf{const}\ a = b,\\ &\qquad b = 2 \times a\end{aligned}$$

because neither of the two constant expressions can be evaluated statically. Furthermore, a constant may only be defined once. So the following definition is invalid:

$$\begin{aligned}&\mathsf{const}\ b = a,\\ &\qquad b = 2 \times a\end{aligned}$$

4.5 Functions

Functions can be defined in two different ways. In a functional definition, we define *what* the result of a function is. An imperative definition states *how* the function result is obtained. The first has the advantage of resulting in compact function definitions without worrying about how the result must be calculated. The latter largely resembles the way in which processes are defined and can easier be implemented in a simulation tool. Furthermore, once familiar with processes, the imperative functions require less effort to learn. In the next sections, both types of function definitions are addressed, where we have a preference for the imperative function definition.

Imperative definition

By convention, the name of a function is printed in the Roman font style. The function definition starts with the declaration of the arguments and a range type of the function. For example, the function natlen, which determines the length of a list of Naturals, starts with the following definition:

$$\mathsf{func}\quad \mathrm{natlen}(xs : \mathrm{nat}^*) \to \mathrm{nat}$$

This function takes a list of Naturals as an argument and delivers a Natural as the result. The argument declarations are enclosed in parenthesis, separated by commas. The declaration of the function's arguments and result is called the type of a function. This type can be used as any other primitive type in χ. For example, we can put a function like natlen in a tuple with:

$$\langle(\mathrm{nat}^*) \to \mathrm{nat}\rangle$$

After the introduction of functions, at the end of this section, we will use this feature in an example.

A function call consists of the function name and the arguments, enclosed in parenthesis and separated by commas. For example:

$$\text{natlen}(xs)$$

calls the function natlen with the list xs as the argument. A function call is a valid expression in the formalism χ and can be used in all places where expressions can be used.

The declaration of local function variables is similar to the declaration of local process variables and is discussed in Section 4.4. The arguments of a function are all referenced by value. This means that no side-effects can occur when a function is called.

A function definition has the form:

$$\textsf{func}\ F(A) \rightarrow R = [\![\ V \mid F\ ]\!]$$

where A represents the declaration of arguments, and R defines the range of the function. Local variables are declared by V. The evaluation of a function is defined by statements, denoted by F. These statements are similar to the statements used in a process for the discrete behaviour. There are, however, some exceptions. For example, the event statements cannot be used in functions.

$$F \quad ::= \quad \varepsilon \mid x := e \mid F_1 ; F_2 \mid \uparrow e \mid X \mid *X$$

In functions, the only guarded command allowed is the selection statement and its repetitive variant. Furthermore, the statements after the guard of a selection statement may not contain event statements, only function statements.

$$X \quad ::= \quad [\,[\!]_{i=1}^{n}\ b_i \longrightarrow F_i\ ]$$

The result of a function is defined by the return statement ($\uparrow e$) and can be used more than once in a function definition. The return statement is always the last statement executed.

As an example, the function natlen can be defined by:

$$\begin{array}{l}\textsf{func}\ \text{natlen}(xs : \text{nat}^*) \rightarrow \text{nat} = \\ [\![\ [\ xs = [\,] \longrightarrow \uparrow 0 \\ \quad [\!]\ xs \neq [\,] \longrightarrow \uparrow 1 + \text{natlen}(\text{tl}(xs)) \\ \quad]\ \\ ]\!]\end{array}$$

The function may also be defined in an iterative way instead of a recursive way:

```
func natlen(xs : nat*) → nat =
[[ l : nat
| l := 0
; *[ xs ≠ [] ⟶ l := l + 1; xs := tl(xs) ]
; ↑l
]]
```

Note that the argument xs is used as the destination of an assignment. This is allowed because all arguments are referenced by value. The use of arguments in this way does not cause any side-effects in the process or function that calls this function. Thus, arguments may be considered as local variables in a function definition.

Functions are introduced for data manipulation purposes. Because of the strong typing of their arguments, they are closely related to the data objects they operate on. In the following example, this relation is expressed by including a function in the definition of a data type. We define a list in which the elements are sorted according to a given criterion. The criterion used here is rather simple: we sort the elements in ascending order. The elements themselves are of type real.

To preserve the order of elements in a sorted list, we define a function for adding elements to the list. This function puts an element at the proper position in the list, according to the given sort criterion:

```
func put(xs : real*, x : real) → real* =
[[ [ x ⩽ hd(xs) ⟶ ↑[x] ++ xs
   [] x > hd(xs) ⟶ ↑[hd(xs)] ++ put(tl(xs), x)
   ]
]]
```

The order of the elements is now determined by the function put. The sort criterion is a relation between the elements in a list. It defines when an element x precedes an element y. The following function tests two values on this relationship:

```
func ascend(x, y : real) → bool =
[[ ↑x ⩽ y ]]
```

This function returns true if the argument x should precede the argument y in our sorted list. This sort function and the list are so closely related, that we define the following data type for the ascending list:

```
type alist = ⟨real* × (real, real) → bool⟩
```

The left element of this tuple contains the list of elements. The right element contains the sort criterion in the form of a function.

We can now change the function put to use the sort function contained in the ascending list type alist:

```
func put(xs : alist, x : real) → alist =
⟦ [   xs.1(x, hd(xs.0)) ⟶ xs.0 := [x] ++ xs.0
  [] ¬ xs.1(x, hd(xs.0)) ⟶ xs.0 := [hd(xs.0)] ++ put(⟨tl(xs.0), xs.1⟩, x)
  ]
  ↑xs
⟧
```

Functional definition

We will now briefly introduce a functional definition of functions. For earlier mentioned reasons, we prefer the imperative function definition. Therefore, the following functional definition may not be complete and is merely included as an illustration. The presented definition of functions is derived from Gofer [Gof91].

The type definition is similar to imperative functions as is illustrated by the example function natlen:

```
func natlen : (nat*) → nat
```

where the type of the function is denoted by (nat*) → nat. The function application is the same:

```
natlen(xs)
```

The definition of the function result consists of one or more function definition expressions. A definition expression is a function pattern followed by a denotation of the function value for this pattern. For example, the function natlen can be defined with:

```
func  natlen        :  (nat*) → nat
      natlen([])    =  0
      natlen(x:xs)  =  1 + natlen(xs)
```

In the first line, the name and type of the function are defined. In the second line, the pattern 'natlen([])' denotes the application of the function natlen to an empty list, which results in the value 0. The last line in this function definition states that when the function is applied to a list, constructed from an element

x concatenated with a list xs, the result of the function is defined as 1 plus the length of xs.

The list construction operation (x:xs) provides a list that is guaranteed to contain at least one element (x) which is followed by the list of remaining elements (xs), which may be an empty list.

Within a function definition expression, it is possible to define a guarded function result. Consider the following example, where the function put is defined for a list of Reals:

$$
\begin{array}{lll}
\textbf{func}\ \text{put} & : & (\text{real}^*, \text{real}) \to \text{real}^* \\
\quad \text{put}([\,], e) & = & [e] \\
\quad \text{put}(x{:}xs, e) & = & [\ e \leqslant x \longrightarrow [e] \mathbin{+\!\!+} [x] \mathbin{+\!\!+} xs \\
 & & []\ e > x \longrightarrow [x] \mathbin{+\!\!+} \text{put}(xs, e) \\
 & &]
\end{array}
$$

In a function definition, a list may also be defined with qualifiers. For example:

$$[x^2 \mid x \leftarrow [2,4,6]]$$

defines the list $[4, 16, 36]$. Here the $\leftarrow$ means that the value of x is successively taken from the list $[2, 4, 6]$.

The qualified list can be used to define a list when the elements of the list are not known, but their characteristics can be denoted using qualifications. For example, the function put, operating on the type alist from the previous section, can be defined with:

$$
\begin{array}{lll}
\textbf{func}\ \text{put} & : & (\text{alist}, \text{real}) \to \text{alist} \\
\quad \text{put}(\langle xs, \text{f}\rangle, e) & = & \langle [x \mid x \leftarrow xs, \text{f}(x, e)] \mathbin{+\!\!+} [e] \mathbin{+\!\!+} [x \mid x \leftarrow xs, \text{f}(e, x)], \text{f}\rangle
\end{array}
$$

Comparing the examples of the functions, defined in an imperative or a functional way, we can see that a functional definition leads to a more compact definition. This is because the functional definition only states what the result of a function is and not how to obtain this result. This is one of the reasons that functional languages are more difficult to implement. Furthermore, the functional way of defining a function is quite different from the definition of processes. A systems engineer, wanting to use χ, must get familiar with the functional language as well. It is for these practicable reasons that we prefer the imperative function definitions.

4.6 Parameters

In some cases, processes have almost identical behaviours and differ only in small characteristics such as a buffer capacity or a vessel's volume. For these situations we introduce the concept of parameterisation. A process can be defined with a list of parameters, that specify its distinctive behavioural aspects. For example, we can use a parameter to pass a constant value to the process specification. Such values can be used to define characteristic properties of the process such as distinctive measurements or capacities. Also, a function can be passed as a parameter to alter the behaviour of the process. An example of such a parameter can be a function that represents the characteristic curves of a water turbine.

The formal parameter list in a process definition is enclosed in parenthesis and follows the process identifier:

$$\text{proc } P(\textit{formal parameters}) = [\![\ldots]\!]$$

When a process is used in a system, we create a process instantiation by filling in the actual parameters for the process:

$$\text{syst } Y = [\![\ldots \parallel P(\textit{actual parameters}) \parallel \ldots]\!]$$

By using parameterised process specifications, the channels, connected to a process, must be supplied as parameters as well. Recall, that the formalism requires that all channels are one-to-one connections and, therefore, two instantiations of the same process definition can not share any channels. The channels, connected to a process, are supplied by including them in the parameter list. Because of the different kinds of channels, we distinguish the following channel parameter types:

- Incoming discrete channel parameter.
- Outgoing discrete channel parameter.
- Synchronisation channel parameter.
- Incoming vector-type continuous channel parameter.
- Outgoing vector-type continuous channel parameter.
- Scalar-type continuous channel parameter.

Besides the kind of channel, also the type of a communicated object is included in the channel parameter declaration. For the above mentioned channel parameters, this results in declarations of the form:

- $c : ?\, type$
- $c : !\, type$
- $c : {}^{\sim}\, \text{void}$
- $c : \downarrow type$
- $c : \uparrow type$
- $c : \updownarrow type$

Observe that a synchronisation channel does not communicate a value. We introduce the type void to denote the 'empty type'.

The following example illustrates the use of parameters. The process B is a buffer that stores objects of type real. The maximum capacity is given by n and the function add determines how the objects are arranged in the buffer. The channels p and q respectively receive and send objects of type real.

$$
\begin{array}{l}
\textsf{proc}\ B(p : ?\,\text{real},\ q : !\,\text{real},\ n : nat,\ \text{add} : (\text{real}^*, \text{real}) \to \text{real}^*) = \\
[\![\ x : \text{real},\ xs : \text{real}^* \\
\mid *[\ \text{len}(xs) < n\ ;\ p?x \qquad \longrightarrow xs := \text{add}(xs, x) \\
\quad [\!]\ \text{len}(xs) > 0\ ;\ q!\,\text{hd}(xs) \longrightarrow xs := \text{tl}(xs) \\
\quad] \\
]\!]
\end{array}
$$

An example for the function parameter add can be defined as:

$$
\begin{array}{l}
\textsf{func}\ \text{fifo}(xs : \text{real}^*, x : \text{real}) \to \text{real}^* = \\
\qquad [\![\ \uparrow xs +\!\!+ [x]\ ]\!]
\end{array}
$$

which defines a FIFO ordering (First In, First Out). We can also use the function put, as described on page 50, to define a sorted order for the objects in the buffer.

Besides increasing the flexibility and modularity of process specifications, parameterisation is also introduced to enable hierarchical specifications, as will be discussed in the next section.

4.7 Hierarchy

As mentioned in Section 4.1, processes are grouped into a system and a system can act as a process in other system definitions. In this way, a hierarchical structure can be built. To achieve this hierarchy, a system definition is extended with a parameter list similar to that of process definitions. This allows a system to be used as a process in a system definition.

$$\textsf{syst}\ Y(\textit{formal parameters}) = [\![\ldots]\!]$$

Furthermore, the instantiation of a system is similar to the instantiation of a process:

$$\textsf{syst}\ S = [\![\ldots \parallel Y(\textit{actual parameters}) \parallel \ldots]\!]$$

The channels, connected to a system, are internally connected to the processes that comprise the system. Other channels, that exist between processes in a system, are declared in the system. We therefore extend a system definition with a declaration section:

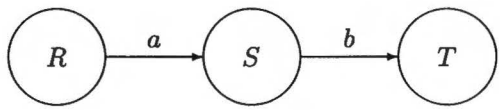
$$\textsf{syst}\ Y(\ldots) = [\![D \mid P_1(\ldots) \parallel \cdots \parallel P_n(\ldots)]\!]$$

where D denotes the declaration of channels. A channel declaration consists of an identifier for the channel followed by the type of the channel. The direction is not defined in the declaration, but is determined by the processes that are connected by the channel:

$$D ::= c_1 : \textit{type}_1,\ \ldots,\ c_n : \textit{type}_n$$

Note that we use the type void as the type of a synchronisation channel.

To illustrate the use of hierarchical structures, consider the following example:



$$\textsf{syst}\ F = [\![a, b : \text{real} \mid R(a) \parallel S(a,b) \parallel T(b)]\!]$$

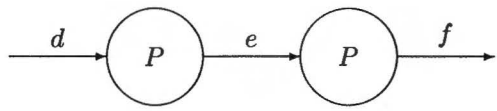
System F is defined to comprise of the three components R, S, and T. It is not clear from this definition whether the components of this system are processes or systems. This is intentionally. The components are connected by two channels a and b of type real, where channel a connects the components R and S and channel b connects S with T.

The components R and T are assumed to be processes, defined with:

$$\textsf{proc}\ R(x : !\,\text{real}) = [\![\ldots]\!]$$
$$\textsf{proc}\ T(y : ?\,\text{real}) = [\![\ldots]\!]$$

The channel a, connected to R, is in the definition of R denoted by the formal parameter x. It is declared as a discrete channel of type real, and is intended to be an outgoing communication channel. Similarly, channel parameter y denotes an incoming discrete channel of type real.

The component S is a system, composed of two (equal) components P:



syst $S(d : ?\,\text{real},\ f : !\,\text{real}) = [\![\ e : \text{real} \mid P(d,e) \parallel P(e,f)\ ]\!]$

proc $P(x : ?\,\text{real},\ y : !\,\text{real}) = [\![\ \ldots\ ]\!]$

Note that the identifiers in a system or process definition are all local identifiers. Note further that all hierarchical specifications can be rewritten as a single system, comprising of processes only. For example:

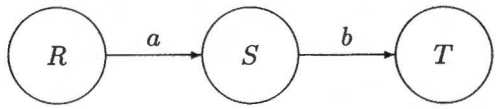
syst $\hat{F} = [\![\ c_1, c_2, c_3 : \text{real} \mid R(c_1) \parallel P(c_1,c_2) \parallel P(c_2,c_3) \parallel T(c_3)\ ]\!]$

specifies exactly the same behaviour as system F, where the components R, P, and T are defined as described earlier. Comparing the two specifications, we can see that channel c_1 in $\hat{F}$ is equivalent to the concatenation of channel a of F and channel d of S in the hierarchical specification.

Next to the hierarchical ordering of processes and systems, we introduce bundles to create a hierarchy in channels. A bundle is declared similar to a tuple, except that parenthesis are used to delimit a bundle.

(real^2)

represents a bundle, consisting of two channels. All channels in a bundle are either continuous or discrete channels and have the same type. Furthermore, the directions of all channels in a bundle must be the same. The graphical representation of a bundle is equal to that of a channel. As an example of the use of bundles, we replace the channels in the previous example by bundles of the type (real^2):



syst $F = [\![\ a, b : (\text{real}^2) \mid R(a) \parallel S(a,b) \parallel T(b)\ ]\!]$

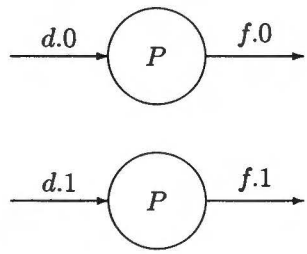
The processes R and T are defined with:

$$\mathsf{proc}\ R(x : !(\mathrm{real}^2)) = [\![\ldots]\!]$$
$$\mathsf{proc}\ T(y : ?(\mathrm{real}^2)) = [\![\ldots]\!]$$

Within a process, a channel, contained in a bundle, can be addressed using its index in the bundle. For example, to receive from the first channel in the bundle, we can use:

$$\mathsf{proc}\ T(x : ?(\mathrm{real}^2)) = [\![\ldots ; x.0\,?\,a ; \ldots]\!]$$

where a is a variable of type real. Indexing can also be applied to bundles in a system:



$$\mathsf{syst}\ S(d : ?(\mathrm{real}^2),\ f : !(\mathrm{real}^2)) = [\![P(d.0, f.0) \parallel P(d.1, f.1)]\!]$$

$$\mathsf{proc}\ P(x : ?\,\mathrm{real},\ y : !\,\mathrm{real}) = [\![\ldots]\!]$$

Observe that the type (real^2) is an abbreviation for (real $\times$ real). To create a bundle with synchronisation channels, we can use:

$$(\mathrm{void}^3)$$

Bundles can be grouped into a new bundle, thus creating a hierarchical structure of channels. For example, the following bundle can be created:

$$((\mathrm{nat}^5)^2)$$

For these hierarchical bundles, we also demand that the types and the directions, if relevant, of all elements in a bundle are the same.

4.8 Stochastic behaviour

Difficulties in the analysis of a system's behaviour often arise from the fact that the behaviour of a system is not deterministic but stochastic of nature. To enable the investigation of such systems, we introduce statistical distributions and a sampling operator on them to specify stochastic behaviour.

A statistical distribution is a function that defines the probability for a certain outcome when drawing a sample from the distribution's domain. We distinguish two kinds of domains for distributions: a discrete domain and a continuous domain. A discrete domain contains only a fixed number of values, where a continuous domain defines an infinite number of values.

A number of distributions are defined in χ that are commonly used in system specifications (see also Appendix B):

- Discrete distributions:
  - dun — Discrete Uniform
  - ber — Bernoulli
  - bin — Binomial
  - geo — Geometric
  - poi — Poisson
- Continuous distributions:
  - cun — Continuous Uniform
  - nex — Negative Exponential
  - erl — Erlang
  - nor — Normal
  - gam — Gamma
  - wei — Weibull

Drawing a sample from a distribution can not be defined by a function because the result of such a function would not be the same for every call. Therefore, we define a sample operator σ to draw a sample from a distribution. Furthermore, since a distribution is not like any type of data we introduced so far, we define the type dist to denote statistical distributions. In a declaration of a distribution, we also indicate the type of the samples, taken from the distribution. For example, we use dist(nat) to define a distribution that generates samples of type nat. Thus, we have the types dist(nat) and dist(int) for discrete distributions and dist(real) for continuous distributions.

For example, to define a variable, representing a Binomial distribution, we use:

d : dist(nat)
$d := \text{bin}(p, n)$

where d is declared as a distribution variable and $\text{bin}(p, n)$ defines a Binomial distribution with probability p and number of trials n. Note that bin is a function of type $(\text{real}, \text{nat}) \rightarrow \text{dist(nat)}$. A sample from this distribution is of type nat.

The expression:

$\sigma\, d$

denotes drawing a sample from the distribution d. Observe that the expression $\sigma\, d - \sigma\, d$ does not necessarily evaluate to the value 0.

4.9 Summary

In this chapter, we presented the formalism χ, starting with a minimal language, only containing the primitive language constructs of χ. Systems and processes are the main building blocks of the formalism. The state of a system is recorded in the variables, which are declared as local variables in the processes. We introduced two behaviour definitions, one for discrete behaviour and one for continuous behaviour. The two kinds of behaviours can be combined in a single process, thus specifying a hybrid behaviour.

Some extensions to the minimal language are proposed to make the language more applicable to the specification of industrial systems. Data types, such as lists and tuples, enable the definition of hierarchical data structures. Continuous variables are declared with the definition of their units of measurement. Some units are predefined in χ, others can be created with unit declarations. Furthermore, constants can be declared for those values that are global for a system specification, such as the acceleration of gravity.

We presented two kinds of function definitions. Imperative functions, that use similar statements as the statements used in processes. Functional definitions tend to result in more compact function descriptions that are, however, not as easily mastered.

We introduced parameters as a means to make specifications more flexible and to allow the reuse of existing specifications. Parameters also introduced the definition of hierarchical models. Processes can be grouped into systems and a system can be grouped with other processes and systems into a new system. Next to the hierarchy of systems and processes, we defined bundles to create hierarchical structured channels.

The definition of the formalism χ is completed with the introduction of stochastic behaviour. Discrete and continuous statistical distributions are defined as well as a special operator for taking samples from these distributions.

Chapter 5

The Bottle-filling System

To illustrate the use of χ for the specification of industrial systems, we will set up a specification of the bottle-filling system as described earlier in Section 3.1.

The first step in the modelling process is the system identification. We isolate the system from its environment and determine the interactions of the system with the environment. The objects that are involved in the interactions are distinguished and suitable data structures are defined to represent these objects. Also, the functions for the manipulation of the data structures are defined.

After the system has been identified, its structure is defined. The system is decomposed into subsystems and processes that have relations with each other. These relations are modelled as data objects, moving through the system. The objects are represented by data structures and these data structures are used in the definition of the relations between the model components. Finally, we discuss the behaviour descriptions of the processes in the model.

The objectives for the model are to find out how many customer orders can be handled by the system. In this analysis, we assume that the system operates ideally (i.e. bottles do not break and machines do no fail). Experiments can be performed to investigate the influence of changes in the customer orders, for instance, "What happens if the average amount of bottles ordered will increase?".

5.1 The system environment

The bottle-filling system has interactions with the customers and the suppliers of bottles and liquid. In our model, we assume that there is always a sufficient amount of liquid available in a reservoir to refill the vessel. Because of this assumption, there is no need to include the (external) supplier of liquid to this

reservoir in the model. Also, the supplier of bottles is treated in the same way and is therefore not included in the system's environment. The environment of the system consists therefore of the customers only (Figure 5.1). The customers

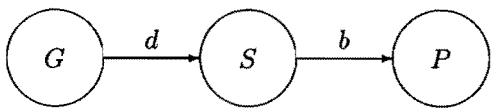


Figure 5.1: The bottle-filling system environment.

are modelled by two processes G and P. The first process generates the orders of the customers. These orders are sent along channel d to the bottle-filling system S. The second process P is a pile that collects the deliveries of the bottles, sent along channel b.

The interactions with the system involve the flow of orders and the flow of filled bottles from and to the customers. These objects travel along the channels d and b respectively. We will now define data structures to represent these objects.

An order is a request for a certain amount of bottles. The system delivers bottles of two sizes (1 litre and 5 litre bottles). Therefore, the order contains two amounts, one for each size. Thus, we define the following type to represent an order:

$$\textsf{type}\quad \text{ord} \qquad = \langle \text{nat}^2 \rangle$$

The filled bottles are delivered to the customer as a single package, containing both sizes of ordered bottles. Before we define a data structure for this package, we first define a data structure for a single bottle. Since there are two sizes of bottles, the size must be indicated in the data structure. Furthermore, each bottle has a label, stating its contents. Thus, a type to represent a bottle may be defined with:

$$\textsf{type}\quad \text{bot} \qquad = \langle \text{char} \times \text{dat} \rangle$$

in which the first element (of type char) denotes the size of a bottle (x for 1 litre bottles and y for 5 litre bottles). The type dat represents the label attached to the bottle. This label states the actual pH value and amount of liquid that is contained in the bottle. This label is defined with:

$$\textsf{type}\quad \text{dat} \qquad = \langle \text{real}^2 \rangle$$

A delivery of bottles to a customer is a collection of (small and large) bottles. This collection is modelled by a list of bottles, denoted by:

$$\text{bot}^*$$

Order generator

The first process that models the environment is the process that generates the orders for the system. An order defines in what amounts the two bottle types are required. Both these values, that comprise an order, are represented by statistical distributions. The kind of distribution is defined in the instantiation of the process. The type of outcome for a sample from these distributions is defined in the process definition. In this case, the amounts are restricted to Naturals.

New orders are created with a call to the function neword, where the arguments denote the requested amounts of bottles. The function is defined as:

$$\mathsf{func}\quad \text{neword}(x, y : \text{nat}) \rightarrow \text{ord} = [\![\uparrow(x, y)]\!]$$

Although this seems to make little sense, it has some advantages over using the literal expression for a tuple directly in the process specification. It increases the readability of the specification. Furthermore, future changes to the type ord will affect the data manipulation functions only and not the specifications in which the functions are used. Therefore, throughout this chapter, we define all data manipulation with functions.

The generation of orders is defined by the process G:

$$\begin{array}{l}\mathsf{proc}\ G(d : !\,\text{ord},\ d_x, d_y : \text{dist(nat)}) = \\ [\![*[\ d\,!\,\text{neword}(\sigma\, d_x, \sigma\, d_y)\ ]\ ]\!]\end{array}$$

In the first line of this specification, the channel d is declared to transfer objects of type ord. Also, the distributions, used to generate the various amounts of bottles, are declared as parameters. Here, only the type of outcome for the sample operation on a distribution is defined. The declaration dist(nat) declares a distribution of Naturals. The actual distribution used by the process will be defined with the instantiation of the process later on.

The behaviour of the process is described by an endless loop in which the process tries to send a new order along channel d. The arguments of the function call neword are samples from the two distributions.

Note that the generator is always prepared to send an order. However, an order is only sent if the system is willing to accept the order. Meanwhile, the generator will be suspended. This mechanism implies that the bottling system operates at its maximum capacity.

Bottle pile

The filled bottles, requested by an order, are delivered to a pile. The only task of the pile process is to collect the deliveries.

$$\text{proc } P(b : ?\,\text{bot}^*) = [\![\, xs : \text{bot}^* \mid *[\, b?\,xs \,]\,]\!]$$

Observe that the received deliveries are ignored by the process. We are not interested in the filled bottles, only in the fact that they are delivered. If we were to find out how long it would take to handle an order, we could note the time of delivery and compare it with the time at which the corresponding order was sent.

The specification of the system, depicted in Figure 5.1, is formally defined with:

$$\begin{array}{l}\text{syst } M = \\ [\![\, d : \text{ord},\ b : \text{bot}^* \\ \mid G(d, \text{dun}(40,80), \text{dun}(10,30)) \parallel S(d,b) \parallel P(b) \\ ]\!]\end{array}$$

First the two channels d and b are declared with the appropriate types. The instantiation of process G defines two discrete uniform distributions, denoted by $\text{dun}(min, max)$ for the amounts of the two sizes of bottles. In the next section, we develop a specification of the system S including its control system.

5.2 The system

The bottle-filling system consists of a vessel, two filling lines and a packaging machine in which the filled bottles are wrapped. These subsystems are controlled by a system controller that handles the customer orders. The structure of the bottle-filling system is depicted in Figure 5.2 where VS represents the vessel system, FS is a filling line, and W is the packaging machine. System controller SC is placed in the centre of the picture.

The specification of this system can be given by:

$$\begin{array}{l}\text{syst } S(d_s : \text{ord},\ b_f : \text{bot}^*) = \\ [\![\, f_x, f_y : \text{bot},\ n_x, n_y : \text{nat},\ r_x, r_y : \text{void}, \\ \quad p_x, p_y : \text{real},\ l_x, l_y : \text{liq},\ d_w : \text{ord} \\ \mid VS(p_x, p_y, l_x, l_y) \\ \parallel FS(n_x, p_x, r_x, f_x, l_x, \text{x}, 1.0) \\ \parallel SC(d_s, d_w, n_x, n_y, r_x, r_y) \\ \parallel FS(n_y, p_y, r_y, f_y, l_y, \text{y}, 5.0) \\ \parallel W(d_w, f_x, f_y, b_f) \\ ]\!]\end{array}$$

The two channels, defining the interactions with the system's environment, are declared as parameters of the system. Other (internal) channels are declared

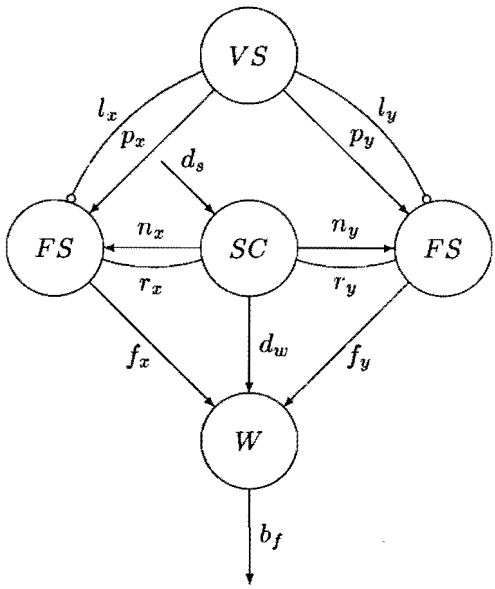


Figure 5.2: Bottle-filling system.

within the system and define the relations between the various subsystems. Finally, the subsystems are instantiated with the proper parameter values and channels. Note that the two filling lines share the same specification FS. The differences are expressed by using different values for the parameters that indicate the kind of bottles to be filled. The small bottles are denoted by the character x and have a size 1.0, whereas the large bottles have a size 5.0 and are indicated by the character y. Furthermore, the two filling lines have different relations with their environment and thus are connected by different channels.

The meaning of the channels in this system is made clear in the remaining of this chapter in which the subsystems are discussed in more detail. We start with the specification of the system controller. Its behaviour plays a central role in the behaviour of the system. By specifying the control system first, we gain a clear overview of the cooperation of the various subsystems and their relations.

System Controller

The system controller handles the incoming orders one by one. An order is received from a customer via channel d_s. This order is passed to the packaging machine so that the filled bottles can be collected and wrapped according to this order. The two filling lines are informed about the amounts of bottles that

have to be filled. These values are sent via the channels n_x for small bottles and n_y for large bottles. Finally, a signal must be received from these filling lines when the bottles are filled. Only then, the next order can be processed. The specification of the system controller can be given by:

$$
\begin{array}{l}
\mathsf{proc}\ SC(d_s : ?\,\mathrm{ord},\ d_w : !\,\mathrm{ord},\ n_x, n_y : !\,\mathrm{nat},\ r_x, r_y : {}^{\sim}\,\mathrm{void}) = \\
[\![\ d : \mathrm{ord} \\
|\ *[\ d_s ?\, d \\
\quad ;\ d_w !\, d \\
\quad ;\ n_x !\, \mathrm{nx}(d) \\
\quad ;\ n_y !\, \mathrm{ny}(d) \\
\quad ;\ [\ r_x{}^{\sim} \longrightarrow r_y{}^{\sim}\ [\![\ r_y{}^{\sim} \longrightarrow r_x{}^{\sim}\ ] \\
\quad] \\
]\!]
\end{array}
$$

We use the functions nx and ny to extract the two amounts from an order. These functions are defined with:

$$\mathsf{func}\quad \mathrm{nx}(d : \mathrm{ord}) \to \mathrm{nat} = [\![\ \uparrow \mathrm{lft}(d)\ ]\!]$$

$$\mathsf{func}\quad \mathrm{ny}(d : \mathrm{ord}) \to \mathrm{nat} = [\![\ \uparrow \mathrm{rgt}(d)\ ]\!]$$

The order in which the 'ready' signals from the filling lines are received is not known in advance. Therefore, we use a guarded command to handle both possibilities: first r_x, then r_y, or first r_y and then r_x.

The order in which the subsystems are addressed next is inspired by the flow of material through the system. This flow starts at the vessel system that supplies the liquid. Next, the filling lines are presented and, finally, the packaging machine or bottle wrapper is specified.

5.3 The vessel system

The main function of the vessel system (Figure 5.3) is to supply liquid with a predetermined pH value to both filling lines. The vessel is fed from an inexhaustible source of liquid through an input valve that is controlled by a level controller. This controller's task is to keep the level in the vessel between a minimum and a maximum value.

The acidity of the liquid changes slightly because of exposure to the air. The pH value can be adjusted by the addition of an acid to the liquid in the vessel. A pH-controller measures the current pH value and controls a valve through which the acid can be supplied. We assume, as with the liquid, that there is an inexhaustible source of acid available.

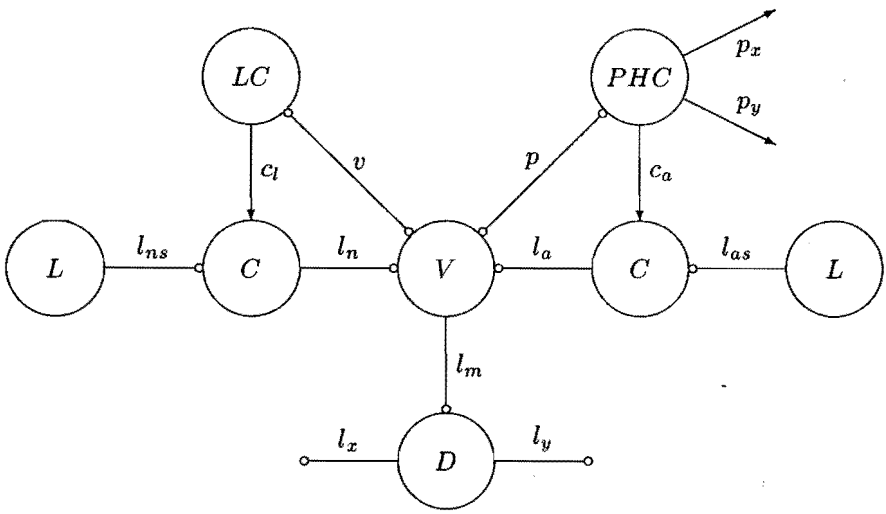


Figure 5.3: The vessel system.

From the vessel, the liquid is led to the filling lines. The flow is split up into two sub-flows, one for each filling line. A flow distributor models the behaviour of the liquid flow at this fork. The specification of the vessel system, as shown in Figure 5.3, is given by:

$$
\begin{array}{l}
\textsf{syst}\ VS(p_x, p_y : !\,\text{real},\ l_x, l_y : \uparrow \text{liq}) = \\
[\![\ l_{ns}, l_{as}, l_n, l_a, l_m : \text{liq},\ c_l, c_a : \text{cmd},\ v : \text{vol},\ p : \text{pH} \\
\mid\ LC(v, c_l, 0.05, 0.75) \\
\|\ L(l_{ns}, 2.5{\cdot}10^5, 0.0017) \\
\|\ C(l_{ns}, l_n, c_l, \alpha) \\
\|\ V(l_n, l_a, l_m, v, p, 0.785) \\
\|\ C(l_{as}, l_a, c_a, \alpha) \\
\|\ L(l_{as}, 1.5{\cdot}10^5, 0.0018) \\
\|\ PHC(p_x, p_y, p, c_a, 3.8, 0.1) \\
]\!]
\end{array}
$$

A number of data types have been used in the declaration of channels and parameters. The type liq denotes a liquid and is used for both the blend and the acid flows in the system. It is defined as:

$$\textsf{type}\quad \text{liq} \qquad = (\text{flow} \times \text{press} \times \text{conc})$$

A liquid is represented by a triple containing a flow, a pressure, and a concentration. The types flow and press are defined by:

$$\begin{array}{lll} \textsf{type} & \text{flow} & = [\mathrm{m}^3 \cdot \mathrm{s}^{-1}] \\ , & \text{press} & = [\mathrm{N} \cdot \mathrm{m}^{-2}] \end{array}$$

where the unit N stands for Newtons and is defined as:

$$\textsf{unit} \quad \mathrm{N} \qquad = \mathrm{kg} \cdot \mathrm{m} \cdot \mathrm{s}^{-2}$$

The concentration denotes the amount of pure acid in a blend and is defined with:

$$\textsf{type} \quad \text{conc} \qquad = [\mathrm{m}^3 \cdot \mathrm{m}^{-3}]$$

For the amount of liquid, contained in the vessel or a bottle, we introduce the type vol, defined as:

$$\textsf{type} \quad \text{vol} \qquad = [\mathrm{m}^3]$$

A pH value cannot be expressed in SI units. To represent a pH value with a continuous variable, we define the type pH as:

$$\textsf{type} \quad \text{pH} \qquad = [-]$$

Control valves

A control valve can be seen as a resistor in a pipe and thus causes a pressure loss Δp between the inlet and outlet of the valve. The resistance is denoted by the resistor coefficient ζ. Both the resistance and the speed v of the liquid flowing through the valve apply to the aperture A of the valve. The volume flow through the valve can be defined with $\varphi = v \times A$. The pressure loss is defined by [Dub]:

$$\Delta p = \frac{\zeta \times \rho \times v^2}{2}$$

Thus, the relation between the volume flow through the valve and the pressure loss over the valve is defined by:

$$\varphi^2 = A^2 \frac{2 \times \Delta p}{\rho \times \zeta}$$

By changing the aperture A of the valve, we can control the volume flow through the valve. We introduce a rate r by which we can control the aperture of the valve and define the value α as:

$$\alpha = \frac{2 \times A^2}{\rho \times \zeta}$$

so that the equation to describe the control valve becomes:

$$\varphi^2 = r \times \alpha \times \Delta p$$

We can now define the process C that specifies a control valve by:

```
proc C(l1 : ↓liq, l2 : ↑liq, c : ?cmd, α : real) =
[[ φ : flow, p1,p2 : press, x : conc, s : cmd, r : real
| l1 —o (φ,p1,x), l2 —o (φ,p2,x)
| φ² = r × α × (p1 − p2)
, [ s = open   ⟶ r = 1.0
  [] s = dribble ⟶ r = 0.1
  [] s = close  ⟶ r = 0.0
  ]
| s := close; *[ c?s ]
]]
```

Note that this specification is only valid if $p_1 \geqslant p_2$ at all times. If this restriction cannot be guaranteed, we can adjust the specification by replacing the first equation with the following equation:

$$\varphi = r \times \sqrt{\alpha \times \mathrm{abs}(p_1 - p_2)} \times \mathrm{sgn}(p_1 - p_2)$$

where the functions abs and sgn are defined as:

```
func abs(x : real) → real =
[[ [ x < 0 ⟶ ↑−x
  [] x ⩾ 0 ⟶ ↑  x
  ]
]]

func sgn(x : real) → int =
[[ [ x < 0 ⟶ ↑−1
  [] x = 0 ⟶ ↑  0
  [] x > 0 ⟶ ↑+1
  ]
]]
```

We assume that all valves and pipes, used in the bottle-filling system, have similar characteristics. If we assume certain values for A, ρ, and ζ, the value of α can be computed as:

$$\alpha = \frac{2 \times A^2}{\rho \times \zeta} = \frac{2 \times (1 \cdot 10^{-3})^2}{1000 \times 0.25} = 8 \cdot 10^{-9}$$

Since this value will be used in several specifications, we declare it as a constant with:

$$\textsf{const } \alpha \qquad = 8 \cdot 10^{-9}$$

Liquid and acid supply

In the specification of the vessel system, the process L models an inexhaustible source of liquid. By applying the proper parameter values, it can be used for both the liquid and the acid suppliers for the vessel. We define a pressure (with values $2.5 \cdot 10^5[\mathrm{N} \cdot \mathrm{m}^{-2}]$ and $1.5 \cdot 10^5[\mathrm{N} \cdot \mathrm{m}^{-2}]$ for liquid and acid supply resp.) as a driving force for the liquid flow. The last parameter indicates the concentration of pure acid in the blend: a value of $0.0017[\mathrm{m}^3 \cdot \mathrm{m}^{-3}]$ for a liquid with the requested pH value of 3.8 and a value of $0.0018[\mathrm{m}^3 \cdot \mathrm{m}^{-3}]$ for the acid supply which corresponds to a pH value of 2.2.

The process definition of L can be given by:

$$\begin{array}{l}
\textsf{proc } L(l : \uparrow \mathrm{liq},\ p_s, s : \mathrm{real}) = \\
[\![\ \varphi : \mathrm{flow},\ p : \mathrm{press},\ x : \mathrm{conc} \\
\mid l \multimap \langle \varphi, p, x \rangle \\
\mid \varphi^2 = \alpha \times (p_s - p),\ x = s \\
]\!]
\end{array}$$

Here, p_s is the pressure at which the liquid supply takes place. The acidity of the liquid is denoted by the concentration of acid in the liquid given by the parameter s. We use the constant α to describe the flow characteristics of the outlet of the supplier.

Vessel

The vessel has two inlets and one outlet. Furthermore, it contains two sensors that are used to pass information to the level and pH controllers. The specification of the vessel reads as follows:

$$\begin{array}{l}
\textsf{proc } V(l_n, l_a : \downarrow \mathrm{liq},\ l_m : \uparrow \mathrm{liq},\ v : \updownarrow \mathrm{vol},\ p : \updownarrow \mathrm{pH},\ A : \mathrm{real}) = \\
[\![\ \varphi_n, \varphi_a, \varphi_m : \mathrm{flow},\ p_n, p_a, p_m : \mathrm{press},\ x_n, x_a, x_m : \mathrm{conc}, \\
\quad V : \mathrm{vol},\ pH_c : \mathrm{pH} \\
\mid l_n \multimap \langle \varphi_n, p_n, x_n \rangle,\ l_a \multimap \langle \varphi_a, p_a, x_a \rangle,\ l_m \multimap \langle \varphi_m, p_m, x_m \rangle, \\
\quad v \multimap V,\ p \multimap pH_c
\end{array}$$

$$
\begin{array}{l}
\| \; V' = \varphi_n + \varphi_a - \varphi_m \\
, \; x'_m \times V = (x_n - x_m) \times \varphi_n + (x_a - x_m) \times \varphi_a - 0.35{\cdot}10^{-6} \\
, \; p_n = p_{atm} \\
, \; p_a = p_{atm} \\
, \; p_m = p_{atm} + \rho \times g \times V/A \\
, \; [\; x_m < 0.00125 \longrightarrow pH_c = -400 \times x_m + 11.5 \\
\quad [\!] \; x_m \geqslant 0.00125 \longrightarrow pH_c = -16000 \times x_m + 31 \\
\quad] \\
\|
\end{array}
$$

The vessel has a continuous behaviour, fully described by equations. In the first equation, the volume balance is defined with the sum of incoming and outgoing flows. The pressure at the top of the vessel is the atmospheric pressure, defined with the constant:

$$\mathsf{const}\ p_{atm} = 1.01{\cdot}10^5$$

At the bottom (at the outlet), the pressure is increased by the term $\rho \times g \times v/A$, where A is the cross-section of the vessel. The concentration of acid in the vessel is denoted by x_m which is of the type $[\mathrm{m}^3{\cdot}\mathrm{m}^{-3}]$. This type defines that the concentration is expressed in volume fractions. The constant $0.35{\cdot}10^{-6}$ denotes the change in the concentration due to atmospheric influences. The calculation of the pH value depends on the value of x_m as defined by the guarded equation.

Level Controller

The amount of liquid in the vessel is kept between a minimum value v_n and a maximum value v_x.

$$
\begin{array}{l}
\mathsf{proc}\ LC(v : \updownarrow \mathrm{vol},\ c_l : !\,\mathrm{cmd},\ v_n, v_x : \mathrm{real}) = \\
[\![\; v_c : \mathrm{vol} \\
| \; v \multimap v_c \\
| \; *[\; v_c \geqslant v_n \,;\, \nabla\, v_c < v_n \longrightarrow c_l\,!\,\mathsf{open} \\
\quad [\!] \; v_c \leqslant v_x \,;\, \nabla\, v_c > v_x \longrightarrow c_l\,!\,\mathsf{close} \\
\quad] \\
]\!]
\end{array}
$$

The current volume is available as the value of v_c. The controller's behaviour can be described as follows. If the current volume is greater or equal to the minimum volume then we wait for the current volume to drop below this minimum and we open the valve. If the current volume is less or equal to the maximum volume then we wait for the vessel to be filled to its maximum volume and we close the valve.

pH-Controller

The behaviour of the pH-controller is a little more complicated compared to that of the level controller.

$$
\begin{array}{l}
\mathsf{proc}\ PHC(p_x, p_y : !\,\mathrm{real},\ p : \updownarrow \mathrm{pH},\ c_a : !\,\mathrm{cmd},\ pH_r, tol : \mathrm{real}) = \\
[\![\ pH_c : \mathrm{pH},\ d, ok : \mathrm{bool} \\
\mid\ p \multimap pH_c \\
\mid\ ok := (\mathrm{abs}(pH_c - pH_r) \leqslant tol) \\
;\ *[\ ok;\ p_x\,!\,pH_c \\
\quad [\![\ ok;\ p_y\,!\,pH_c \\
\quad [\![\ \nabla(\mathrm{abs}(pH_c - pH_r) > tol) \\
\qquad \longrightarrow ok := \mathrm{false} \\
\qquad\ ;\ d := \mathrm{true};\ c_a\,!\,\mathsf{dribble} \\
\qquad\ ;\ *[\ \neg ok \wedge d;\ \nabla pH_c > 11.0 \qquad\qquad \longrightarrow d := \mathrm{false};\ c_a\,!\,\mathsf{open} \\
\qquad\qquad [\![\ \neg ok \wedge \neg d;\ \nabla pH_c < 11.0 \qquad\ \longrightarrow d := \mathrm{true};\ c_a\,!\,\mathsf{dribble} \\
\qquad\qquad [\![\ \neg ok;\ \nabla(\mathrm{abs}(pH_c - pH_r) \leqslant tol) \quad \longrightarrow ok := \mathrm{true};\ c_a\,!\,\mathsf{close} \\
\qquad\qquad] \\
\quad] \\
]\!]
\end{array}
$$

The pH-controller keeps the current pH (pH_c) of the liquid in the vessel close to the set-point pH_r. As long as the current pH is within range, this pH value can be sent along the channels p_x and p_y to inform the filling lines that the pH is OK and that bottles may be filled. If the pH is out of tolerance, a correction procedure starts. This procedure changes the pH of the liquid in the vessel by dribbling acid into it. If the pH crosses the critical value of 11, the acid valve is completely opened. The procedure stops when pH_c is within tolerance boundaries again. Note that the interactions $p_x\,!\,pH_c$ and $p_y\,!\,pH_c$ cannot occur during the correction procedure.

Flow distributor

The flow distributor is a T-fork that splits the flow coming from the vessel into two sub-flows, one for each filling line. All characteristics of the liquid are the same for all flows at this process, i.e. the pressure and the concentration of acid. The only behaviour described by this process is based on the mass balance equations:

$$
\begin{array}{l}
\mathsf{proc}\ D(l_m : \downarrow \mathrm{liq},\ l_x, l_y : \uparrow \mathrm{liq}) = \\
[\![\ \varphi_x, \varphi_y, \varphi_m : \mathrm{flow},\ p : \mathrm{press},\ x : \mathrm{conc} \\
\mid\ l_x \multimap \langle \varphi_x, p, x \rangle,\ l_y \multimap \langle \varphi_y, p, x \rangle,\ l_m \multimap \langle \varphi_m, p, x \rangle \\
\mid\ \varphi_x + \varphi_y = \varphi_m \\
]\!]
\end{array}
$$

This specification of the flow distributor concludes the definition of the vessel system. The next section discusses the specification of the filling lines.

5.4 The filling system

Each filling line consists of a bottle supply and release unit and a filling station. A controller coordinates the various activities of these subsystems. Figure 5.4 shows the structure of a filling line, where B is the supplier of empty bottles, T is the bottle transport and release process, F is the filling station, C is a valve, and FC is the filling controller.

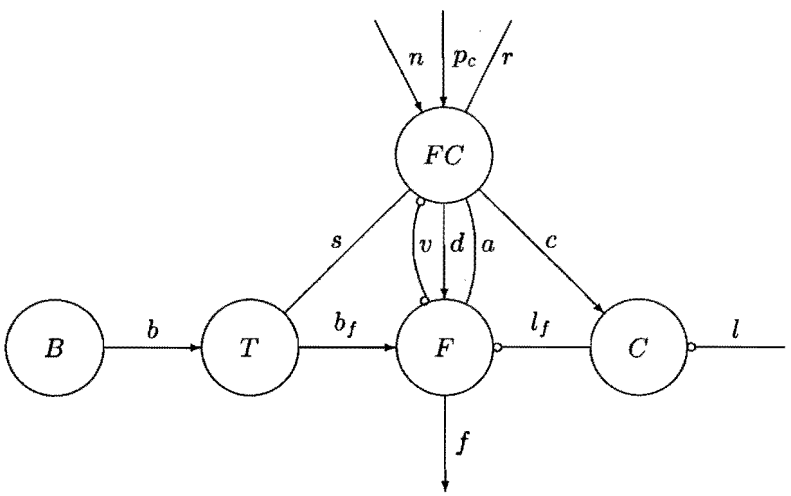


Figure 5.4: The filling system.

The specification of the filling line defines the subsystems and processes and their relations:

```
syst FS(n : ?nat, p_c : ?real, r : ~void, f : !bot,
        l : ↓liq, t : char, v_f : real) =
[[ b, b_f : bot, s, a : void, v : vol, d : dat, c : cmd, l_f : liq
| FC(n, r, s, a, p_c, v, c, d, v_f)
|| B(b, t)
|| T(b, b_f, s, 10)
|| F(b_f, f, d, l_f, v, a)
|| C(l, l_f, c, α)
]]
```

We will now have a closer look at the different processes that comprise the filling line.

Bottle supply

The bottle supply unit models an inexhaustible source of empty bottles. Each filling line has its own supply unit that supplies bottles of the appropriate size, indicated by the parameter z:

$$\text{proc } B(b : !\,\text{bot},\ z : \text{char}) = [\![\ *[\ b\,!\,\text{newbot}(z)\ ]\ ]\!]$$

New empty bottles are created by the function newbot that attaches the size indication z to each bottle. The initial data is set for each new bottle by the function newdat that is called from newbot.

$$\text{func}\quad \text{newbot}(c : \text{char}) \to \text{bot} = [\![\ \uparrow\langle c, \text{newdat}(0.0, 0.0)\rangle\ ]\!]$$

$$\text{func}\quad \text{newdat}(p, v : \text{real}) \to \text{dat} = [\![\ \uparrow\langle p, v\rangle\ ]\!]$$

Bottle release process

$$\begin{array}{l}
\text{proc } T(b : ?\,\text{bot},\ b_f : !\,\text{bot},\ s : {}^{\sim}\,\text{void},\ n : \text{nat}) = \\
[\![\ xs : \text{bot}^*,\ x : \text{bot} \\
\mid xs := [] \\
;\ *[\ \text{len}(xs) < n\ ;\ b\,?\,x \longrightarrow xs := xs +\!\!+ [x] \\
\quad [\!]\ \text{len}(xs) > 0\ ;\ s^{\sim} \longrightarrow \Delta\,1.5;\ b_f\,!\,\text{hd}(xs);\ xs := \text{tl}(xs) \\
\quad] \\
]\!]
\end{array}$$

The bottle release process is a conveyor belt with a limited capacity of n bottles. As long as there is free space on the belt, a new bottle can be received on channel b, which is then put in the list xs. Bottles are, after a request on channel s, removed from the belt and thus removed from the list xs. After the bottles are removed from the conveyor belt, they slide down a chute to the position under the filling process. This takes 1.5 time units. The conveyor belt behaves as a limited capacity FIFO buffer.

Filling process

The filling process models the actual filling of the bottles:

$$
\begin{array}{l}
\textsf{proc}\ F(b : ?\,\mathrm{bot},\ q : !\,\mathrm{bot},\ d : ?\,\mathrm{dat},\ l : \downarrow \mathrm{liq},\ v : \updownarrow \mathrm{vol},\ a : {}^{\sim}\,\mathrm{void}) = \\
[\![\ \varphi : \mathrm{flow},\ p : \mathrm{press},\ x : \mathrm{conc},\ v_b : \mathrm{vol},\ f : \mathrm{bot},\ lbl : \mathrm{dat} \\
\mid l \multimap \langle \varphi, p, x \rangle,\ v \multimap v_b \\
\mid v_b' = \varphi,\ p = p_{atm} \\
\mid *[\ b?f;\ v_b := 0.0;\ a^{\sim};\ d?lbl;\ f := \mathrm{addlbl}(f, lbl);\ q!f\ ] \\
]\!]
\end{array}
$$

A new empty bottle f is received through channel b. The bottle's contents is set to empty ($v_b := 0.0$) and a signal is sent on channel a to indicate the arrival of the new bottle. After the bottle has been filled, a label with the corresponding data is received on channel d that is attached to the bottle with the function addlbl. This function is defined as:

$$
\textsf{func}\quad \mathrm{addlbl}(b : \mathrm{bot}, d : \mathrm{dat}) \to \mathrm{bot} = [\![\ \uparrow \langle \mathrm{lft}(b), d \rangle\ ]\!]
$$

Finally the bottle is removed from the filling process through channel q.

Filling Controller

The filling controller coordinates the different activities of a filling line. Its specification is given by:

$$
\begin{array}{l}
\textsf{proc}\ FC(n : ?\,\mathrm{nat},\ r, s, a : {}^{\sim}\,\mathrm{void},\ p : ?\,\mathrm{real},\ v : \updownarrow \mathrm{vol}, \\
\qquad\qquad c : !\,\mathrm{cmd},\ d : !\,\mathrm{dat},\ v_f : \mathrm{real}) = \\
[\![\ k : \mathrm{nat},\ pH_c : \mathrm{real},\ v_c : \mathrm{vol} \\
\mid v \multimap v_c \\
\mid *[\ n?k \\
\quad ; *[\ k > 0 \longrightarrow p?pH_c \\
\qquad\qquad\qquad ; s^{\sim}; a^{\sim} \\
\qquad\qquad\qquad ; c!\mathsf{open}; \nabla v_c \geqslant v_f; c!\mathsf{close} \\
\qquad\qquad\qquad ; d!\langle pH_c, v_c \rangle \\
\qquad\qquad\qquad ; k := k - 1 \\
\qquad] \\
\quad ; r^{\sim} \\
\quad] \\
]\!]
\end{array}
$$

After receiving an order on channel n to fill k bottles, the filling controller starts filling. For each bottle, it first checks whether the pH in the vessel is within range ($p?pH_c$). Then it puts a new bottle under the filler by first triggering the bottle release process ($s^{\sim}$) and then waiting for the bottle to arrive at the filling position ($a^{\sim}$). It opens the filling valve ($c!\mathsf{open}$), waits until the bottle is filled ($\nabla v_c \geqslant v_f$) and closes the valve ($c!\mathsf{close}$). When the bottle is filled, the label data is sent to the filling process to complete the filling ($d!\langle pH_c, v_c \rangle$). After filling k bottles, a ready signal is sent back to the System Controller ($r^{\sim}$).

5.5 Bottle wrapping

The bottle wrapping process or packaging machine is the last process in the specification of the bottle-filling system. It collects the filled bottles that are received from the filling lines. The specification of the packaging machine reads as follows:

$$
\begin{array}{l}
\mathsf{proc}\ W(d_w : \mathrm{ord},\ f_x, f_y : ?\,\mathrm{bot},\ b_f : !\,\mathrm{bot}^*) = \\
[\![\ d : \mathrm{ord},\ n_x, n_y : \mathrm{nat},\ b_x, b_y : \mathrm{bot}^*,\ b : \mathrm{bot} \\
\mid\ *[\ b_x := [\,];\ b_y := [\,] \\
\quad ;\ d_w\,?\,d \\
\quad ;\ n_x := \mathrm{nx}(d);\ n_y := \mathrm{ny}(d) \\
\quad ;\ *[\ n_x > 0;\ f_x\,?\,b \longrightarrow b_x := b_x +\!\!+ [b];\ n_x := n_x - 1 \\
\qquad [\!]\ n_y > 0;\ f_y\,?\,b \longrightarrow b_y := b_y +\!\!+ [b];\ n_y := n_y - 1 \\
\qquad] \\
\quad ;\ b_f\,!\,b_x +\!\!+ b_y \\
\quad] \\
]\!]
\end{array}
$$

We use two lists of type bot^* to hold the bottles, one for each size. From the order d, the two amounts of ordered bottles are extracted with the functions nx and ny. The two counters n_x and n_y denote the number of small and large bottles respectively that are to be received to complete the delivery for the current order. Finally, the two lists are concatenated to a single list and sent to the customer.

5.6 Summary

In this chapter, we have set up a specification of a bottle-filling system. It serves as an illustration for the use of the formalism χ. First the system has been identified and isolated from its environment. Relevant elements in this environment are included in the specification to gain a closed system.

The different subsystems have been identified and a specification has been given for each of the processes that comprise the subsystems. Data structures have been defined for the representation of objects and substances that exist in the system. The relations between elements in the specification are modelled with channels between the model elements.

In the resulting specification, some processes have a discrete behaviour, some a continuous behaviour, while others have a combined discrete and continuous behaviour. The formalism provides the necessary means to create such a specification. The decision for a certain kind of behaviour is made at a process level.

Depending on this decision, the relations with the process' environment are defined. When the relations have been defined, the structure of the specification can be set up. This implies that the formalism prescribes a bottom-up approach when creating a specification. However, as illustrated in this chapter, also a top-down approach can be applied. The structure of a system can be defined using the graphical representation in which not all relations have to be defined from the start. After the specification of the identified model elements, the final relations can be supplied to complete the specification.

The given objectives for the specification of the bottle-filling system are fictive and vaguely formulated. Since a specification heavily depends on this problem statement, many different specifications may qualify for the given objectives. The presented specification is just one instance of the possible specifications. The primary goal of this chapter is to illustrate the use of the formalism and not to answer questions about the system.

Chapter 6

Design Decisions

In Chapter 4, we introduced the formalism, named χ, for the specification of the dynamical behaviour of systems. The use of this formalism is illustrated by an example of a bottle-filling system in Chapter 5. In this chapter, we review the design process that led to the formalism χ. A great number of design decisions have been made during this process. Some of these design decisions are self-explanatory while others may be based on aesthetic grounds. In this chapter, a number of interesting design decisions are discussed and some possible alternative designs are presented.

6.1 Basic concepts

The formalism is based on the idea that systems can be seen as to consist of concurrently operating subsystems or processes. This approach is known as the process-interaction approach. In earlier developments, this approach has proved to be suitable for the description of the dynamical behaviour of industrial systems [Roo82b, Ove87, Wor91]. The major advantages of this approach are its modularity and the possibility to deal with the parallelism in a system. The approach originates from the field of discrete-event modelling techniques. A similar approach has been used for the specification of continuous systems in, for instance, Omola [And90]. In our formalism, we use the process-interaction approach as the basis for the specification of both discrete and continuous behaviours, as well as a combination of the two.

The description of the discrete behaviour of a process is defined with a sequential program. The parallelism in a system has been taken into account in the decomposition of a system into (sequentially operating) processes. There is, therefore, no need to define additional parallelism in a process. Furthermore, it is much

easier to think about sequential activities than about parallel activities when modelling a process' behaviour.

The continuous behaviour of a process is defined by a set of differential and algebraic equations. Although it is not included in the definition of the formalism as yet, the use of partial differential equations is yet allowed. Nevertheless, in the field of application, for which the formalism is designed, we do expect to encounter systems that require an extensive use of partial differential equations. Also, we made the restriction to differential equations for reasons concerning the implementation of a supporting simulation tool.

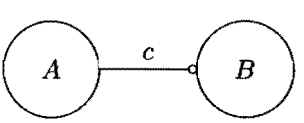
6.2 Continuous channels and links

The relation, defined by a continuous channel, has been explained in the Section 'Variable linking' on Page 36. There are, however, a number of considerations that led to this definition of a continuous channel. These are discussed in this section.

A continuous variable of a process represents a physical quantity that is part of the process' state. A single quantity may be used in different processes. Since all variables are local variables, different variables are used to represent the same physical quantity. This relation, between local variables of different processes, is defined by a continuous channel connecting the processes involved. Note that a channel is a one-to-one connection between two processes. Therefore, if more processes are involved, more channels are needed to define the relations. Recall the example of two connected vessels (see: Page 37):

$$
\begin{array}{l}
\mathsf{syst}\ S = \\
[\![\ c : [\mathrm{m}^3 \cdot \mathrm{s}^{-1}] \mid A(c) \parallel B(c)\ ]\!]
\end{array}
$$

$$
\begin{array}{l}
\mathsf{proc}\ A(c_1 : \uparrow [\mathrm{m}^3 \cdot \mathrm{s}^{-1}]) = \\
[\![\ \varphi_1 : [\mathrm{m}^3 \cdot \mathrm{s}^{-1}],\ V_1 : [\mathrm{m}^3] \\
\mid c_1 \multimap \varphi_1 \\
\mid V_1' = -\varphi_1 \\
]\!]
\end{array}
$$



$$
\begin{array}{l}
\mathsf{proc}\ B(c_2 : \downarrow [\mathrm{m}^3 \cdot \mathrm{s}^{-1}]) = \\
[\![\ \varphi_2 : [\mathrm{m}^3 \cdot \mathrm{s}^{-1}],\ V_2 : [\mathrm{m}^3] \\
\mid c_2 \multimap \varphi_2 \\
\mid V_2' = \varphi_2 \\
]\!]
\end{array}
$$

In this example, there is a liquid flow from vessel A to vessel B. The specifications of the processes A and B both refer to this physical quantity. In process A the local variable φ_1 is used for this purpose, in process B the variable φ_2 is used. This relation between the variables φ_1 and φ_2 is defined by the channel c in system S.

Thus, a continuous channel defines a relation between a variable of one process and a variable of another process. Both variables are continuous variables and represent the same physical quantity. The relation, defined by a continuous channel, is an equality relation. This means that the value of the first variable is equal to the value of the second variable ($\varphi_1 = \varphi_2$). A physical quantity, however, is more than just a numerical value.

The value of a quantity is expressed in a certain unit of measurement. In our example, φ_1 is expressed in $[\mathrm{m}^3 \cdot \mathrm{s}^{-1}]$. The unit of measurement in which a value is expressed, defines the meaning of the value. A relation between two values only makes sense if the values are expressed in the same unit of measurement. Suppose that φ_2 were expressed in $[\mathrm{l} \cdot \mathrm{s}^{-1}]$ (litres per second), then the relation $\varphi_1 = \varphi_2$ is no longer valid. Therefore, to define a relation between two variables as described above, we demand that both variables are expressed in the same unit of measurement. To achieve this, we have introduced the unit type by which the unit of measurement is defined. Furthermore, a channel can only be connected to variables of the same type, thus ensuring that the units of the connected variables are the same.

We can also resolve this problem by a unit transformation. If two variables are to be related that are expressed in different units of measurement, their values must be transformed to the same unit of measurement. Only then, the variables can be set equal to each other. In this solution, we need to define the unit of measurement for each continuous variable. Furthermore, when relating two variables by a channel, a transformation must be defined. The channel would be an obvious choice to handle the transformation. For example, if φ_1 is expressed in $[\mathrm{m}^3 \cdot \mathrm{s}^{-1}]$ and φ_2 in $[\mathrm{l} \cdot \mathrm{s}^{-1}]$, this would lead to the relation $1000 \times \varphi_1 = \varphi_2$.

Both solutions require the definition of the unit of measurement of each continuous variable. The second solution, however, has the disadvantage of an additional definition of a unit transformation. To keep our formalism as simple as possible, we have decided in favour of the first solution.

The second issue in the representation of a physical quantity is the definition of a direction. Some physical quantities have an associated direction that is essential to their definition. We distinguish two kinds of quantities: scalar quantities and vector quantities. The difference between the two kinds is that vector quantities have a direction and scalar quantities have not. Examples of the first kind are temperature, pressure, and mass. Examples of the second kind are force, velocity, and all kinds of flows.

The direction of a vector quantity, is usually expressed in a properly chosen coordinate system. Depending on the dimension of the coordinate system, we can distinguish one, two, and three dimensional vector quantities. As with the unit of measurement, an equality relation between two variables can only be established if the associated directions of the quantities, represented by the variables, are expressed in the same coordinate system. This problem is similar to the earlier mentioned problem and, thus, we have two possible solutions. The first is to ensure that variables, that are connected by a channel, are expressed in the same coordinate system. The second solution is a coordinate system transformation.

A coordinate system transformation requires the definition of the coordinate system for each continuous variable. Furthermore, the transformation must be defined within the connecting channel. This would surely increase the complexity of variable and channel definitions in our formalism.

To apply the first solution, we must make a few restrictions to the physical quantities that can be represented by variables. A channel is a one dimensional connection between two processes. If we define a direction for a continuous channel (as we have done for discrete channels), the channel can serve as a coordinate system definition for the connected variables. Since a channel is declared in the parameter list of a process, this coordinate system is locally available to the process and its variables. In this way, no additional coordinate system definitions or transformations are needed. Note that we restrict ourselves to one dimensional vector quantities. However, we expect that higher dimensional quantities are not likely to occur very often in the specification of industrial systems.

6.3 Discrete channels

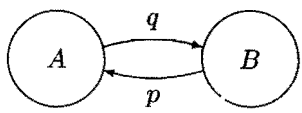
Discrete interactions can be defined with a synchronous or an asynchronous interaction mechanism. With a synchronous mechanism, the process that performs the interaction statement first, will be suspended until its communication partner performs the counterpart statement. An asynchronous mechanism buffers the send action until the other process consumes it with a receive action. The sending process can never be suspended, the receiving process only if no sending actions are currently buffered. Another difference is that a synchronous mechanism can work in both directions, where an asynchronous mechanism is uni-directional.

The two mechanisms are not mutually exclusive. A synchronous interaction can be described using two asynchronous interactions. Consider the following example:

```
syst S_a =
[[ p, q : void
| A(p, q) || B(p, q)
]]

proc A(p : ??void, q : !!void) =
[[ ... ; p??; q!!; ... ]]

proc B(p : !!void, q : ??void) =
[[ ... ; p!!; q??; ... ]]
```



The two channels p and q are asynchronous channels. In this example, an asynchronous send action on a channel p is denoted by $p!!$, and a receive action by $p??$. The synchronisation is initiated by an asynchronous communication on channel p. This ensures that both processes are ready for a communication on channel q, by which the actual synchronisation is established.

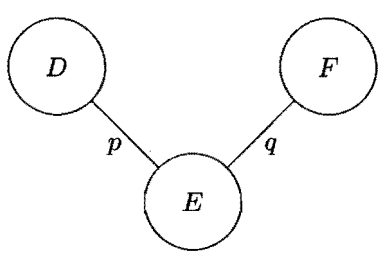
Also, an asynchronous interaction can be described by two synchronous interactions and an additional process, a buffer. The buffer stores the interaction, sent by the process D in the following example:

```
syst S_s =
[[ p, q : void
| D(p) || E(p, q) || F(q)
]]

proc D(p : ~ void) =
[[ ... ; p~; ... ]]

proc E(p, q : ~ void) =
[[ *[ p~; q~ ] ]]

proc F(q : ~ void) =
[[ ... ; q~; ... ]]
```



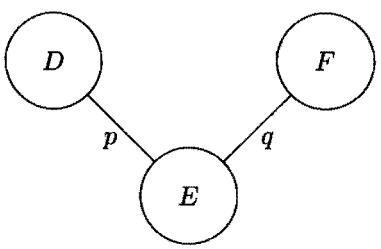
Here, the buffer E is a one-place buffer. Only one interaction from process D can be stored, so that, if process F does not perform an interaction, a second interaction will suspend the process D. If more interactions should be buffered, a buffer with an infinite capacity can be applied:

```
syst S_s =
[[ p, q : void
| D(p) || E(p,q) || F(q)
]]

proc D(p : ~ void) =
[[ ... ; p~ ; ... ]]

proc E(p, q : ~ void) =
[[ n : nat
| n := 0
; *[          ; p~ ⟶ n := n + 1
   [] n > 0 ; q~ ⟶ n := n - 1
   ]
]]

proc F(q : ~ void) =
[[ ... ; q~ ; ... ]]
```



If there exists a buffer in a system to hold synchronisation or communication messages, it should be visible in a specification. Only then, the capacity of the buffer can be estimated instead of making it infinite.

6.4 Connecting channels

All channels (discrete and continuous) are one-to-one connections. Other connection mechanisms would be one-to-many, many-to-one, and many-to-many. These mechanisms, however, have some disadvantages over one-to-one connections which are discussed next.

Suppose we have a discrete communication channel that is connected using a one-to-many connection with other processes. The data, supplied by the sending process, can be received by any of the receiving processes. It is not known in advance which process will receive the data. We have thus modelled an implicit non-deterministic choice.

Similar situations occur with many-to-one connections and many-to-many connections. We aim at specifications in which the behaviour of a system is explicitly described. We, therefore, refrain from using any form of implicit modelling. If

in a system, a non-deterministic choice has to be made, we prefer to explicitly describe this choice in the behaviour of a process. This can be done using a selective waiting command as described on Page 40. Moreover, one-to-one connections are simpler in their definition and use and are therefore preferable according to the requirements for our formalism.

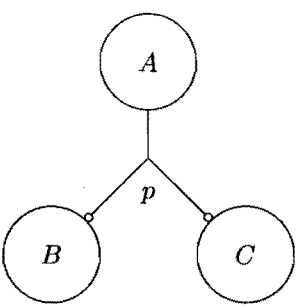
In Section 6.2, we have discussed the meaning of continuous channels. We have described how to use the direction of a channel to define a coordinate system for the linked variables. This method can only be used with one-to-one connected channels. Consider, for example, the following system:

```
syst S =
[[ p : [m^3·s^-1]
| A(p) || B(p) || C(p)
]]

proc A(p : ↑[m^3·s^-1]) =
[[ a : [m^3·s^-1]
| p ⊸ a
⋮
]]

proc B(p : ↓[m^3·s^-1]) =
[[ b : [m^3·s^-1]
| p ⊸ b
⋮
]]

proc C(p : ↓[m^3·s^-1]) =
[[ c : [m^3·s^-1]
| p ⊸ c
⋮
]]
```



In this example, we have defined a one-to-many connected channel p. The flow, represented by variable a in process A is an outgoing flow. In the two processes B and C we have two incoming flows, represented by the variables b and c respectively. All three variables are linked to the channel p. For the semantics of the relation, defined by this channel, we have two possibilities. The first is that all variables have an equal value:

$$a = b = c$$

This is, however, physically impossible. It would imply that medium is created when it flows from A to B and C.

The second option for the semantics is to split the flow a into two subflows b and c. This results in a relation of the form:

$$a + b + c = 0$$

The flows b and c are not properly defined in this way. The ratio between b and c cannot be determined. The values of these two subflows are determined by the characteristics of the pipes through which the medium flows. It depends, for instance, on the construction of the T-fork. The behaviour of the medium at the fork should be specified in a process. The latter solution is used in, for instance, Dymola [Elm78].

Summarising, we can say that channels, connected to more than two processes, tend to define a certain implicit behaviour. Because we prefer to specify the behaviour explicitly in processes, we restrict channels to be one-to-one connections only.

6.5 Mixed behaviour

The behaviour of a process can be continuous, discrete, or a combination. Interactions between the continuous and discrete behaviour are specified within a process. If we disallow a mixed behaviour in a process, the interactions between continuous and discrete processes would have to be defined by channels. This implies that more kinds of channels must be defined. For instance, we need to have a channel to define a state-event, an interaction from a continuous to a discrete behaviour. Another kind of channel defines the interaction from the discrete behaviour to the continuous behaviour, a discontinuity.

Setting aside how such channels should be defined, they implicitly model a certain behaviour. We prefer to specify the behaviour in the processes and not in the channels. Furthermore, the number of channel types would increase and would make the formalism more complex. This can be avoided when allowing a mixed behaviour in a process.

6.6 The language

Many design decisions have led to the set of language constructs as described in Chapter 4. Some of these decisions are discussed here.

Keywords and symbols

We have decided to use symbols instead of keywords to define our language elements. For example, to enclose the behaviour description of a process, we do not use Pascal-like keywords like begin and end, but we use the symbols ⟦ and ⟧ instead. Also, to delimit a guarded command we use the symbols [and]. Other symbols have been defined for operators like σ for sampling a statistical distribution and ∇ to denote state-events.

The use of symbols, as opposed to keywords, results in more compact descriptions that are easier to comprehend. The use of keywords tends to make a specification more fuzzy because the keywords attract the attention of the reader. This will distract the reader in perceiving the actual specification. To illustrate this, compare the two following descriptions of a buffer process.

```
proc B(p : ? any, q : ! any, n : nat) =
⟦ x : any, xs : any*
| xs := []
; *[ len(xs) < n ; p ? x        ⟶ xs := xs ++ [x]
   [] len(xs) > 0 ; q ! hd(xs) ⟶ xs := tl(xs)
   ]
⟧

proc B(din p : any, dout q : any, par n : nat)
begin
   var x : any, xs : list of any;
   xs := [];
   do len(xs) < n ; p recv x        then xs := xs ++ [x]
   alt len(xs) > 0 ; q send hd(xs) then xs := tl(xs)
   od
end
```

Observe that we have made an exception to use keywords to denote the beginning of a process, a system, a type or unit definition, and a function. Here, we make use of the earlier described effect that keywords have: focus the reader's attention to the beginning of a specification.

Keywords are usually words derived from a natural language. We choose the English language to derive our keywords from. Avoiding too large a dependence on English is another reason for using as many symbols as possible. It makes the formalism more accessible to non-English users. Note, that the keywords that we use for systems, processes, and functions are four-letter keywords. We abbreviated these words to make them more practicable and because the translations of these words in many natural languages only differ from their English counterparts after the first four letters.

Types

A crucial point when making a model is how to represent the data that exists in the model. Many different kinds of data should be representable in comprehensive data structures. As in programming languages, we use variables to store the data of our models in. Furthermore, our specification language is a strongly typed language which means that each variable is of a certain type. The types we proposed for building data structures are tuples and lists.

For the sake of simplicity of concept, we only define these two structured types. The first is intended to define a fixed data structure in which data of possibly different types can be grouped. A list is a collection of data elements of one type where the number of elements, contained in the collection, may vary.

The restriction to tuples and lists as the only structured types seems to be adequate to specify the systems we aim at. Well-known data types, such as arrays and objects or records, may easily be defined with these two types. In Smalltalk-80 [Gol83], other, more flexible, data structures are defined. It is, for instance, possible to create a list, containing elements of different types. We refrain from defining such data structures. Moreover, a data structure, built up of this kind of lists, should carefully be reconsidered. Such data structures may lead to unpredictable or even erroneous behaviour due to the inability to anticipate what data type to expect when manipulation such data structures.

Combining event statements

Although a guarded command seems to be a complex language construct at first, it contributes to the requirement of a minimal number of primitives. It provides the required control structures to describe the possible sequence in which statements are performed. It allows simple selection and selective event-handling (communication, time-outs and state-events), both guarded by Boolean expressions, as well as repetition. A well appreciated advantage of the guarded command is its ability to combine the different types of event statements in an orthogonal manner.

The selective waiting command, a variant of the guarded command, combines events, from which one is chosen for execution. There is yet another way of combining events. Suppose we want to model an assembly workstation. In this workstation, several parts are needed to assemble one product. The order, in which the parts arrive at the workstation, is not relevant. The following part of a specification illustrates this:

$$\ldots; x?p; y?q; \ldots$$

Note, however, that the process, that provides the assembly parts q is suspended until the parts p is received. An additional language construct can be used to

avoid such suspending of processes. In the next example, the two events are combined in such a way, that each event must be executed once, but their order is not prescribed:

$$\ldots ; x?p, y?q; \ldots$$

Although this language construct seems useful, we refrain from including it in the formalism. The aforementioned situation does not seem to occur frequently enough to define an additional language construct for it. Moreover, it can be constructed with a guarded command as follows:

$$\ldots ; [\ x?p \longrightarrow y?q \ [\!]\ y?q \longrightarrow x?p\ ]; \ldots$$

Stochastic behaviour

Statistical distributions form the basis for the specification of non-deterministic discrete behaviour. Although we give a list of predefined distributions in Appendix B, this list may freely be extended to one's needs. The formalism provides a type (dist) and an operator (σ) on this type to draw a sample from a distribution.

Where the creation of a distribution is defined by a function, taking a sample from a distribution cannot be defined by a function. A function must give the same result each time it is called with the same set of arguments. This is normally not the case when drawing a sample from a distribution. Therefore, we defined an operator for this purpose at the cost of an additional primitive in our specification language.

6.7 Functions and procedures

The formalism provides a means to define functions. These functions can be used for data manipulation only, not for communication with other processes. Also, we do not allow the definition of procedures. Procedures can be useful in creating various abstraction levels within a process specification. On the other hand, if procedures are used, the specification of a process is distributed over different parts of a specification. Without procedures, a process is specified in one single process description.

In our experience, process specifications tend to be rather small. They easily fit on a single letter size sheet of paper. Furthermore, the structures of the processes we came across appear to be straightforward and do not require procedural abstraction to make them easier to comprehend. For the time being, we refrain from including a form of procedural abstraction in the formalism. It would

make the formalism more complex and it does not increase its applicability or usability.

The one form of abstraction within a process specification we do allow is the abstraction of data manipulation. Particularly control processes incline to largely consist of data manipulation statements. The use of functions for this purpose, can help to enlighten the structure of a process. By abstracting from the data manipulation, the communication of a process becomes more distinguishable. Furthermore, the data, handled by a function, is defined with global data types. It therefore seems justified to define global functions that act upon these data types. Moreover, we recommend to arrange the data type definitions and their accompanying functions close together.

Chapter 7

The State-Transition Controller: A Case

In this chapter, we present a case in which the formalism is applied to the development of a control system. The case is adopted from Van de Kamp and Voorbraak [Kam95]. The controlled system is a hot water tank. From this tank, containers are filled with water that has a certain temperature. The water tank is similar to the vessel, in the example described in Chapter 3 and Chapter 5. The difference is that, in the water tank system, not the pH value of the liquid is controlled but the temperature instead.

The formalism is used in several stages in the development of the control system. First, the hot water tank system and its environment are identified and a model of this system and its control system is set up. The behaviour of the control processes depend on the chosen control concepts. In [Kam95], several control concepts are elaborated, such as P- and PI-controllers, sliding-mode controllers, fuzzy logic controllers, state-transition controllers, and neural-net controllers. In this chapter, we only use one control concept: state-transition control. After the model has been completed, the control system is tested, using a simulator. Finally, it is used in the real-life system to control the water tank system.

A schematic diagram of the water tank system is shown in Figure 7.1. It shows the water tank and its control system. Two sensors, marked T and L, measure respectively the temperature and the level of the water in the tank. The first task of the control system is to keep both temperature and level within given limits. The temperature and the level can be adjusted by supplying hot or cold water. A surplus of water can be drained off. The hot water, cold water and drain flows are controlled by opening and closing the appropriate valves. The second task of the control system is to open and close the filling valve to fill the containers.

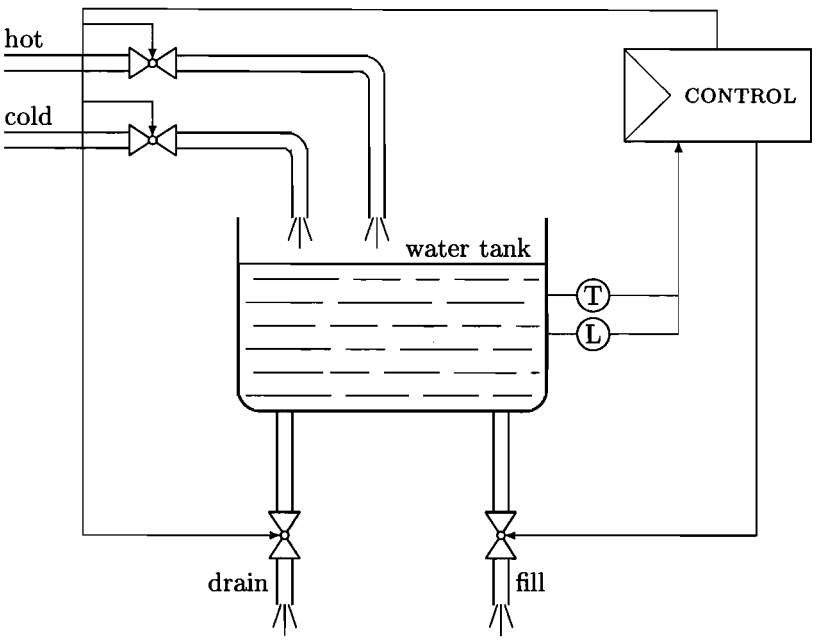
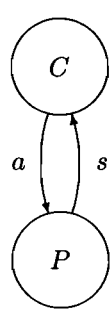


Figure 7.1: Schematic diagram of the water tank system.

From the schematic diagram, two subsystems can be identified: the physical system and the control system. The physical system consists of the water tank, the valves in the input and output streams, and the two sensors. The interactions between the two subsystems consist of the sensor information about the current temperature and level in the tank and the actuator information for controlling the valves. The model W of the system reflects this subdivision in subsystems in:

$$
\begin{array}{l}
\mathsf{syst}\ W = \\
[\![\ a : (\mathrm{int}^4),\ s : (\mathrm{int}^2) \\
\mid\ C(s,a) \parallel P(a,s) \\
]\!]
\end{array}
$$



The physical system, the water tank and all its fittings, is denoted by P. C represents the control system. The interactions between the water tank and

its control system are modelled by two bundles, one for the sensor information (bundle s), and one for the actuator signals (bundle a).

7.1 The controlled system

The controlled system is the physical part of the water tank system. We consider the suppliers of hot and cold water, as well as the drain, to be part of the physical system. The model of the hot water tank is depicted in Figure 7.2.

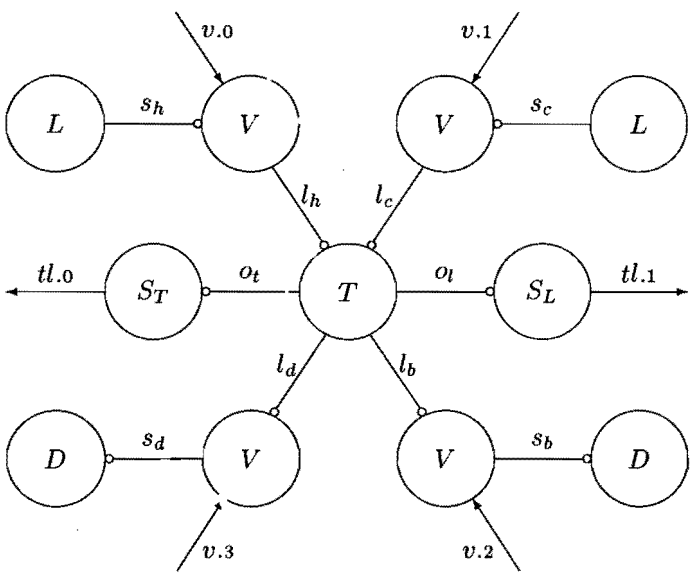


Figure 7.2: The water tank system P.

In this figure, the valves are modelled by the processes V. L are water suppliers and D are processes that draw water form the tank, e.g. the drain and the containers. The sensors are named S_T and S_L. Furthermore, for the various channels, we use the indexes h and c for respectively hot and cold water streams, an index d is used for streams associated with the drain, and an index f is used for the streams concerning the filing of containers.

The interactions with the control system are modelled by two bundles. The actuator information controls the valves and are grouped in the bundle v. Since there are four valves, this bundle contains four channels, indexed 0 through 3. The sensor information is passed to the controller by two channels in the bundle tl.

Before the model of the water tank system can be given, a number of types are defined:

$$\begin{array}{lll}
\textsf{unit}\ \mathrm{N} & = \mathrm{kg{\cdot}m{\cdot}s^{-2}} & \\
 & & \\
\textsf{type}\ \mathrm{flow} & = [\mathrm{m^3{\cdot}s^{-1}}] & \\
,\ \ \mathrm{press} & = [\mathrm{N{\cdot}m^{-2}}] & \\
,\ \ \mathrm{temp} & = [\mathrm{K}] & \\
,\ \ \mathrm{liq} & = \langle \mathrm{flow} \times \mathrm{press} \times \mathrm{temp} \rangle & \\
,\ \ \mathrm{level} & = [\mathrm{m}] &
\end{array}$$

Furthermore, the following constants are defined that are used in the specification of the hot water tank system:

$$\begin{array}{llll}
\textsf{const}\ A_t & = 0.036 & \text{-- } [\mathrm{m^2}] & \text{tank cross section} \\
,\ \ p_h & = 1.25{\cdot}10^5 & \text{-- } [\mathrm{N{\cdot}m^{-2}}] & \text{hot water pressure} \\
,\ \ p_c & = 4.30{\cdot}10^5 & \text{-- } [\mathrm{N{\cdot}m^{-2}}] & \text{cold water pressure} \\
,\ \ p_{atm} & = 1.03{\cdot}10^5 & \text{-- } [\mathrm{N{\cdot}m^{-2}}] & \text{atmospheric pressure} \\
,\ \ T_h & = 333 & \text{-- } [\mathrm{K}] & \text{hot water temperature} \\
,\ \ T_c & = 295 & \text{-- } [\mathrm{K}] & \text{cold water temperature} \\
,\ \ \tau_v & = 0.1 & \text{-- } [\mathrm{s}] & \text{valve opening/closing time} \\
,\ \ \alpha & = 8{\cdot}10^{-9} & \text{-- } [\mathrm{m^5{\cdot}N^{-1}{\cdot}s^{-1}}] & \text{characteristic value} \\
 & & \text{--} & \text{for the valves} \\
,\ \ N_T & = 2 & \text{-- number of temperature boundaries} & \\
,\ \ N_L & = 2 & \text{-- number of level boundaries} & \\
,\ \ R_T & = \langle 307.75, 308.25 \rangle & \text{-- } \langle [\mathrm{K}]^{N_T} \rangle & \text{temperature boundaries} \\
,\ \ R_L & = \langle 0.0475, 0.0525 \rangle & \text{-- } \langle [\mathrm{K}]^{N_L} \rangle & \text{level boundaries}
\end{array}$$

The constants N_T, N_L, R_T, and R_L are used in the state-transition controller. Their meaning is described in Section 7.2. The model of the physical system, as depicted in Figure 7.2, is represented by:

$$\begin{array}{l}
\textsf{syst}\ P(v : ?(\mathrm{int}^4),\ tl : !(\mathrm{int}^2),\ k_T, k_L : \mathrm{nat}) = \\
[\![\ s_h, s_c, s_d, s_f, l_h, l_c, l_d, l_f : \mathrm{liq},\ o_t : \mathrm{temp},\ o_l : \mathrm{level} \\
\mid\ T(l_h, l_c, l_d, l_f, o_l, o_t, A_t) \\
\|\ L(s_h, p_h, T_h) \qquad \|\ V(s_h, l_h, v.0, \alpha, \tau_v) \\
\|\ L(s_c, p_c, T_c) \qquad \|\ V(s_c, l_c, v.1, \alpha, \tau_v) \\
\|\ V(l_d, s_d, v.2, \alpha, \tau_v) \qquad \|\ D(s_d, p_{atm}) \\
\|\ V(l_f, s_f, v.3, \alpha, \tau_v) \qquad \|\ D(s_f, p_{atm}) \\
\|\ S_T(o_t, tl.0, k_T, N_T, R_T) \qquad \|\ S_L(o_l, tl.1, k_L, N_L, R_L) \\
]\!]
\end{array}$$

Control valves

The control valves are similar to the valves used in the bottle-filling system of Chapter 5. However, the opening and closing of the valves does not occur instantaneous. It takes τ_v seconds to open or close a valve. This is modelled by the second (guarded) differential equation in the following specification of a control valve:

$$
\begin{array}{l}
\textsf{proc}\ V(l_1 : {\downarrow}\textsf{liq},\ l_2 : {\uparrow}\textsf{liq},\ c : ?\textsf{int},\ \alpha, \tau_v : \textsf{real}) = \\
[\![\ \varphi : \textsf{flow},\ p_1, p_2 : \textsf{press},\ T : \textsf{temp},\ r : [],\ s : \textsf{int} \\
\mid\ l_1 \multimap \langle \varphi, p_1, T \rangle,\ l_2 \multimap \langle \varphi, p_2, T \rangle \\
\mid\ \varphi = r \times \alpha \times (p_1 - p_2) \\
,\ [\ r \geqslant 0.0 \wedge r \leqslant 1.0 \longrightarrow r' = s/\tau_v \\
\quad [\!]\ r \leqslant 0.0 \wedge s \leqslant 0.0 \longrightarrow r' = 0.0 \\
\quad [\!]\ r \geqslant 1.0 \wedge s \geqslant 0.0 \longrightarrow r' = 0.0 \\
\quad] \\
\mid\ *[\ c?s\ ] \\
]\!]
\end{array}
$$

Note, that we assume $p_1 \geqslant p_2$ at all times. Otherwise, the specification should be changed as described on Page 69.

Water supply and drain

The water supply (L) and drain (D) processes are self-explanatory and are specified with:

$$
\begin{array}{l}
\textsf{proc}\ L(l : {\uparrow}\textsf{liq},\ p_s, t : \textsf{real}) = \\
[\![\ \varphi : \textsf{flow},\ p : \textsf{press},\ T : \textsf{temp} \\
\mid\ l \multimap \langle \varphi, p, T \rangle \\
\mid\ \varphi^2 = \alpha \times (p_s - p),\ T = t \\
]\!]
\end{array}
$$

$$
\begin{array}{l}
\textsf{proc}\ D(l : {\downarrow}\textsf{liq},\ p_s : \textsf{real}) = \\
[\![\ \varphi : \textsf{flow},\ p : \textsf{press},\ T : \textsf{temp} \\
\mid\ l \multimap \langle \varphi, p, T \rangle \\
\mid\ p = p_s \\
]\!]
\end{array}
$$

Water tank

The water temperature decreases due to heat losses to the environment of the tank. Three kinds of heat losses are distinguished: convection, conduction, and

radiation. The influence of radiation on the water temperature is considered to be insignificant and thus not included in the model. For the remaining two kinds of heat losses, resistance factors are introduced.

The continuous behaviour of the hot water tank is described by the mass and energy balance equations:

$$
\begin{aligned}
A \times h' &= \varphi_h + \varphi_c - \varphi_d - \varphi_f \\
A \times h \times T' &= \varphi_h \times (T_h - T) + \varphi_c \times (T_c - T) + \frac{\gamma_d \times h + \gamma_v}{c_p \times \rho} \times (T_e - T)
\end{aligned}
$$

in which:

φ_h	hot water flow	T_h	hot water temperature
φ_c	cold water flow	T_c	cold water temperature
φ_d	drain water flow	T	tank water temperature
φ_f	outlet water flow	T_e	environment temperature
h	liquid level	h	water level in the tank
A	tank cross section	g	accelaration of gravity
c_p	heat capacity	γ_d	heat resistance factor for conduction
ρ	specific mass	γ_v	heat resistance factor for convection

The specification of the water tank reads as follows:

$$
\begin{aligned}
&\text{proc } T(l_h, l_c : \downarrow \text{liq},\ l_d, l_b : \uparrow \text{liq},\ l : \uparrow \text{level},\ t : \uparrow \text{temp},\ A : \text{real}) = \\
&[\![\ \varphi_h, \varphi_c, \varphi_d, \varphi_b : \text{flow},\ p_h, p_c, p : \text{press},\ T_h, T_c, T : \text{temp},\ h : \text{level} \\
&\mid\ l_h \multimap \langle \varphi_h, p_h, T_h \rangle,\ l_c \multimap \langle \varphi_c, p_c, T_c \rangle,\ l_d \multimap \langle \varphi_d, p, T \rangle,\ l_b \multimap \langle \varphi_b, p, T \rangle \\
&\quad t \multimap T,\ l \multimap h \\
&\mid\quad A \times h' = \varphi_h + \varphi_c - \varphi_d - \varphi_b \\
&,\ A \times h \times T' = \varphi_h \times (T_h - T) + \varphi_c \times (T_c - T) + \frac{\gamma_d \times h + \gamma_v}{c_p \times \rho} \times (T_e - T) \\
&,\qquad p_h = p_{atm} \\
&,\qquad p_c = p_{atm} \\
&,\qquad p = p_{atm} + \rho \times g \times h \\
&]\!]
\end{aligned}
$$

The behaviour of a sensor, that measures the level or the temperature of the water in the tank, depends on the kind of control system. In the next section, we describe the control system for the hot water tank, using a state-transition control concept. Therefore, we postpone the specification of the sensors till the control concepts are discussed.

7.2 The control system

The control system we use in this example is a state-transition controller [Pre93]. This controller generates a control signal whenever a state-transition occurs. A state-transition occurs when a state variable crosses a boundary. For each state variable, a number of such boundaries are defined, resulting in a discretized state space.

For each state transition, a control signal is defined. All possible control signals for all possible state transitions are recorded in a so-called control logic table. Together with the choice of state boundaries, the design of such a table determines the quality of the controller. A number of design techniques for control logic tables are discussed in [Pre93, Kam95].

The control structure

The control system has two input values: the level and temperature of the water in the tank. Furthermore, the control system generates four output signals: one for each valve in the system. The control of the hot water tank system involves three phases. In the first phase, the temperature and the level are adjusted to meet their set-points. This phase is referred to as the *start-phase*. In the *filling-phase*, containers are filled with water of a certain temperature. After filling one batch of containers, a new batch must be supplied. Meanwhile, no containers can be filled, and the control system is said to be in its *rest-phase*.

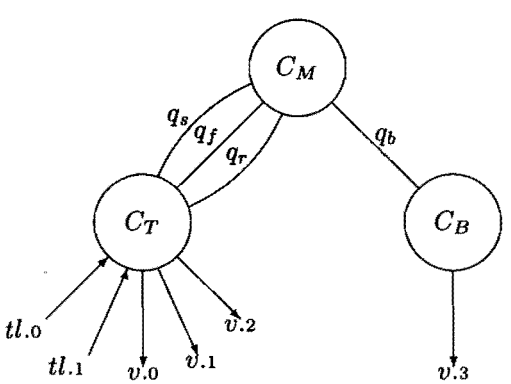


Figure 7.3: The water tank control system C.

The specification of the control system shows two subcontrollers and a supervisory controller. The task of the first subcontroller, the tank controller C_T, is to keep the level and temperature close to their set-points. This subcontroller controls three of the four valves: the hot and cold water supply valves, and the

drain valve. The second subcontroller, C_B, handles the filling of the containers by opening and closing the filling valve. The supervisory controller C_M coordinates the activities of the underlying subcontrollers. The structure of the control system C is given by Figure 7.3. The specification of the control system is given by:

$$
\begin{array}{l}
\textsf{syst } C(tl : ?(\text{int}^2),\ v : !(\text{int}^4)) = \\
[\![\ q_s, q_f, q_r, q_b : \text{void} \\
\mid\ C_M(q_s, q_f, q_r, q_b) \parallel C_B(q_b, v.3, 12) \\
\parallel C_T(q_s, q_f, q_r, tl.0, tl.1, v.0, v.1, v.2, 1, 1) \\
]\!]
\end{array}
$$

The interactions between the subcontrollers and the supervisory controller are explained on Page 103 where the model of the supervisory controller is discussed.

State-transition controller

A state-transition controller generates a control signal whenever a state transition occurs. A state transition is an event that occurs when a state variable crosses some boundary. The state space of a system is discretized by dividing the domain of each state variable into a number of subdomains. For a variable x, we define a set B of n state boundaries such that:

$$B = \{\, b_i : 1 \leqslant i < n : b_{i-1} < b_i \,\}$$

The domain of a variable x, therefore, has $n + 1$ subdomains. We define the discrete state variable $\hat{x}$ to represent the discrete state in which x resides such that:

$$x \in d_k \ \Rightarrow\ \hat{x} = k \qquad ; 0 \leqslant k \leqslant n$$

where:

$$
\begin{array}{ll}
d_0 = (-\infty, b_0\,] & \\
d_i = (b_{i-1}, b_i\,] & ; 1 \leqslant i < n \\
d_n = (b_n\ \ , +\infty) &
\end{array}
$$

In our example of the water tank, the state variables, that we monitor for state transitions, are the temperature and level of the water in the tank. The state space, represented by these variables, can be discretized as illustrated in Figure 7.4. We define two boundaries for both the temperature and the level, resulting in a 3×3 state space. A discrete state is denoted by the tuple $\langle T, L \rangle$, where T is the temperature and L is the level.

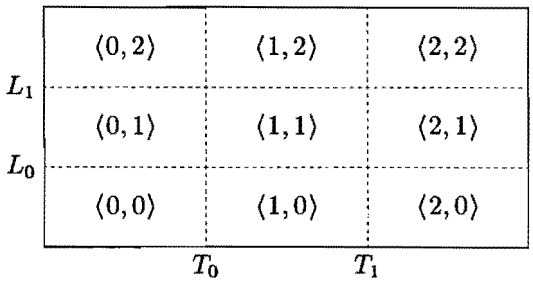


Figure 7.4: Discretized state space of the water tank system.

The control signal, generated for each state transition, consists of three output values, denoted by the tuple $\langle h, c, d \rangle$, where h is the control value for the hot water valve, c controls the cold water valve, and d controls the drain. The control values depend on the phase in which the tank controller operates. For the start-phase and rest-phase, the control values are identical. The values for the filling-phase deviate slightly from those of the other two phases.

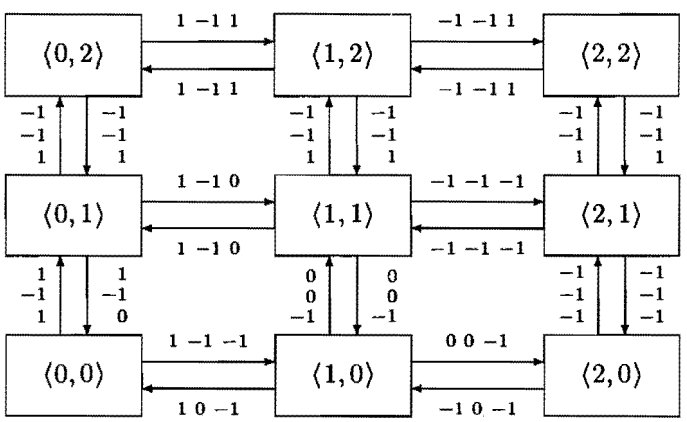


Figure 7.5: Control values for the start-phase and rest-phase.

In Figure 7.5, the control values for the start-phase and the rest-phase are given. These values are derived from [Kam95]. A value 1 denotes opening a valve, the value -1 denotes closing a valve, and the value 0 means that the current control value for the valve does not change. Now suppose, the state changes from $\langle 1, 0 \rangle$ to $\langle 0, 0 \rangle$. This implies that the temperature is decreasing. Due to this state change, the controller determines a new control action. From Figure 7.5, we can derive that the new control action is defined by the tuple $\langle 1, 0, -1 \rangle$. This means that the hot water valve is opened (denoted by the value 1), the state of the

cold water valve does not change (control value 0), and the drain valve is closed (control value -1).

The control values for the filling-phase are given in Figure 7.6.

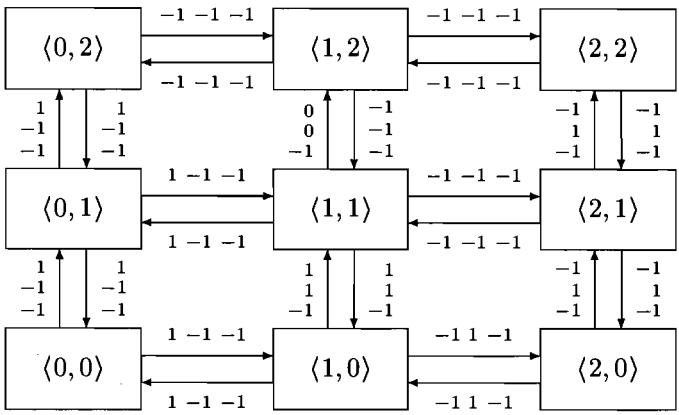


Figure 7.6: Control values for the filling-phase.

For the tank controller to determine the proper control values, they must be represented in a data structure, called the control logic table. For each phase, we distinguish four situations: the temperature increases or decreases, or the level increases or decreases. The control logic table for an increasing temperature in the start-phase is given by:

$$
\begin{aligned}
C^{+}_{TS} = \langle\ & \langle\langle\ \ 1,-1,-1\rangle, \langle\ \ 0,\ \ 0,-1\rangle\rangle \\
, & \langle\langle\ \ 1,-1,\ \ 0\rangle, \langle -1,-1,-1\rangle\rangle \\
, & \langle\langle\ \ 1,-1,\ \ 1\rangle, \langle -1,-1,\ \ 1\rangle\rangle \\
& \rangle
\end{aligned}
$$

The table C^{+}_{TS} contains the control actions associated with the left-to-right arrows in Figure 7.5. The first line in the table represents the control values for the situation in which the temperature decreases while the level is in state 0. Similarly, the second line of the table is associated with a changing temperature while the level is in state 1 and the last line of the table is associated with level state 2. Note that the table is up-side-down compared to Figure 7.5.

We have eight of these control logic tables, four for each phase. The index T is used for a changing temperature, the index L for a changing level. Furthermore, the tables for the start-phase are indexed with an S and the table for the filling-phase with an F. Finally, a $+$ denotes an increasing state, a $-$ denotes a decreasing state.

$$C^+_{TS} = \langle \langle\langle 1,-1,-1\rangle,\langle 0,0,-1\rangle\rangle, \langle\langle 1,-1,0\rangle,\langle -1,-1,-1\rangle\rangle, \langle\langle 1,-1,1\rangle,\langle -1,-1,1\rangle\rangle \rangle \quad C^-_{TS} = \langle \langle\langle 1,0,-1\rangle,\langle -1,0,-1\rangle\rangle, \langle\langle 1,-1,0\rangle,\langle -1,-1,-1\rangle\rangle, \langle\langle 1,-1,1\rangle,\langle -1,-1,1\rangle\rangle \rangle$$

$$C^+_{LS} = \langle \langle\langle 1,-1,1\rangle,\langle -1,-1,1\rangle\rangle, \langle\langle 0,0,-1\rangle,\langle -1,-1,1\rangle\rangle, \langle\langle -1,-1,-1\rangle,\langle -1,-1,1\rangle\rangle \rangle \quad C^-_{LS} = \langle \langle\langle 1,-1,0\rangle,\langle -1,-1,1\rangle\rangle, \langle\langle 0,0,-1\rangle,\langle -1,-1,1\rangle\rangle, \langle\langle -1,-1,-1\rangle,\langle -1,-1,1\rangle\rangle \rangle$$

$$C^+_{TF} = \langle \langle\langle 1,-1,-1\rangle,\langle -1,1,-1\rangle\rangle, \langle\langle 1,-1,-1\rangle,\langle -1,-1,-1\rangle\rangle, \langle\langle -1,-1,-1\rangle,\langle -1,-1,-1\rangle\rangle \rangle \quad C^-_{TF} = \langle \langle\langle 1,-1,-1\rangle,\langle -1,1,-1\rangle\rangle, \langle\langle 1,-1,-1\rangle,\langle -1,-1,-1\rangle\rangle, \langle\langle -1,-1,-1\rangle,\langle -1,-1,-1\rangle\rangle \rangle$$

$$C^+_{LF} = \langle \langle\langle 1,-1,-1\rangle,\langle 1,-1,-1\rangle\rangle, \langle\langle 1,1,-1\rangle,\langle 0,0,-1\rangle\rangle, \langle\langle -1,1,-1\rangle,\langle -1,1,-1\rangle\rangle \rangle \quad C^-_{LF} = \langle \langle\langle 1,-1,-1\rangle,\langle 1,-1,-1\rangle\rangle, \langle\langle 1,1,-1\rangle,\langle -1,-1,-1\rangle\rangle, \langle\langle -1,1,-1\rangle,\langle -1,1,-1\rangle\rangle \rangle$$

We combine the corresponding tables for an increasing and a decreasing state into one tuple, which results in the following four control logic tables:

$$\begin{aligned}
&\textsf{const } C_{TS} = \langle C^+_{TS}, C^-_{TS}\rangle \\
&\quad , \quad C_{TF} = \langle C^+_{TF}, C^-_{TF}\rangle \\
&\quad , \quad C_{LS} = \langle C^+_{LS}, C^-_{LS}\rangle \\
&\quad , \quad C_{LF} = \langle C^+_{LF}, C^-_{LF}\rangle
\end{aligned}$$

The function ctrl is defined to obtain the proper control values for a given state transition in a certain control phase as:

$$\begin{aligned}
&\textsf{func ctrl}(p, s : \text{char},\ t, l : \text{nat},\ c : \text{int}) = \\
&[\![\ c := \text{sgn}(1-c) \\
&;\ [\ s = \textsf{T} \longrightarrow t := \min(t, t-c) \\
&\qquad\qquad ;\ [\ p = \textsf{F} \longrightarrow \uparrow C_{TF}.c.l.t \\
&\qquad\qquad\quad [\!]\ p \neq \textsf{F} \longrightarrow \uparrow C_{TS}.c.l.t \\
&\qquad\qquad\quad] \\
&\quad [\!]\ s \neq \textsf{T} \longrightarrow l := \min(l, l-c) \\
&\qquad\qquad ;\ [\ p = \textsf{F} \longrightarrow \uparrow C_{LF}.c.t.l \\
&\qquad\qquad\quad [\!]\ p \neq \textsf{F} \longrightarrow \uparrow C_{LS}.c.t.l \\
&\qquad\qquad\quad] \\
&\quad] \\
&]\!]
\end{aligned}$$

The argument p denotes the current phase of the tank controller (S for start-phase, R for rest-phase, and F for filling-phase). The state that caused the state transition is denoted by s, where the value T represents a temperature change and L a change of the level. The values t and l represent the 'old' state of the system, before the state transition occurred. If the state, denoted by the parameter s increases, the parameter c has the value 1. Otherwise, it has the value -1. For example, if the temperature decreases from state $\langle 1,0\rangle$ to $\langle 0,0\rangle$ during the start-phase, the following expression delivers the new control action for this state transition:

$$\mathrm{ctrl}(\mathsf{S},\mathsf{T},1,0,-1)$$

Domain observers

A domain observer has a continuous input and a discrete output. The input is transformed into a range index. The range boundaries are given in the parameter B. An output is generated only when an index change occurs. An output -1 denotes a decrease of the index, and a value $+1$ denotes an increase. The range index lies between 0 and n, where index i denotes an input value less than boundary value $B.i$, and index n denotes values greater than $B.(n-1)$. The observer is initialised with the index k in the following specifications:

$$
\begin{array}{l}
\mathsf{proc}\ S_T(a:{\downarrow}\mathrm{temp},\ b:!\,\mathrm{int},\ k,n:\mathrm{nat},\ B:\langle\mathrm{real}^n\rangle) = \\
[\![\ x:\mathrm{temp},\ i:\mathrm{nat} \\
|\ a \multimap x \\
|\ i := k \\
;*[\ i>0;\nabla x \leqslant B.(i-1) \longrightarrow i := i-1\ ;b!-1 \\
\quad [\!]\ i<n;\nabla x \geqslant B.i \quad\ \longrightarrow i := i+1\ ;b!+1 \\
\quad] \\
]\!]
\end{array}
$$

$$
\begin{array}{l}
\mathsf{proc}\ S_L(a:{\downarrow}\mathrm{level},\ b:!\,\mathrm{int},\ k,n:\mathrm{nat},\ B:\langle\mathrm{real}^n\rangle) = \\
[\![\ x:\mathrm{level},\ i:\mathrm{nat} \\
|\ a \multimap x \\
|\ i := k \\
;*[\ i>0;\nabla x \leqslant B.(i-1) \longrightarrow i := i-1\ ;b!-1 \\
\quad [\!]\ i<n;\nabla x \geqslant B.i \quad\ \longrightarrow i := i+1\ ;b!+1 \\
\quad] \\
]\!]
\end{array}
$$

In our example, we have two input values: temperature and level of the water in the tank. Therefore, we define a domain observer for each type of input variable. They differ only in the type of the input.

The water tank controller

In the previous section, we have presented the control concepts for the tank controller. We can now complete the specification of the control system by describing the three control processes, mentioned in Section 7.2.

Supervisory controller

The supervisory controller C_M activates and deactivates the two subcontrollers and determines the phase in which the tank controller operates.

$$
\begin{array}{l}
\mathsf{proc}\ C_M(q_s, q_f, q_r, q_b : \sim \mathrm{void}) = \\
[\![\ q_s\sim\ ;\ q_s\sim\ ;\ q_b\sim \\
\ ;\ *[\ q_f\sim\ ;\ q_b\sim\ ;\ q_r\sim\ ;\ q_b\sim\ ] \\
]\!]
\end{array}
$$

A synchronisation on channel q_s initiates the start-phase. When both level and temperature are at their respective set-points, a second synchronisation on channel q_s ends the start-phase. The filling of containers is initiated by a signal on channel q_b, activating the filling controller C_B. While filling the containers, the tank controller is put in its filling-phase operation by a signal on q_f. After completion of a batch of containers, the filling controller C_B is deactivated by a signal on channel q_b. Now the supervisory controller ends the filling-phase and starts the rest-phase of the tank controller C_T by synchronising on channel q_r. The rest-phase ends with a signal on channel q_b, which starts the filling of containers again. The supervisory controller alternates between filling-phase and rest-phase. Note, that in the filling-phase, both the filling controller C_B and the tank controller C_T are active, while in the rest-phase, only the tank controller is active.

Filling controller

The filling controller handles the filling of the containers. The containers are supplied in a batch, consisting of n containers. It takes τ_{fill} seconds to fill a container. If a container has been filled, it is replaced by the next container from the batch. It takes τ_{tr} seconds to replace the filled container by an empty one. Although, during this period, no containers are being filled, the supervisory controller remains in the filling-phase. After all containers of a batch have been filled, a new batch must be supplied. The exchange of batches takes τ_{con} seconds. During this period, the supervisory controller is in its rest-phase. The filling controller is represented by:

$$
\begin{array}{l}
\mathsf{proc}\ C_B(q : {\sim}\,\mathrm{void},\ v : !\,\mathrm{int},\ n : \mathrm{nat}) = \\
[\![\ i : \mathrm{nat} \\
\mid v\,!\,{-1} \\
;\ *[\ q{\sim} \\
\quad ;\ i := n \\
\quad ;\ *[\ i > 0 \longrightarrow v\,!\,{+1};\ \Delta\tau_{full};\ v\,!\,{-1};\ \Delta\tau_{tr};\ i := i-1\ ] \\
\quad ;\ q{\sim} \\
\quad ;\ \Delta\tau_{con} \\
\quad] \\
]\!]
\end{array}
$$

Initially, the valve is closed ($v\,!\,{-1}$). Then, the controller makes itself active ($q{\sim}$) and starts filling n containers. When the containers are filled, the controller deactivates ($q{\sim}$) and a new batch is supplied ($\Delta\tau_{con}$).

Observe that we did not define a data structure to model the containers. The objectives for this example are to design a control system for the water tank system and it is therefore sufficient to model the various activities by delay statements.

Tank controller

The tank controller has three modes of operation, one for each phase. The current mode of the controller is recorded in the variable m. It is set to S, denoting the start-phase, when a synchronisation on channel q_s occurs. The initial control values are assigned to y and sent to the valves.

$$
\begin{array}{l}
\mathsf{proc}\ C_T(q_s, q_f, q_r : {\sim}\,\mathrm{void},\ t, l : ?\,\mathrm{int},\ v_h, v_c, v_d : !\,\mathrm{int},\ T_s, L_s : \mathrm{int}) = \\
[\![\ y : \langle \mathrm{int}^3 \rangle,\ m : \mathrm{char},\ T, L : \mathrm{nat},\ c : \mathrm{int} \\
\mid q_s{\sim} \\
;\ m := \mathsf{S} \\
;\ y := \langle 1, -1, -1 \rangle \\
;\ v_h\,!\,y.0;\ v_c\,!\,y.1;\ v_d\,!\,y.2 \\
;\ *[\qquad\qquad\qquad\qquad\qquad\qquad t\,?\,c \longrightarrow y := \mathrm{map}(\mathrm{ctrl}(m, \mathsf{T}, T, L, c)) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad ;\ v_h\,!\,y.0;\ v_c\,!\,y.1;\ v_d\,!\,y.2 \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad ;\ T := T + c \\
\quad [\!] \qquad\qquad\qquad\qquad\qquad\qquad l\,?\,c \longrightarrow y := \mathrm{map}(\mathrm{ctrl}(m, \mathsf{L}, T, L, c)) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad ;\ v_h\,!\,y.0;\ v_c\,!\,y.1;\ v_d\,!\,y.2 \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad ;\ L := L + c \\
\quad [\!]\ m = \mathsf{S} \wedge T = T_s \wedge L = L_s ;\ q_s{\sim} \longrightarrow m := \mathsf{F} \\
\quad [\!]\ m = \mathsf{F} \qquad\qquad\qquad\qquad ;\ q_r{\sim} \longrightarrow m := \mathsf{R} \\
\quad [\!]\ m = \mathsf{R} \qquad\qquad\qquad\qquad ;\ q_f{\sim} \longrightarrow m := \mathsf{F} \\
\quad] \\
]\!]
\end{array}
$$

State transitions are received through the channels t and l. A new set of control values is computed by the function ctrl and mapped on the current values with the function map:

$$
\begin{array}{l}
\mathsf{func}\ \mathrm{map} : (x, y : \langle \mathrm{int}^3 \rangle) \to \langle \mathrm{int}^3 \rangle = \\
[\![\ z : \langle \mathrm{int}^3 \rangle,\ i : \mathrm{nat} \\
|\ i := 0 \\
;\ *[\ i < 3 \longrightarrow [\ x.i = 0 \longrightarrow z.i := y.i \\
\qquad\qquad\qquad\quad [\!]\ x.i \neq 0 \longrightarrow z.i := x.i \\
\qquad\qquad\qquad\quad] \\
\qquad\qquad\quad ;\ i := i + 1 \\
\quad] \\
;\ \uparrow z \\
]\!]
\end{array}
$$

This function changes the current values to the new values, unless a new value is equal to 0. In that case the current value is preserved.

The mode changes of the tank controller are accomplished by synchronising on the channels q_s, q_f, and q_r. Note that we only switch from start-phase to filling-phase if the current state is the desired state, denoted by the parameters T_s and L_s.

7.3 Summary

In this chapter, we have demonstrated how the formalism χ can be used in the design process of a control system for a hot water tank. We have made a model of the physical system that represents the relevant aspects, needed to develop the control system. It contains the behaviour of the water tank and the valves that can be influenced by the control system. Next, the processes that model the relevant aspects of the environment of the system are included in the model. These relevant aspects are the supply and drain of water. We have not included a model of the containers and their movements because they are not assumed relevant for the control system of the water tank.

The design of the control system starts with the definition of the interactions with the controlled system. Furthermore, the structure of the control system is defined. The behaviours of the processes in the control system depend on the chosen control concepts. We have presented state-transition control as a means to control the level and temperature of the water in the tank. A data structure has been developed that can be used in the specification of a state-transition controller. Also, the functions to manipulate these data structures have been defined.

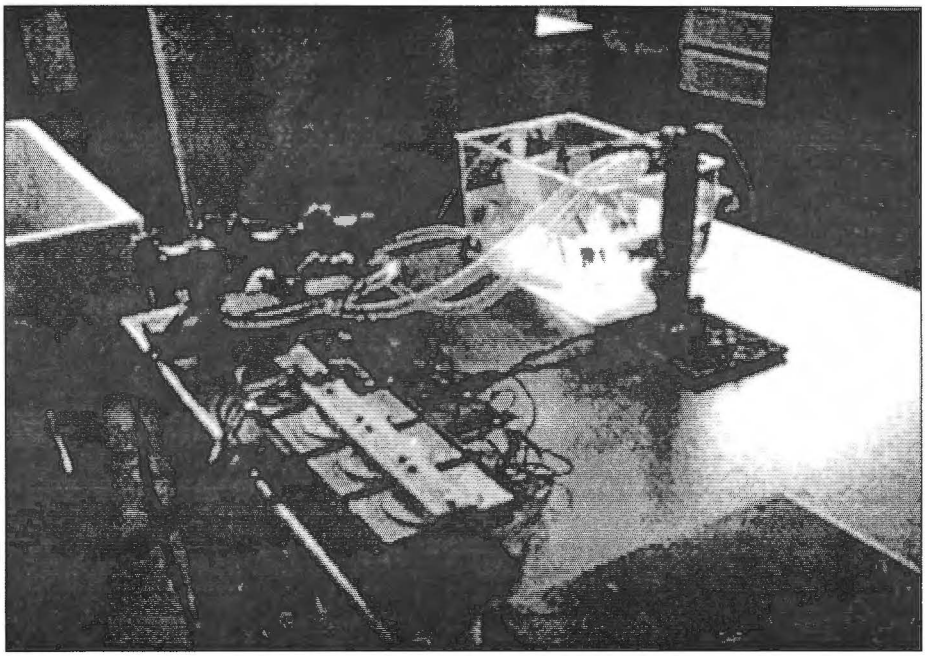


Figure 7.7: Test set-up of the water tank [Kam95].

Finally, the control processes are specified, based on the chosen control concepts. At this point, we have yet no means to test the developed control system. However, van de Kamp and Voorbraak [Kam95] have built a preliminary χ simulator that is used to evaluate the control system design. For this purpose, a test set-up (Figure 7.7) was made to test the controllers in a real-life environment. We refer to this work for detailed information about the performance of the state-transition controller for the water tank system.

Chapter 8

Conclusion

The ever increasing complexity of industrial systems demands for more sophisticated techniques to support the design process of these systems. In this dissertation, we present the formalism χ for the specification of the dynamical behaviour of industrial systems. The formalism is based on the process-interaction approach in which a system is viewed as a collection of concurrently operating and interacting components, called processes. Each process models the behaviour of a system component. The formalism χ provides means to specify this behaviour as a continuous or as a discrete behaviour. Moreover, the behaviour can be specified as a combination of continuous and discrete behaviour.

Hierarchical models are achieved by grouping processes into a system, which in its turn can act as a process. A system can be grouped with other processes and systems to form a new system. This hierarchy enables the decomposition of a system into smaller subsystems.

Interactions between processes are defined by channels. Different channels are defined for different kind of interactions. In this way, the specification of a process can be made independent from other processes in the model. The channels define an interface to the environment of the process, thus, allowing the process' behaviour to be specified without any knowledge of the behaviour of other processes. This modularity is one of the requirements for the specification formalism.

In our approach to the development of the formalism, we consider continuous and discrete behaviours as equally important. The interactions between continuous and discrete behaviours are specified within a process and not in channels. Only two language constructs are defined to specify these interactions: state-events and guarded equations. This approach has led to a formalism that allows an easy and straightforward specification of mixed continuous and discrete systems. We consider this one of the major advantages of the formalism χ.

The language, used by the formalism to represent models, is based on the guarded command language for the discrete behaviour descriptions and on differential and algebraic equations for the continuous behaviour descriptions. The language consists of a symbol set and the definition of the semantics. The language, used for the representation of a model, influences our perception of the system that we model. If a modelling language is only capable of representing sequential activities, a modelling engineer, thinking in terms of this language, will not perceive any existing parallelism in the system. It is therefore necessary that a language is suitable for the application area in which it is used, otherwise potentially important information will be lost. Furthermore, in designing a language, it is important to define enough symbols to be able to represent what needs to be represented. On the other hand, too many symbols make the language hard to learn and use. The best way to resolve this dilemma is to let the language evolve to fit to its application area.

The formalism χ is the next generation in a series of specification languages and tools for industrial systems. The evolution process started with SOLE [Roo82a] and proceeded with D86 [Mun86], S84 [Roo84] and ROSKIT [Ros85] to Process-Tool [Wor90]. This evolution process does not stop with χ. Using the formalism will reveal new insights and defects that will lead to improved formalisms in the future. Nevertheless, there is a difference between χ and its predecessors. The formalism χ is not developed as a simulation tool and is thus independent of a programming language. It is meant to initiate the development of a calculus, dedicated to the design of industrial systems and their control systems.

We recommend that future research will be carried out in two directions. From the use of the predecessors of χ, we can conclude that a supporting simulation tool is indispensable to the application of the formalism. Recently, a simulation tool has been developed that supports the simulation of discrete behaviour models [Nau95]. This project will be continued and must lead to the implementation of a simulation tool that supports both discrete and continuous behaviours. Eventually, making the specifications executable, allows the specification of control systems to be used as the real-time control system in the real system.

In the development of a simulation tool, special attention must be paid to the specification of experiments that are performed on models. A formal specification of an experiment serves as a documentation and discussion object, and enables the repeatability of an experiment. Anticipating these developments, we have not included a means to initialise the state of a process. Initialisation is part of the initial value problem associated with performing simulations. The initial state of a system should therefore be defined in the experiment specification.

More research should be carried out to develop our formalism into a calculus. A calculus provides means to reason about a specification and to prove properties of the specification. For example, it would be useful to prove that a specification is free of deadlock, or to prove that certain states of a system cannot

be reached [Bae90]. Some research has already been done on this subject by Verreijken [Ver95].

A specification formalism, like χ, is a tool to assist the systems engineer in designing industrial systems. It formalises the way in which models are represented. It does not state how to build a model. Nevertheless, a formalism influences our perception of systems. The use of the formalism χ changes the way in which we think about industrial systems. Ultimately, this should lead to better designs.

Bibliography

[Ake78]
J.E. van Aken,
On the Control of Industrial Organizations.
Dissertation. Martinus Nijhoff Social Sciences Division, Leiden, 1978.

[And90]
M. Andersson,
Omola - An Object-Oriented Language for Model Representation.
Thesis. Lund Institute of Technology, Lund, 1990.

[Are94a]
N.W.A. Arends, J.M. van de Mortel-Fronczak and J.E. Rooda,
Continuous Systems Specification Language.
In: *CISS - First Joint Conference of International Simulation Societies Proceedings.* Society for Computer Simulation, Zürich, 1994, (pp. 76–79).

[Are94b]
N.W.A. Arends, J.M. van de Mortel-Fronczak and J.E. Rooda,
Specification Language for Continuous Systems .
In: *Applied Modelling and Simulation.* Proceedings of the IASTED International Conference, Lugano, 1994.

[Bee95]
D.A. van Beek, S.H.F. Gordijn and J.E. Rooda,
Integrating Continuous-time and Discrete-event Concepts in Process Modelling, Simulation and Control.
In: *Proc. of the First World Conference on Integrated Design and Process Technology*, Society for Design and Process Science, 1995, (pp. 197–204).

[Bae90]
J.C.M. Baeten and J.A. Bergstra,
Process Algebra.
Cambridge Tracts in Theoretical Computer Science 18, Cambridge University Press, 1990.

[Ber68]
L. von Bertalanffy,
General System Theory.
George Braziller Inc., New York, 1968. (Revised edition.)

[Bur93]
A. Burns and G. Davies,
Concurrent Programming.
International Computer Science Series. Addison-Wesley, Amsterdam, 1993.

[Bra93]
L.E.M.W. Brandts,
Design of Industrial Systems.
Dissertation. Eindhoven University of Technology, Eindhoven, 1993.

[Coh86]
B. Cohen, W.T. Harwood, M.I. Jackson,
The Specification of Complex Systems.
Addison-Wesley, Amsterdam, 1986.

[CSS67]
The SCi Continuous System Simulation Language (CSSL).
In: *Simulation*, **9**(6), 1967, (pp. 281–303).

[Dij74]
E.W. Dijkstra,
Guarded Commands, Nondeterminacy and a Calculus for the Derivation of Programs.
Nederlands Rekenmachine Genootschap, 1974.

[Dub]
W. Beitz and K.H. Küttner,
Dubbel. Taschenbuch für die Machinenbau. 14th revised edition.
Springer-Verlag, Berlin, 1981. (In German.)

[Elm78]
H. Elmqvist,
A Structured Model Language for Large Continuous Systems.
Dissertation. Lund Institute of Technology, Lund, 1978.

[Gai79]
B. Gaines,
General Systems Research: Quo Vadis.
In: *General Systems Yearbook*, **24**, 1979, (pp. 1–9).

[Gof91]
M.P. Jones,
An Introduction to GOFER.
1991.

[Gol83]
A. Goldberg and D. Robson,
Smalltalk-80: The Language and its Implementation.
Addison-Wesley, Massachusetts, 1983.

[Hoa85]
C.A.R. Hoare,
Communicating Sequential Processes.
Prentice Hall, Englewood-Cliffs, New York, 1985.

[Hoo91]
J. Hooman,
Specification and Compositional Verification of Real-Time Systems.
Springer-Verlag, Berlin, 1991.

[Kam95]
G.F.J.W. van de Kamp and E.M.M. Voorbraak,
Control of Hybrid Industrial Systems. A Case: The Liquid Tank.
Postgraduate Thesis. Stan Ackermans Instituut, Eindhoven University of Technology, Eindhoven, 1995.

[Lee74]
A.C.J. de Leeuw,
Systeemleer en organisatiekunde. Een onderzoek naar de mogelijke bijdragen van de systeemleer tot een integrale organisatiekunde.
Stenfert Kroese, Leiden, 1974. (In Dutch.)

[Mel86]
S.J. Mellor and P.T. Ward,
Structured Development for Real-Time Systems.
Prentice-Hall, Inc., Englewood Cliffs, NJ, 1986.

[Min65]
M. Minsky,
Models, Minds, Machines.
In: *Proceedings IFIP congres New York 1965*, MacMillan, London, 1965, (pp. 45–49).

[Mor95a]
J.M. vand de Mortel-Fronczak,
Application of Concurrent Programming to Specification of Industrial Systems.
In: *Proceedings of the 1995 IFAC Symposium on Information Control Problems in Manufacturing*, Beijng, 1995, (pp. 421–426).

[Mor95b]
J.M. van de Mortel-Fronczak,
Operational Semantics of χ.
Internal Report WPA 420062. Eindhoven University of Technology, Eindhoven, 1994.

[Mor95c]
J.M. van de Mortel-Fronczak, J.E. Rooda and N.J.M. van den Nieuwelaar,
Specification of a Flexible Manufacturing System Using Concurrent Programming.
In: *Concurrent Engineering: Research and Applications*, **3**(3), 1995, (pp. 187–192).

[Mun86]
B. Munneke and R. Overwater,
Design '86.
Manual. Eindhoven University of Technology, Eindhoven, 1986.

[Nan81]
R.E. Nance,
The Time and State Relationships in Simulation Modelling.
In: *Comm. Ass. Comp. Mach.*, **24**(4), 1981, (pp. 173–179).

[Nau95]
G. Naumoski and W.T.M. Alberts,
The χ Engine: A Fast Simulator for Systems Engineering.
Postgraduate Thesis. Stan Ackermans Instituut, Eindhoven University of Technology, Eindhoven, 1995.

[Ove87]
R. Overwater,
Processes and Interactions. An Approach to the Modelling of Industrial Systems.
Dissertation. Eindhoven University of Technology, Eindhoven, 1987.

[Pet62]
C. Petri,
Kommunikation mit Automaten.
Dissertation. University of Bonn, Bonn, 1962. (In German.)

[Pol89]
L.F. Pollacia,
A Survey of Discrete Event Simulation and State-of-the-art Discrete Event Languages.
In: *Sigsim Simulation Digest*, **20**(3), 1989, (pp. 8–25).

[Pre93]
H.A. Preisig,
More on the Synthesis of a Supervisory Controller from First Priciples.
In: *Proceeding of the IFAC World Congress*, Sydney, Australia, 1993.

[Roo82a]
J.E. Rooda,
Simulation of Logistic Elements (SOLE).
Report. Twente University of Technology, Enschede, 1982.

[Roo82b]
J.E. Rooda,
Transport- en produktiesystemen, modelbouw en simulatie.
In: *Transport + Opslag*, **6**(8), 1982, (pp. 30–35). (In Dutch.)

[Roo83]
J.E. Rooda and J.H.A. Arentsen,
Een structuurmodel voor de beschrijving van transport- en produktiesystemen.
In: *Transport + Opslag*, **7**(10), 1983, (pp. 88–90). (In Dutch.)

[Roo84]
J.E. Rooda, S.M.M. Joosten, T.J. Rossingh and R. Smedinga,
Simulation in S84.
Manual. Twente University of Technology, Enschede, 1984.

[Roo91]
J.E. Rooda,
Procescalculus: nieuw instrument beschrijft industriële systemen.
In: *I^2Werktuigbouwkunde*, **5**, 1991, (pp. 13–15). (In Dutch.)

[Roo95]
J.E. Rooda,
The Modelling of Industrial Systems.
Syllabus (4746). Eindhoven University of Technology, Eindhoven, 1995

[Ros85]
T.J. Rossingh and J.E. Rooda,
Real-time Operating System Kit (ROSKIT).
Manual. Twente University of Technology, Enschede, 1985.

[Tan94]
O. Tanir and S. Sevinc,
Defining Requirements for a Standard Simulation Environment.
In: *IEEE Computer*, **27**(2), 1994, (pp. 28–34).

[Vel92]
J. In 't Veld,
Analyse van organisatieproblemen. Een toepassing van het denken in systemen en processen.
Stenfert Kroese Uitgevers, Leiden, 1992. (In Dutch, Sixth edition.)

[Ver95]
J.J. Verreijken,
A Process Algebra for Hybrid Systems.
Eindhoven University of Technology, 1995.

[Web86]
Webster's Third International Dictionary.
Merriam–Webster Inc., Springfield, 1986.

[Web90]
The New Lexicon Webster's Dictionary of the English Language.
Lexicon Publications Inc., New York, 1990.

[Wor90]
A.M. Wortmann and J.E. Rooda,
The Process-Interaction Environment User Manual.
Manual (WPA0841). Eindhoven University of Technology, Eindhoven, 1990.

[Wor91]
A.M. Wortmann,
Modelling and Simulation of Industrial Systems.
Dissertation. Eindhoven University of Technology, Eindhoven, 1991.

Appendix A

The Syntax

In this appendix, the syntax rules of the specification language are described, using the BNF notation. All non-terminals that are define in this appendix are slanted and capitalised. Other non-terminals, printed in lower-case letters, are not defined by syntax rules. Table A.1 shows the informal meaning of these non-terminals.

name	definition
id	*Identifier*: a sequence of letters and digits, starting with a letter.
ids	*Identifiers*: a sequence of identifiers, separated by commas.
e	*Expression*: a mathematical expression that may evaluate to any type.
b	*Boolean expression*: a mathematical expression that evaluates either to true or false.
nat	*Natural expression*: a mathematical expression that evaluates to a Natural number.
int	*Integer expression*: a mathematical expression that evaluates to an Integer number.
c	*channel identifier*: an identifier that represents a channel.

Table A.1: Informal definition of non-terminals.

A.1 Global declarations

χ ::= *TDEC* | *CDEC* | *UDEC* | *FDEC* | *PDEC* | *SDEC*

Type declarations

TDEC ::= type *TDEF*
TDEF ::= *ids* = *TYPE* | *TDEF, TDEF*

TYPE ::= *PTYP* | *LTYP* | *TTYP* | *FTYP*

PTYP ::= bool | nat | int | real | char | string | dist | void | *id* | [*UNIT*]

LTYP ::= *TYPE**

TTYP ::= ⟨⟩ | ⟨*PTUP*⟩
PTUP ::= *TYPE* | *TYPE*^{nat} | *PTUP* × *PTUP*

FTYP ::= ( ) → *TYPE* | (*ATYP*) → *TYPE*
ATYP ::= *TYPE* | *TYPE, TYPE*

Constant declarations

CDEC ::= const *CDEF*
CDEF ::= *ids* = *e* | *CDEF, CDEF*

Unit declarations

UDEC ::= unit *UDEF*
UDEF ::= *ids* = *UNIT* | *UDEF, UDEF*

UNIT ::= *PUNI* | *PUNI*^{int} | *UNIT*·*UNIT*
PUNI ::= m | kg | s | A | K | mol | cd | − | *id*

A.2 Functions

FDEC ::= func *FDEF* = ⟦ *FBOD* ⟧
FDEF ::= *id*( ) → *TYPE* | *id*(*ARGS*) → *TYPE*
ARGS ::= *ids* : *TYPE* | *ARGS, ARGS*
FBOD ::= *VARS* | *VARS* '|' *FSTM* | *FSTM*

$$
\begin{array}{lcl}
\mathit{FSTM} & ::= & \mathit{STMT} \mid \mathit{FSTM};\ \mathit{FSTM} \mid \uparrow e \mid [\,\mathit{FSEL}\,] \mid *[\,\mathit{FSEL}\,] \\
\mathit{FSEL} & ::= & b \longrightarrow \mathit{FSTM} \mid \mathit{FSEL} \,[]\, \mathit{FSEL}
\end{array}
$$

A.3 Processes

$$
\begin{array}{lcl}
\mathit{PDEC} & ::= & \mathsf{proc}\ \mathit{PDEF} = [\![\ \mathit{PBOD}\ ]\!] \\
\mathit{PDEF} & ::= & \mathit{id} \mid \mathit{id}(\mathit{PARS}) \\
 & & \\
\mathit{PARS} & ::= & \mathit{ids} : \mathit{PART} \mid \mathit{PARS}, \mathit{PARS} \\
\mathit{PART} & ::= & {\sim}\ \mathit{CTYP} \mid ?\ \mathit{CTYP} \mid !\ \mathit{CTYP} \\
 & \mid & \updownarrow \mathit{CTYP} \mid \downarrow \mathit{CTYP} \mid \uparrow \mathit{CTYP} \\
 & \mid & \mathit{TYPE} \\
\mathit{CTYP} & ::= & \mathit{TYPE} \mid (\mathit{BTYP}) \\
\mathit{BTYP} & ::= & \mathit{CTYP} \mid \mathit{CTYP}^{nat} \\
 & & \\
\mathit{PBOD} & ::= & \mathit{VARS} \\
 & \mid & \mathit{VARS}\ \text{`}|\text{'}\ \mathit{LINK} \\
 & \mid & \mathit{VARS}\ \text{`}|\text{'}\ \mathit{LINK}\ \text{`}|\text{'}\ \mathit{EQTN} \\
 & \mid & \mathit{VARS}\ \text{`}|\text{'}\ \mathit{EQTN} \\
 & \mid & \mathit{VARS}\ \text{`}|\text{'}\ \mathit{LINK}\ \text{`}|\text{'}\ \mathit{EQTN}\text{`}|\text{'}\ \mathit{PSTM} \\
 & \mid & \mathit{VARS}\ \text{`}|\text{'}\ \mathit{EQTN}\ \text{`}|\text{'}\ \mathit{PSTM} \\
 & \mid & \mathit{VARS}\ \text{`}|\text{'}\ \mathit{LINK}\ \text{`}|\text{'}\ \mathit{PSTM} \\
 & \mid & \mathit{VARS}\ \text{`}|\text{'}\ \mathit{PSTM} \\
 & \mid & \mathit{PSTM} \\
 & & \\
\mathit{VARS} & ::= & \mathit{ids} : \mathit{TYPE} \mid \mathit{VARS}, \mathit{VARS} \\
 & & \\
\mathit{LINK} & ::= & \mathit{ids} \multimap \mathit{LEXP} \mid \mathit{LINK}, \mathit{LINK} \\
\mathit{LEXP} & ::= & \mathit{id} \mid \langle \mathit{LIDS} \rangle \\
\mathit{LIDS} & ::= & \mathit{LEXP} \mid \mathit{LEXP}, \mathit{LEXP} \\
 & & \\
\mathit{EQTN} & ::= & e = e \mid \mathit{EQNS} \mid [\ \mathit{GDEQ}\ ] \\
\mathit{EQNS} & ::= & \mathit{EQTN}, \mathit{EQTN} \\
\mathit{GDEQ} & ::= & b \longrightarrow \mathit{EQNS} \mid \mathit{GDEQ} \,[]\, \mathit{GDEQ} \\
 & & \\
\mathit{PSTM} & ::= & \mathit{STMT} \mid \mathit{PSTM};\ \mathit{PSTM} \mid \mathit{ESTM} \\
 & \mid & [\ \mathit{PSEL}\ ] \mid *[\ \mathit{PSEL}\ ] \\
 & \mid & [\ \mathit{PSLW}\ ] \mid *[\ \mathit{PSLW}\ ] \\
 & & \\
\mathit{STMT} & ::= & \mathsf{skip} \mid x := e \\
 & & \\
\mathit{ESTM} & ::= & \Delta e \mid c{\sim} \mid c?x \mid c!e \mid \nabla b
\end{array}
$$

$$PSEL \;::=\; b \longrightarrow PSTM \;\mid PSEL \;[\!]\; PSEL$$

$$\begin{aligned} PSLW \;::=\;& ESTM \\ \mid\;& ESTM \longrightarrow PSTM \\ \mid\;& b;\ ESTM \\ \mid\;& b;\ ESTM \longrightarrow PSTM \\ \mid\;& PSLW \;[\!]\; PSLW \end{aligned}$$

A.4 Systems

$$\begin{aligned} SDEC \;&::=\; \textsf{syst}\ SDEF = [\![\ SBOD\ ]\!] \\ SDEF \;&::=\; id \;\mid id(PARS) \end{aligned}$$

$$\begin{aligned} SBOD \;::=\;& CHAN \\ \mid\;& CHAN\ \text{`}|\text{'}\ PROC \end{aligned}$$

$$CHAN \;::=\; ids : CTYP \;\mid CHAN,\ CHAN$$

$$PROC \;::=\; id \;\mid id(APAR) \;\mid PROC \;[\!]\; PROC$$

$$APAR \;::=\; e \;\mid APAR,\ APAR$$

Appendix B

Statistical Distributions

Statistical distributions allow the modelling of stochastic discrete behaviour in processes. A number of such distributions are predefined in χ. We distinguish between discrete and continuous distributions. A sample from a discrete distribution is a fixed number, usually a Natural or an Integer. Samples from continuous distributions are continuously distributed on the range of the distribution. They are normally of type real.

The remainder of this appendix presents the predefined distributions of χ. For each distribution, a number of characteristics are listed in a table. First, the function, to create a distribution, is presented in the upper-right corner of the table. The domains of the arguments of this function are defined, as well as the domain of samples (x), taken from the distribution. Finally, the mean (μ), the variance (σ^2), and the density function ($f(x)$) are given for each type of distribution.

B.1 Discrete distributions

Uniform			dun(a, b)
$a \in \{-\infty, \ldots, \infty\}$		$x \in \{a, \ldots, b\}$	
$b \in \{-\infty, \ldots, \infty\}$			
$\mu = \frac{a+b}{2}$		$\sigma^2 = \frac{(b-a+1)^2 - 1}{12}$	
$f(x) = \frac{1}{b-a+1}$			

A uniform distribution is used as a first model for a quantity that is felt to be randomly varying between a and b, about which little else is known.

Bernouilli		ber(p)
$p \in [0,1]$	$x \in \{0,1\}$	
$\mu = p$	$\sigma^2 = p(1-p)$	
$f(x) = \begin{cases} 1-p & : \ x=0 \\ p & : \ x=1 \end{cases}$		

A Bernoulli random variable can be thought of as the outcome of an experiment that either fails or succeeds. The probability parameter p is the probability of success. Such an experiment is called a Bernoulli trial and provides a convenient way of relating several other discrete distributions to the Bernoulli distribution.

Binomial		bin(p,n)
$p \in [0,1]$	$x \in \{0,\ldots,n\}$	
$n \in \{0,\ldots,\infty\}$		
$\mu = np$	$\sigma^2 = np(1-p)$	
$f(x) = \binom{n}{x} p^x (1-p)^{n-x}$		

The Binomial distribution defines the number of successes in n independent Bernoulli trials with probability parameter p of success on each trial. A possible application for the Binomial distribution is to denote the number of defective items in a batch of size n.

Geometrical		geo(p)
$p \in [0,1]$	$x \in \{0,\ldots,\infty\}$	
$\mu = \dfrac{1-p}{p}$	$\sigma^2 = \dfrac{1-p}{p^2}$	
$f(x) = p(1-p)^x$		

A Geometrical distribution defines the number of failures before the first success in a sequence of independent Bernoulli trials with probability p of success on each trial. It can be used to denote the number of items inspected before encountering the first defective item.

Poisson		poi(p)
$p \in [0, \infty)$	$x \in \{0, \ldots, \infty\}$	
$\mu = p$	$\sigma^2 = p$	
$f(x) = \frac{e^{-p}p^x}{x!}$		

The Poisson distribution defines the number of events that occur in an interval of time when the events occur independently of each other. An example application is to denote the number of items in a batch of random size.

B.2 Continuous distributions

Uniform		cun(a, b)
$a \in (-\infty, \infty)$	$x \in [a, b]$	
$b \in (-\infty, \infty)$		
$\mu = \frac{a+b}{2}$	$\sigma^2 = \frac{(a+b)^2}{12}$	
$f(x) = \frac{1}{b-a}$		

As with the discrete Uniform distribution, it is used for a randomly varying quantity whose value lies between a and b.

Negative Exponential		nex(a)
$a \in (0, \infty)$	$x \in [0, \infty)$	
$\mu = \frac{1}{a}$	$\sigma^2 = \frac{1}{a^2}$	
$f(x) = ae^{-ax}$		

The Negative Exponential distribution can be used for the definition of the time between independent events, for instance, arrivals at a service facility, that occur at a constant rate (inter arrival times). Another example of its use is the definition of life-times of devices with constant hazard rate.

Erlang		erl(k, a)
$k \in \{1, \infty\}$	$x \in [0, \infty)$	
$a \in (0, \infty)$		
$\mu = \frac{k}{a}$	$\sigma^2 = \frac{k}{a^2}$	
$f(x) = \frac{ae^{-ax}(ax)^{k-1}}{(k-1)!}$		

The Erlang distribution represents a time to complete some task, e.g. customer service, machine repair or operation time. It consists of the product of k subsequent samples from an exponential distribution.

Gamma		gam(p, a)
$p \in (0, \infty)$	$x \in [0, \infty)$	
$a \in (0, \infty)$		
$\mu = \frac{p}{a}$	$\sigma^2 = \frac{p}{a^2}$	
$f(x) = \frac{ae^{-ax}(ax)^{p-1}}{\Gamma(p)}$		

The Gamma distribution is similar to the Erlang distribution, except that the parameter p is not restricted to a whole number. Its application is the same as that of an Erlang distribution.

Normal		nor(m, s)
$m \in (-\infty, \infty)$	$x \in (-\infty, \infty)$	
$s \in (0, \infty)$		
$\mu = m$	$\sigma^2 = s^2$	
$f(x) = \frac{e^{-\frac{1}{2}\left(\frac{x-m}{s}\right)}}{s\sqrt{2\pi}}$		

The Normal distribution is used to represent quantities, for instance, measurement errors, that are sums of a large number of other quantities.

Weibull		wei(c, a)
$c \in (0, \infty)$	$x \in [0, \infty)$	
$a \in (0, \infty)$		
$\mu = \frac{1}{a}\Gamma\left(\frac{c+1}{c}\right)$	$\sigma^2 = \frac{1}{a^2}\left(\Gamma\left(\frac{c-2}{c}\right) - \Gamma\left(\frac{c+1}{c}\right)^2\right)$	
$f(x) = ac(ax)^{c-1}e^{-(ax)^c}$		

The Weibull distribution is used in reliability models for life-times of devices and is used to define a time to complete some task.

Index

Curriculum Vitae

Norbert Arends was born on the 13th of February 1964 in Eindhoven. After finishing his Atheneum B in 1982 at the St. Joris College in Eindhoven, he started his studies at the Eindhoven University of Technology, Department of Mechanical Engineering. During the final years of his study, he was involved in the modelling of wafer production plants. He graduated, in the section Systems Engineering, on a research project on the implementation of the simulation tool 'Process-Interaction Environment' in December 1989.

On January the 1st 1990, he started a research project on Man-Machine Interfaces at Eindhoven University of Technology, supervised by Prof.dr.ir. J.E. Rooda. Since May 1991, his research focused on the development of a specification formalism for the modelling of the dynamical behaviour of industrial systems.

Stellingen

behorende bij het proefschrift

A Systems Engineering Specification Formalism

1. Voor de specificatie van de interacties tussen discreet gedrag en continu gedrag is het voldoende te beschikken over state-events en guarded equations.

   Dit proefschrift

2. Een model van een systeem kan niet worden opgesteld zonder vooraf het doel of de doelen van het model te formuleren.

   Dit proefschrift

3. Een specificatie-taal moet een evolutie doormaken om een geschikte taal te worden.

   Dit proefschrift

4. Voor het uitvoeren van een experiment middels simulatie is het nodig een begintoestand te definiëren voor het te simuleren model. Daar de begintoestand betrekking heeft op het experiment dient initialisatie van variabelen van het model te geschieden in de experiment-beschrijving en niet in de model-beschrijving.

5. Abstractie is geen tekortkoming van een specificatietaal. Het maakt de taal juist geschikt voor specificaties.

6. Het opstellen van de specificatie van het dynamisch gedrag van een systeem leidt tot een beter ontwerp.

   Dit proefschrift

7. Een specificatie-formalisme verdient pas dan het predicaat hybride als het, naast een gecombineerd discreet en continu gedrag, ook zuiver discreet of zuiver continu gedrag kan beschrijven.

   Dit proefschrift

8. Het feit dat in de laatste jaren steeds meer wiskunde en informatica wordt toegepast bij het ontwerpen van industriële systemen wijst niet op een trend dat werktuigbouwkundigen van hun vak vervreemden maar juist dat de wetenschappelijke waarde van de werktuigbouwkunde toeneemt.

9. Het bestaan van vooroordelen over het beoefenen van de schietsport wordt veroorzaakt door personen die schieten niet als sport zien.

10. Uit de bewondering die bestaat over vaardigheden uit een ver verleden die worden aangeduid met een zinsnede als "Wat knap dat ze dat toen al konden.", kan geconcludeerd worden dat de ontwikkeling van deze vaardigheden sindsdien vooruit is gegaan. Deze conclusie is onjuist.

11. Als gevolg van de vertragende werking van onze zintuigen kunnen wij de grens tussen verleden en toekomst niet waarnemen. Daarmee leven wij in het verleden.

Norbert Arends Eindhoven, 17 juni 1996

The design process of an industrial system is a process of decision-making, where each decision has its impact on the resulting system. To substantiate design decisions, a formal specification is inevitable. In this research, we introduce the formalism χ as a means to specify the dynamic behaviour of industrial systems.

This dissertation describes the syntax and semantics of the formalism χ The application of the formalism is illustrated by examples of industrial systems, including their control systems. Furthermore, it gives a survey of the design decisions that resulted in the presented formalism. This research is meant to initiate the development of a calculus, dedicated to the analysis of the behaviour of industrial systems.

The picture on the cover is a so-called 3D-stereogram. It contains a threedimensional image of discrete and continuous objects. To experience the image, you must look at it out of focus, a few inches behind the picture. You can also start by keeping the picture at a one inch distance before your eyes and stare at it. Now slowly move the picture away without refocusing. Let your eyes adjust and the image should reveal itself. Just relax and let it happen...

ISBN 90-386-0028-3