

Copyright is owned by the Author of the thesis. Permission is given for a copy to be downloaded by an individual for the purpose of research and private study only. The thesis may not be reproduced elsewhere without the permission of the Author.

A Formal Framework For Data Flow Diagrams With Control Extensions

A dissertation presented
in partial fulfillment of the requirements
for the degree of Doctor of Philosophy
in Computer Science at Massey University

Robert Bertrand France

1989

Abstract

In this thesis a formal foundation for data flow diagrams (DFDs) with control extensions is developed. The DFD is the primary specification tool of the Structured Analysis (SA) approach to requirements analysis and specification.

In recent times, a number of extensions to DFDs, which enhance their use in the specification of behaviour of complex applications (i.e. applications with concurrent or real-time aspects), have been proposed. Such extensions tend to concentrate on increasing the descriptive power of DFDs, while paying less attention to providing the extended DFDs with a formal foundation. Such a foundation would facilitate the generation of formal specifications from DFDs, which could then be used to rigorously validate the DFDs and the behavioural properties they capture, and could also be used as the basis of formal verification activities where subsequent specifications are verified against the formal specifications generated from DFDs. Also, the simple, graphical nature of DFDs, supported by a formal foundation, facilitates their use in formal development strategies. Their use in this respect achieves a level of understandability not usually associated with formal specification tools.

The formal foundation introduced in this thesis consists of two parts: the Picture Level (PL) and the Specification Level (SL). The PL is an algebraic specification characterizing the syntactic aspects of DFDs. The specification is associated with an operational semantics which provides an effective means for investigating the syntactic properties of DFDs with the PL.

The SL consists of tools and techniques for describing control aspects of applications, and for formally specifying the data, functional, and control aspects of the control-extended DFDs. The control-extended DFDs are called Extended DFDs (ExtDFDs). An ExtDFD depicts the types of interactions that can take place between DFD components, as well as the events that affect the mode of operation of the application it models. A formal specification, called the Behavioural Specification (BS), is generated from an ExtDFD and supporting specifications characterizing the data objects and primitive processing components of the ExtDFD. The role of the BS in formal validation and verification activities is discussed in this thesis.

Acknowledgements

I would like to thank Thomas Docker for starting me on this research and for his support during the investigative parts of the research, as well as for his much appreciated efforts in creating an environment conducive to my research in New Zealand. I would also like to thank Professor Mark Apperley, my chief supervisor, for his support during the preparation of this thesis, especially in the latter stages, and to Dr. John Hudson, my second supervisor, for his valuable comments, and efforts in reviewing the more technical aspects of this thesis. The reviewing of this thesis has not been an easy task, given the volume of technical notation and detail, and my own failure to write in a clearer manner in some cases, and I am grateful to the above three persons for their efforts in this respect. Thanks also to the entire staff of the Department of Computer Science at Massey University for making study far from home bearable.

Contents

Chapter 0 : Introduction

0.1 The context	1
0.1.1 The requirements specification problem	1
0.1.2 Formal requirements specification	2
0.1.3 Thesis objectives	3
0.2 Formal specifications from data flow diagrams	4
0.3 Overview of thesis	5

Chapter 1: Data Flow-Orientated Requirements Specification Techniques

1.0 Introduction	7
1.1 Structured Analysis (SA) specification techniques.....	8
1.1.1 Data flow diagrams (DFDs)	8
1.1.2 The data dictionary and process specifications	14
1.1.3 SA and design	15
1.1.4 Limitations of SA specification tools and techniques	17
1.2 Extensions to SA	20
1.2.1 Yourdon's Structured Method (YSM)	20
1.2.2 Hatley's extensions	25
1.2.3 ADISSA	28
1.2.4 DARTS	31
1.2.5 Tse's extensions: Formal DFDs (FDFDs)	33
1.2.6 Extended DFDs (EXT-DFDs)	35
1.3 Conclusion	36

Chapter 2: Syntactic and Semantic Aspects of DFDs

2.0 Introduction	38
2.1 A computer-based library application	38
2.2 Syntactic aspects of DFDs	39
2.2.1 Syntactic aspects of flat DFDs	44
2.2.2 Syntactic aspects of hierarchies of DFDs	46
2.3 Semantics aspects of DFDs	53
2.3.1 Flattening hierarchies of DFDs	54

2.3.2	Describing the control aspects of applications	56
2.3.3	Semantics aspects of data flows and data stores	60
2.3.4	Semantic aspects of processes	64
2.3.5	Specifying the interactions in a DFD	67
2.4	Summary	71

Chapter 3: Positive-Negative Relational Specifications: An Algebraic Approach to Specification

3.0	Introduction	72
3.1	Positive-negative relational specifications (RSs)	73
3.1.1	Specifications and algebras	73
3.1.2	Hierarchical RSs	79
3.1.3	RS schemas	83
3.2	Model-theoretic interpretation of RSs	84
3.2.1	Equality and inequality assumptions	84
3.2.2	Negated relation assumptions	87
3.3	An operational semantics for RSs	87
3.3.1	Relational conditional term rewriting systems (R-CTRSs)	88
3.3.2	Sufficient conditions for termination and confluence of R-CTRSs	91
3.3.3	Correctness of R-CTRSs	94
3.4	Summary	95

Chapter 4: The Picture Level: Characterizing the syntactic aspects of DFDs

4.0	Introduction	96
4.1	Characterizing the syntactic aspects of flat DFDs	97
4.1.1	Characterizing structurally correct flat data flows	98
4.1.2	Characterizing structurally correct flat processes	98
4.1.3	Characterizing structurally correct flat external entities and data stores	100
4.1.4	Characterizing structurally correct process structures	101
4.1.5	The RS characterizing structurally correct flat DFDs	106
4.2	Characterizing the syntactic aspects of hierarchical DFDs (H_DFDs)	107
4.2.1	Characterizing structurally correct hierarchical data flows ...	107
4.2.2	Characterizing structurally correct hierarchical processes	113
4.2.3	The RS characterizing H_DFDs	117

4.3	Model and operational semantics for the PL	118
4.3.1	The PL R-CTRS	120
4.4	Limitations of the PL	122

Chapter 5: The Specification Level: Deriving Behavioural

Specifications from DFDs

5.0	Introduction	124
5.1	The Data Environment (DE)	126
5.1.1	Characterizing the object classes associated with data entities	126
5.1.2	Characterizing the structure of data entities	131
5.2	The Behavioural Specification (BS)	133
5.2.1	Algebraic state transition systems (ASTSs)	134
5.2.2	Specifying the behaviour of ExtDFD processes	135
5.2.3	Specifying ExtDFD actions	137
5.2.4	Characterizing the behaviour of data flows and data stores ..	141
5.2.5	Deriving the BS	143
5.3	The BS as a formal basis for reasoning with ExtDFDs	147
5.3.1	Investigating behavioural properties of ExtDFD with the BS	147
5.3.2	Proving implementations of the BS	148
5.4	Conclusion	150

Chapter 6: Two Examples of Deriving Behavioural Specifications from ExtDFDs

6.0	Introduction	151
6.1	The automobile cruise application	151
6.2	The computer-based university library application	170
6.3	Conclusion	194

Chapter 7: Conclusion

7.1	Thesis summary and achievements	196
7.1.1	Achievements	197
7.1.2	Comments	198
7.2	Further work	199
7.3	Conclusion	200

Bibliography	201
---------------------------	-----

CHAPTER 0

Introduction

0.1 The context

This section outlines the context in which the research described in this thesis should be placed.

0.1.1 The requirements specification problem

The increasing size and cost of software have been major concerns of software developers since the late sixties. These concerns are especially relevant today given the growing demand for, and scope of software in diverse application areas, and the widening influence of software on human welfare.

While there is no general consensus on the central problems afflicting software development, there is increasing evidence that the lack of thorough attention to the requirements analysis and specification phase of software development is a major contributor [YZCC84]. The evidence usually cited takes the form of extensive rewriting of the software and cancellations of projects whose completion was found to be unfeasible as a consequence of inadequate or inappropriate requirements analysis and specification [Boe76, Boe81]. The importance of the requirements analysis and specification stage as the first stage of software development should be self-evident. The result of this phase, the requirements specification, as well as being the basis for further development, provides the means by which the quality and applicability of the software can be measured [FREQ79]. In order to adequately support such a role in development, requirements specifications should have the following properties:

- *Understandability* : It is important that a requirements specification be understandable by users and implementors, as well as the specifiers, in order for effective communication to take place. This property is considered as being of prime importance by Balzer and Goldman [BG87]. Tse and Pong [TP86a] identify two main aspects of understandability - *complexity* and *clarity of description*. The reduction of complexity in an application can be achieved by the use of abstraction, and partition [YZCC84]. The use of abstraction allows one to suppress certain detail while concentrating on other essential detail, while partitioning permits one to represent the whole as the sum of its parts. The use of abstraction results in hierarchies of specifications, where a specification at a

lower level in the hierarchy presents detail ignored at the higher levels. For this reason, abstraction is viewed as a vertical decomposition tool. Partitioning allows for the modular building of specifications, and can be viewed as a horizontal decomposition tool. On the clarity of description, it is generally felt that graphic-based languages with few constructs are easier to understand than mainly textual languages.

- *Precision* : The requirements specification, as the basis of further development, must be stated in a precise, and unambiguous manner. This characteristic is necessary to reduce confusion or misunderstandings arising from information obtained from the specification.
- *Testability* : A requirements specification is said to be *testable* if it can be used to establish in an effective manner that an implemented application is, in some well defined sense, "equivalent" to it. In general, a notion of equivalence is based on a mapping from information in the requirements specification to information in the implemented application. If it can be proved that an implemented application is equivalent to a specification, then the implementation is said to be *correct with respect to the specification*. The activity of determining the equivalence of an implementation and its specification is called *verification*. As a prerequisite to verification, it must be possible to determine whether the different parts of the specification are consistent with each other. Such an activity is called *validation*.
- *Modifiability*: It is foolhardy to assume that requirements once given remain fixed throughout the development life of the software. Requirements can, and often do, change over time, thus it should be possible to modify a requirements specification without undue difficulty.

Currently, there is no single requirements specification language in which specifications possessing all the above characteristics can be expressed.

0.1.2 Formal requirements specifications

Requirements specification languages can be classified as being *formal* or *informal*. Formal specification languages have strict syntax and semantics. The specifications that are expressible by them are called *formal specifications*. Formal specification languages are seen by many reserachers as being necessary for expressing in a precise and unambiguous manner the requirements of applications (see for example [YZCC84, TP86a, BG87, FREQ79, Goo84, Zav82, ZY81, FP]). The use of formal specifications also permits validation of the specification by formal means, for example, by logical proof, automatic checks, or simulation.

Formal verification is also facilitated by the use of formal specification languages. Currently, there are two approaches to the formal verification of

software. In the first approach the software is developed independently of the specification, and showing that the software implements the specification means developing a formal proof that the program implements the specification in some well defined sense. After two decades of work on this approach it is now generally accepted that such an approach is not feasible for realistically sized applications [San88]. In the second approach, called the *transformation* approach, software is developed from requirements specifications via a series of refinement steps. The result of each step is a specification which incorporates the design decisions the step encapsulates. Such an approach can be pictorially depicted as a sequence of specifications as shown below:

$$SP_0 \rightarrow SP_1 \rightarrow \dots \rightarrow S$$

where SP_0 is the requirements specification and S is the implemented application. Each specification in the sequence can be thought of as an implementation of its predecessor, for example SP_1 can be thought of as an implementation of SP_0 . If each individual step can be proved correct, that is, if it can be proved that SP_i implements SP_{i-1} , then S itself is guaranteed to be correct with respect to SP_0 . As a formal development method, this approach offers more promise than the first, though it is not without its problems. For example, when applied to large and complex applications the individual specifications SP_i can become large and unwieldy resulting in some difficulty in proving the correctness of refinement steps [San88]. This problem can be solved by appropriately partitioning the specifications and refining them independently. Deriving an appropriate partitioning strategy is still an area of active research.

A number of formal specification languages have been developed since the early seventies, but their use in industry is limited despite their potential usefulness. Both technical and sociological reasons can account for this lack of use. On the sociological side, the proper use of formal specification languages requires a degree of mathematical maturity not previously required by software developers. Furthermore, formal specifications are difficult to read, even by the trained eye. On the technical side, the lack of a firm method addressing the entire development of software, which unifies at least some of the techniques is lacking. Current work on the transformation approach is directed at deriving such a total method for software development.

0.1.3 Thesis objectives

In the wider context, this thesis investigates an approach to integrating formal and informal specification techniques, in order to come up with a specification language which is both understandable, and formal. The approach involves

associating with informal specification tools and associated techniques a formal framework, thus enabling the generation of formal specifications from the (informal) specifications built using the tools and techniques. The informal specifications can thus be viewed as 'fronts' to the formal specifications, and should provide intuitive insight consistent with the formal interpretation it seeks to hide. A developer could then develop a specification in terms of the (seemingly) informal language, which could then be translated into a specification expressed in terms of the underlying formal language. Such an approach is based on a proposal put forward by Naur [Nau82, Nau85], which essentially states that formal expressions are extensions of informal expressions.

In the narrower context, this thesis provides a formal framework for *structured analysis* specification tools, mainly the *data flow diagram*, and also extends the notation so that aspects other than the data flow through an application can be specified. Most current languages provide support only for the specification of *what* the application does, ignoring other non-functional aspects such as timing, performance, and security. This is mainly because there is at present no comprehensive theory or methodology for specifying such requirements [YZCC84]. In this thesis attention is also paid to the specification of the time dependent (or control) aspects of applications.

0.2 Formal specifications from data flow diagrams

Structured Analysis (SA) is a methodology which addresses the requirements analysis and specification phase of software development [DeM78]. The primary tool of SA is the *data flow diagram* (DFD), which is a simple graphical language used for describing the required structure of an application in terms of the data flowing through it. At the time of its inception, SA was hailed as a radical approach to requirements analysis and specification because of its use of graphical specification tools as an aid to understanding. Less attention was paid to the lack of a firm conceptual basis for the tools and techniques until much later when the resulting problems reared their heads. Problems arose mainly from the different uses of the tools and techniques amongst practitioners, a direct result of the lack of a firm conceptual basis for them [Woo78]. This, inevitably, led to disagreements over the "proper" use of the tools and techniques, and encouraged many practitioners to incorporate customized extensions. Added to this, the irreversible nature of the transition from SA specifications to initial Structured Design (SD) specifications [YC78] limited their use in other than the requirements analysis and specification phase of software development [Pet88, Ric86]. Such transitions have also proved difficult to carry out in some cases, and require considerable experience

and skill on the part of the developer carrying out the transition [Ric86, Sho88]. A further problem with the SA approach is that it specifies applications in terms of a single aspect: the data flowing through it. For data processing applications this may have been adequate, but for other types of applications, for example embedded or real-time systems, other aspects are equally important.

Providing SA with a mathematical foundation may solve some of the problems associated with its use, if one can be found. It is this author's view that requirements analysis involves sociological processes which cannot be formalized in terms of any mathematical theory. For this reason this thesis does not attempt to provide an all-encompassing mathematical basis for SA, rather it restricts itself to developing a formal framework for its specification tools, primarily the DFD. The objective is to alleviate the problems associated with the use of SA specifications discussed above, and at the same time provide a specification language which is understandable, precise, and testable.

The formal framework consists of two parts: the *Picture Level* (PL), and the *Specification Level* (SL). The PL provides formal support for constructing DFDs by giving formal rules for building the syntactic entities involved. Specifically, the PL is a system for abstractly characterizing and formally reasoning about the syntactic structures of DFDs. The characterizations are abstract in the sense that they are representation independent. An effective, sound and complete deduction system can be associated with the PL, enabling its use as the formal basis for automated DFD syntax-checking tools which are based on the rules expressed by the PL.

The SL can be viewed as the part of the formal foundation which is used to specify the semantic aspects of DFDs. Specifically, the SL is a set of techniques for formally specifying the data, functional, and control aspects of control-extended DFDs. The data aspects concern the structure of the data depicted in DFDs, and the relationships between them, while the functional aspects concern the input/output behaviour of the processing components of DFDs. The control aspects of DFDs concern the interactions between the processing and data components of DFDs. The primary product of the SL is the *Behavioural Specification* (BS), which is a formal specification characterizing the behaviour of applications depicted by control-extended DFDs. Such a specification facilitates formal validation and verification activities, as is shown in this thesis.

0.3 Overview of thesis

Chapter 1 surveys some of the major extensions made to SA tools and techniques over the years since the inception of the methodology. It describes the

early SA approach of DeMarco [DeM78] and discusses the problems associated with it, and the manner in which some of these problems are tackled by other researchers. Chapter 2 introduces, in an informal setting, the formal basis for DFDs. This chapter can be viewed as the informal 'front' to the more formal parts of the thesis. Chapter 3 details the mathematical and operational foundations of the algebraic specification technique underlying the formal framework. The technique is based on the work of Broy and Wirsing on partial algebraic specifications [WB82], the work of Astesiano et al on relational specifications [ARW86], and the work of Mohan et al on inequational assumptions [MS87]. Chapter 4 describes the PL, while Chapter 5 describes the techniques in the SL. Chapter 6 applies the techniques described in Chapter 5 to both a data intensive application, and a control intensive application. The data-intensive example is a computer-based library application for a university, and the control-intensive example is an automobile cruise-control application. Chapter 7 discusses the merits and the limitations of the formal framework and pinpoints areas which require further research.

CHAPTER 1

Data Flow-Orientated Requirements Specification Techniques

1.0 Introduction

Data flow-orientated specification techniques (DSTs) provide mechanisms for representing the flow and transformation of data in an application. Using DSTs, applications are specified in terms of *flows*, representing data flow through the application, and *processes*, representing the components of the application which transform data.

The earliest indications of the use of DSTs in requirements analysis methods appeared in 1977, with the publication of the definitive papers on SofTech's Structured Analysis and Design Technique (SADT) [Ros77, RS77], which introduced a DST based on two diagramming tools called the *activity* and *data diagram*. During the next two years DeMarco, Weinberg, and Gane and Sarson published seminal books on structured analysis (SA) approaches [DeM78, Wei78, GS79], which used a DST based on a diagramming tool called the *data flow diagram* (DFD). The activity diagrams of SADT can be viewed as an early form of the DFD. Yourdon and Constantine, during the same period, published a second edition of an earlier book on Structured Design (SD), which included a technique for translating SA products into initial SD specifications [YC79]. The combined use of DeMarco's SA approach and SD is popularly known as the Yourdon SA/SD approach. The publications of these easy to read works on SA, coupled with the relatively informal style of the approaches, helped to establish the use of SA/SD as a viable method in the software industry.

The popularity of the SA/SD method stems mainly from its emphasis on creating clear, understandable specifications via the use of graphical notation, rather than text. The lack of an experience base and the novelty of a graphical language, led to the failure of authors to provide a firm conceptual basis for the tools and techniques [Woo88]. This led to variations in the use of SA tools and techniques amongst practitioners. This has been more evident in recent times, with the publication of papers and books which have extended SA/SD method in order to cope with the special nature of certain types of software, or to solve general problems related to the lack of a conceptual basis (see for example [Doc86, Doc87, Pet88, HP87, War86, Gom84, Sho88, TP86b, CTL87]).

This chapter presents an overview of the early SA specification techniques, and surveys some of the notable extensions made to it. The overview of the early SA techniques is based on the published work of DeMarco [DeM78], since it provides a comprehensive description of the tools and techniques.

1.1 Structured analysis (SA) specification techniques

The following are the specifications produced as a result of the use of SA techniques:

- *Data flow diagrams* (DFDs): Pictorial representations of the flow of data in an application. An application is usually represented by a hierarchy of DFDs.
- *A Data dictionary*: A repository containing descriptions of the data objects depicted in DFDs.
- *Process specifications*: Functional descriptions of the bottom level (primitive) processes in a hierarchy of DFDs.

The central specification tool of SA is the DFD. A DFD depicts the data flow relationships between the processing, data storage, and external components of the application. Definitions of the data structures associated with the data flows, and data storage components, are kept in an organized manner in a data dictionary. Process specifications are used to describe the procedural logic of the processing components depicted at the bottom level of a hierarchy of DFDs representing an application.

1.1.1 Data flow diagrams (DFDs)

A DFD is built using the following types of constructs:

- *data flow* - a construct representing the path on which data is conducted from one part of the application to another.
- *process* - a construct representing a component which transforms data.
- *data store* - a construct representing a repository of data.
- *external entity* - a construct representing components on the periphery of the application which send data to, or receive data from the application. External entities can thus be viewed as sources and sinks of data flowing through an application.

The symbolic representations of the above constructs differ amongst the major proponents of SA, as illustrated in Figure 1.1. A brief description of the above constructs follows.







Construct name	DeMarco constructs	Gane and Sarson constructs
Process		
Data store		
External entity		

Figure 1.1 A comparison of DFD constructs

Data Flow

A data flow is associated with a unique name, a direction, and a data type. Instances of the data type of a data flow are transmitted on the data flow in the direction associated with it. Notationally, data flows are depicted as named vectors. Data flows are not associated with any physical limits, nor is there any constraint on *how* the data flows through them. All that is of concern is *what* data is passed through them. Data passing through a data flow cannot be lost, modified, or destroyed during transmission. Also, data flows cannot create data.

Data flows are not meant to be representations of flow of control, nor are they meant to be associated with any control related interpretations (for example, as activators of processes). Data flows simply depict the data paths between the components of a DFD.

Different perspectives on data flows can be taken, depending on the aspect of data movement emphasized [TePi85]. From the standpoint of a process or an external entity, a data flow is an *input* or *output*, depending on whether the direction of movement is inwards or outwards with respect to the process. From the standpoint of a data store, a data flow represents an *update* if the direction of movement is inwards, or it represents a *retrieval* if the direction is outwards with respect to the data store. A data flow representing an update contains data which is to be written to the data store, while a data flow representing a retrieval contains data read from the data store. From the standpoint of a data flow connecting two constructs, the data flow is viewed as a data *interface* between the constructs. In what follows, the construct from which a data flow is directed away is called its *generator*, while the constructs it is directed towards are called its *receivers*. A data flow can have only one generator, though it may have many receivers. The case where a data flow has more than one destination is depicted by a branching data flow. One can view the point where the branching occurs as representing a copy function which creates copies of data on the data flow to be sent on all branches.

Process

A process is associated with a name, a non-empty set of inputs, and a non-empty set of outputs. Processes simply depict data transformations, thus issues related to their initiation and the manner in which they exchange data (via data flows) are not of concern. In other words, processes are not associated with any operational interpretation.

Transformations carried out by a process can be classified as being *logical* or *physical* [Pet88]. A process which logically transforms data does not change the physical appearance of the data. That is, subsequent-use of the data is affected by the way in which the process classifies it, and not by any physical change. For example, a process in an order processing application, which determines whether an order is valid or not, transforms the order logically. A process which physically transforms data, changes it in such a way that it is no longer recognizable. For example, a process which produces an invoice given a valid order and information on the parts needed to fill the order, transforms its inputs physically in order to create the invoice which is its output.

Data Store

A data store is associated with a name, and two sets of data flows representing retrievals and updates. Data stores are often likened to files in the SA literature to provide a more concrete view of what they represent, but can be used to represent other types of repositories of data which do not create or destroy data. Details of data organization, access mechanisms, and storage medium, are not depicted by data stores.

Data flows directed towards data stores are always generated by processes, while data flows directed away from data stores are always directed towards processes. The data flows representing retrievals and updates are the net data flows resulting from read and write accesses made by processes on the data store. Thus, a data flow representing a retrieval contains data retrieved as a result of a read access made by a process on the data store, while a data flow representing an update contains data to be written to a data store as a result of a write access by a process on the data store.

External Entity

An external entity is associated with a name, a set of inputs and a set of outputs. The external entities of a DFD represent the components of the environment with which the application must interface with. Such peripheral components may be persons, systems, or other applications, which generate data to and/or accept data from the application. External entities represent components that lie outside the scope of the application, thus details concerning the manner in which they derive or obtain data, and the way such data is used by the component are not depicted.

The inputs of external entities are always directed away from processes, while their outputs are always directed towards processes, thus external entities are always connected to other external entities and data store via processes.

Constructing and interpreting DFDs

A DFD is a *structural description* of an application in the sense that it depicts the data flows and transformations in an application without showing how the flows and transformations are actually achieved. Flows of control, relationships involving time, and any notion of a process execution or execution precedence, should not be inferred from DFDs. More importantly DFDs are not associated with any *operational* interpretation. A DFD is essentially a *documentation* tool [CTL87], used to depict the data paths in an application.

There are few strict rules guiding the construction of DFDs, permitting a great deal of flexibility in how they are built and used. The early work of DeMarco [DeM78] provides the following major guidelines:

1. Identify all *net inputs* and *outputs*, where a net input is an input whose generator is not a component in the DFD and a net output has at least one receiver which is not a component of the DFD. The net inputs represent the inputs to the application while the net outputs represent the outputs of the application
2. Derive the data paths from the inputs of the application to its outputs. This can be done either in a forward manner starting from the inputs, in a backward manner starting from the outputs, or in a middle-out manner starting from a set of internal data flows.
3. Label the data flows and processes in such a manner that their meanings are reflected in the labels.
4. Do not depict information related to the initialization and termination of the application. In other words, a DFD depicts an application in a "steady state", that is, when it is up and running.

5. Do not depict flow of control or control information.
6. Omit trivial error-handling details. DeMarco feels that one should get the "big picture" right first before paying attention to "odds and ends" like error-handling details.

Some of the above guidelines are open to interpretation, for example, practitioners have found it difficult to decide on what should be viewed as a control flow or a data flow in guideline 5 [Gom84, Ric86], also it is not clear what constitutes a "trivial" error-handling procedure in guideline 6.

Decomposing DFDs

It is easy to see that large and relatively uncomplicated applications, could result in large, complicated DFDs. *Hierarchy* is the abstraction mechanism used in SA to control complexity. The application of hierarchy to DFDs is provided via the *decomposition* activity associated with processes and data flows. The decomposition activity involves examining each process in a DFD to see if it can be broken down into simpler processes which act in concert to transform the inputs of the process to its outputs. If a process of the DFD is felt to be simple enough, that is, it is not necessary to break it down to simpler parts, the process is called *primitive*. The use of hierarchy enables the structured presentation of detail by DFDs.

The decomposition of a process is represented as a diagram, called the *child* diagram, consisting of process, data store, and data flow constructs. The process which is decomposed is called the *parent* process with respect to its child diagram, while the diagram containing the parent process is called the *parent* diagram, with respect to the child diagram. The *net inputs* of a child diagram are the data flows whose receivers are processes in the child diagram, but whose generators are processes not in the child diagram. The processes in a child diagram may be further decomposed, and so on, resulting in a hierarchy of diagrams.

In SA, an application's data flow structure is specified by a hierarchy of DFDs, resulting from successive process decompositions, made up of a top, bottom, and middle levels. The top, or level 0 of the hierarchy, is a single DFD called the *context diagram* which consists of a single process, whose inputs are the net inputs of the application, and whose outputs are the net outputs of the application. The context diagram serves to delineate the boundaries of the application. The bottom level consists of DFDs containing only primitive processes, while the middle levels consist of the intermediate DFDs in the hierarchy.

In order for child diagrams to be interpreted correctly within the context of their parent diagram the following conditions must be satisfied:

- Data flows into and out of a process in a parent diagram correspond to the net inputs to and a subset of the set of all outputs from its child diagram. A child diagram satisfying this rule is said to be *balanced* with respect to its parent diagram. Decomposition of data flows, resulting in data flows representing the constituent parts of the decomposed data flow, is also allowed in parallel with process decomposition. The matching of data flows in such a case depends on the existence of information from which relationships between the different levels of data flows can be established. Such information is kept in the data dictionary.
- Data stores introduced in child diagrams are accessed only by the processes in the child diagram.

Again, few formal rules exist for constructing a leveled set of DFDs, though guidelines do exist. The guidelines are concerned mainly with labeling conventions, data flow balancing, and considerations to be made when deciding on when to stop process decomposition.

Evaluating DFDs

The development of hierarchies of DFDs may lead to DFDs of poor quality. The lack of a firm conceptual basis for DFDs, as reflected in the lack of formal rules for constructing DFDs, makes it difficult to formally state criteria for determining the quality of a DFD. Guidelines and techniques for evaluating the quality of DFDs are provided by DeMarco, and can be classified as follows:

- *Completeness* criteria are concerned with whether there are any missing parts in DFDs. For example, data stores which are read-only or write-only, or processes which do not transform data warrant further questioning.
- *Consistency* criteria are concerned with the compatibility of DFD constructs and their child diagrams. Within a hierarchy of DFDs, consistency is maintained through appropriate connectivity, decomposition, consistent naming of constructs, and through the balancing of data flows.
- *Correctness* criteria are concerned with the use of DFD constructs. For example, a DFD is incorrect if it depicts control flows or flows of control.
- *Communicability* criteria are concerned with the complexity and conceptual clarity of DFDs. These criteria usually emphasize graphic organization, legibility, reproducibility, and presentation quality.

1.1.2 The data dictionary and process specifications

DFDs, as described above depict only the paths of data through an application. It does not provide descriptions of the content of its data flows and data stores, henceforth called the *data objects* of the DFD, nor does it provide details of how the inputs of processes are related to their outputs. Thus, by themselves, DFDs do not provide adequate specifications of an application's requirements. Descriptions of the data objects are provided by a data dictionary, while procedural descriptions of processes are expressed by process specifications associated with the primitive processes of a hierarchy of DFDs.

The data dictionary

A data dictionary provides descriptions of the data objects (data flows and data stores) in a hierarchy of DFDs depicting an application. Three levels of data descriptions can be identified [GS79]:

- *data elements* are items of data which are not usefully decomposed into their components, for example, an age;
- *data structures* are composites of data elements and other data structures; and
- data flows and data stores as described in the previous sections. Data flows and data stores are made up of data structures, while data structures are composed of data elements.

The languages used by data dictionaries to express data definitions are essentially quasi-formal, providing constructs which enable developers to define data objects in terms of their components. For example, a particular language may have notation for representing data *sequences*, a *selection* of data, and *repeated groups* of data. An example of a portion of a data dictionary entry is shown in Figure 1.2.

```
cust_order = cust_name + cust_addr + order_detail_line
order_detail_line = list(part_number + quantity)
part_number = 0001|...|99999
cust_addr = house_number + street_name + city + country +
zip_code
```

Figure 1.2 An example of data dictionary entries

In the figure the "=" symbol means "is composed of", while the "+" symbol builds data structures which are sequences of data items and/or other data structures. For example, *cust_order* consists is a data structure which is defined as a sequence of *cust_name*, *cust_addr*, and *order_detail_line* data structures. The data structure *order_detail_line* is a list of data structures which are sequences of

part_number and quantity data structures. The data structure part_number is a data item which can take any integer value between 0001 and 99999.

As well as storing data definitions, a data dictionary may also contain information about the frequency of occurrence, volume of data, size of data stores, security considerations, priorities, and any other information pertaining to the use of the data objects that is needed to gain an understanding of the requirements of an application. For large applications the data dictionary can become complex, thus making it difficult to manually maintain, and to relate DFD data objects with their definitions. This makes the automation of their maintenance and cross referencing activities essential. A number of automated data dictionary systems supporting such activities are commercially available.

Process specifications

Process specifications describe the procedural logic of the primitive processes in a hierarchy of DFDs. Decision tables and trees are used to describe processes with complex branching conditions, while languages such as Structured English or pseudocode are used to specify less complex processes. Such languages incorporate the basic procedural constructs, *sequence*, *selection*, and *repetition*, with a limited set of natural language phrases. An example of a portion of a process specification in Structured English is shown in Figure 1.3. The process Check_order determines whether an input order is invalid or not, by checking whether the customer is on the files, and checking whether the ordered parts are available.

```
PROCESS: Check_order
select a cust_order, check that customer is in the customer file
if customer not on file then classify cust_order as "INVALID"
else
  for each order_detail_line
    check that the part is in the part file
    if part not in part file then classify cust_order as "INVALID"
    else
      check that there is sufficient parts in stock to satisfy order
      if not sufficient parts then classify cust_order as "INVALID"
```

Figure 1.3 An example of a process specification in Structured English

1.1.3 SA and design

SA is based on a lifecycle model, where requirements analysis and specification phase is followed by application design and implementation. Methodologies incorporating SA tools and techniques, usually employ structured design (SD) tools and techniques [YC79], together with techniques for

transforming SA specifications to initial SD specifications, in order to cover the requirements analysis and design phases of software development [DeM78, YC79].

SD is a strategy for producing modular, top-down designs. As originally conceived, SD was concerned with the systematic derivation of specifications of program structures which were maintainable and easily tested. Using SD to derive designs for applications entails viewing applications as collections of functions. This view permits applications to be specified as a hierarchy of logical functional units, called modules. The primary specification tool of SD is the Structure Chart (SC), which depicts the architecture of an application in terms of hierarchically structured modules. Positions in the hierarchy are determined by the modules' calling relationships, and the data exchanged by them [YC79]. SD also provides a number of heuristics and guidelines for evaluating designs.

The transition from SA to SD is dependent on the type of application represented by the DFDs. Applications can be classified as follows:

- Applications in which the same input data values always produces the same output data values are said to be *transform-oriented*.
- Applications in which the same input data values do not necessarily produce the same output data values are said to be *transaction-oriented*.

The outputs in a transform-oriented application are functions of the inputs alone. Batch-type applications, where the user enters the data then initiates the system, and where the results are always the same if the input values are the same, are examples of transform-oriented applications.

In a transaction-oriented application the output values cannot be regarded as functions of the input values alone, since the application is also associated with different modes of operation, which affect how the input values are used by the application. The modes of operation of an application are called its *states*. Input values received when an application is in a particular state may cause the application to change its mode of operation, that is, move to another state. How data is transformed is dependent on the current state of the application, since information valid in one state may be invalid in another. Thus, the output depends, not only on the inputs, but also on the current state of the application. Alternatively stated, the output from a transaction-oriented application is a function of the series of prior inputs to the application which can cause the application to change its state [Pet88].

Transform-oriented applications may be viewed as transaction-oriented applications having only a single state. Transaction-oriented applications can also be viewed as a combination of transform-oriented sub applications under the control of a 'master' process or module.

The transition from the data flow representations of SA to SD specifications can be carried out in a five-steps [Pre87]:

1. The type of application, with respect to its information flow, is established.
2. The centre or 'master' process in the DFD is identified.
3. The DFD is mapped into an initial program structure specification.
4. The control hierarchy is defined by factoring.
5. The resultant structure is evaluated and refined using SD measures and heuristics.

In order to transform a set of DFDs to an initial SC, the type of the application, as represented by the DFD at level 1 (i.e. the child diagram to the Context diagram), is first determined in step 1, then the process in the DFD which is to act as the 'master' module (called the *centre* of the application) in the initial SC, is determined in step 2. Techniques exist for determining the centres for both transform, and transaction-oriented applications [DeM78, YC79], but this step usually requires a great deal of experience and insight in order to come up with a centre which would lead to a good initial SC [Pet88, Sho88]. Once the centre of an application is chosen, then the surrounding processes become subordinate modules, with their decompositions defining subordinate levels in the hierarchy of modules as is done in step 3. Step 4 defines the control hierarchy by factoring, which results in a structure where the top modules perform only control operations, the bottom level performs the input/output, and computational operations, and the middle levels carry out a mixture of operations. In step 5, the derived SC is refined according to the measures and heuristics associated with SD.

1.1.4 Limitations of SA tools and techniques

The SA approach to requirements analysis generates mainly *descriptive* specifications of applications. DFDs, for example, are no more than documentation tools, while data dictionary definitions and process specifications rely mainly on quasi-formal textual descriptions [Doc87, CTL87]. The limitations of SA tools and techniques stem mainly from the quasi-formal, descriptive nature of the generated specifications, and their sole emphasis on the data flow aspects of an application. On the other hand, the informal nature and simplicity of the tools, coupled with the use of graphic notation supported by hierarchy, are often cited as the major strengths of the tools. These qualities make the approach easy to learn and use but suggest a lack of expressive power, which, together with the lack of a firm conceptual basis for the tools, encourages extensions to the notation and

disagreements over interpretations, making comprehension of the specifications apparent rather than actual [Woo88].

The limitations of the SA tools and techniques for requirements specification identified here are grouped into the following two classes :

1. limitations associated with the use of SA tools and techniques for constructing and validating requirements specification; and
2. limitations associated with the use of SA specifications as a basis for verification.

Limitations on the construction and validation of SA specifications

The lack of a theory formalizing the conceptual basis of SA tools and techniques places limitations on their use in the construction and validation of requirements specifications. In the construction of specifications, the lack of a firm conceptual basis allows a fair degree of flexibility in the manner in which the specifications are created. Such flexibility can lead to apparent misuse and/or disagreements over the "correct" interpretation of specifications. For example, Docker lists the following as the most common forms of misuse of DFDs [Doc87]:

- Structurally inaccurate DFDs, for example, "simplified DFDs" in which external entities communicate directly with data stores, or in which external entities are not shown.
- Viewing and specifying the application at too low a level of abstraction. This usually manifests itself as an overuse of data stores, for example using a data store to hold transactions which are later processed sequentially.
- Over abstraction, where the analysis of an application is finished at too high a level of abstraction.
- Textual glueing, where parts of the application which are not easily expressible in the quasi-formal languages of SA are described in natural language.
- Regarding DFDs as a functional decomposition tool.

Some practitioners may not view some of the above as misuses. Whether the above are actually misuses or not will remain a judgemental issue without a firm conceptual basis for the tools and techniques. Creating such a basis for SA is not an easy task since it requires detailed knowledge of the processes involved in structured analysis and specification. Such processes are currently not well enough understood thus more research, and a larger experience base, are needed before a useful conceptual basis covering all aspects of SA can be developed [Doc87].

Another factor which limits the use of SA tools and techniques for requirements specification is their sole emphasis on data flow aspects of applications. The SA approach ignores other aspects of applications such as the

relationships between data objects of the applications (data aspects), and time-related relationships between processes (control aspects). The emphasis on data flows, and the resulting functional view of processes can be traced to the early use of the SA approach for specifying data processing applications. Such applications are usefully viewed as information processing systems, making the SA approach appropriate. In more recent times application areas have become more diverse, requiring aspects other than data flow to be specified in their requirements specifications. For example, the control aspects of real-time and embedded applications need to be specified in the analysis phase since such aspects are inherent parts of the applications. The insistence that no control detail be specified in the SA approach seriously limits its use for specifying such applications.

Validation of requirements in the SA approach takes the form of user reviews of the generated specifications. As pointed out earlier, the lack of a conceptual basis can make comprehension of such specifications apparent rather than actual. The absence of a formal syntax and semantics for the specification languages also makes it difficult to prove the absence or presence of desired properties in specifications. For these reasons rigorous validation of specifications is difficult.

Limitations on the use of SA specifications as a basis for verification

The specifications produced from SA are of a logical nature, thus no operational model can be consistent with them. This seriously inhibits the use of SA specifications as a base for verification, since detailed designs and implementations are expressed in operational terms. Formal verification of designs and implementations against SA specifications are impossible for this reason. The best that can be done is an intuitive form of verification, which may be inadequate for some complex applications, and not healthy for certain critical applications where software failures could have drastic effects socially or economically.

The SA/SD approach provides techniques for transforming SA specifications to initial SD specifications. The quasi-formal nature of SA specifications means that at best such techniques are themselves quasi-formal. This has led to a number of problems in applying the techniques in practice, with some practitioners actually reporting that the techniques were not applicable in some cases [Ric86, Sho88]. Using the techniques require a great deal of skill and experience, especially when the application has both transaction and transformation characteristics [Sho88]. Furthermore, the transition from SA to SD is an irreversible process, thus changes made in the design phases are not easily reflected in SA specifications [Pet88]. This is because a shift in perspective is made when going from SA specifications, which are concerned mainly with the data flow relationships in an application, to SD

specifications, which are concerned with the operational structure of the application. Once SA specifications are transformed into SD specifications they are of limited use in the subsequent stages of software development.

1.2 Extensions to SA tools and techniques

The limitations discussed in the previous section seriously hamper the use of SA tools and techniques in the specification of complex applications. To address these limitations a number of authors have suggested extensions to SA tools and techniques. Some notable extensions are reviewed in this section.

1.2.1 Yourdon's Structured Method (YSM)

The early work of DeMarco on SA has been extended by the Yourdon group to create a method, called the *Yourdon Structured Method* (YSM) [MW86a, MW86b, Woo88, MP84]. YSM improves upon the earlier SA approach in three ways [Woo88]:

- The emphasis in YSM is on the modelling of *behaviour*, rather than just function.
- YSM introduces tools and new notation for modeling particular aspects of applications ignored in the original approach. Data relationships are expressed via *entity-relationship diagrams* (ERDs), while time-dependent behaviour is expressed with the aid of additional DFD notation and *state transition diagrams* (STDs).
- YSM is divided into three distinct phases. The first is the *feasibility study* which involves the study of any current application and its environment. The second phase is *essential modeling* [MP84] which produces a logical specification called the *essential model*. The third phase is *implementation modeling* which involves incorporating into the essential model aspects of a user's requirements which are dependent on technology. The resulting specification, called the *implementation model*, can be viewed as an initial design specification.

The essential model of an application describes the context in which an application is to exist, and the behaviour of the application. Three aspects of behaviour are described by the essential model: *functional*, *data* and *control*. The functional aspects, which are also modeled in the earlier versions of SA, are concerned with how applications transform their inputs to outputs, while the data aspects are concerned with the structure, and use of data in applications and the relationships between them. The control aspects of an application are concerned with its time-dependent behaviour.

The essential model consists of an extended form of DFDs called *Transformation Schemas (TSs)*, ERDs for modeling data relationships, and a data dictionary for defining data objects. TSs depict both data and control dependencies, using additional constructs for depicting control aspects on DFDs. The constructs used in TSs are shown in Figure 1.4.

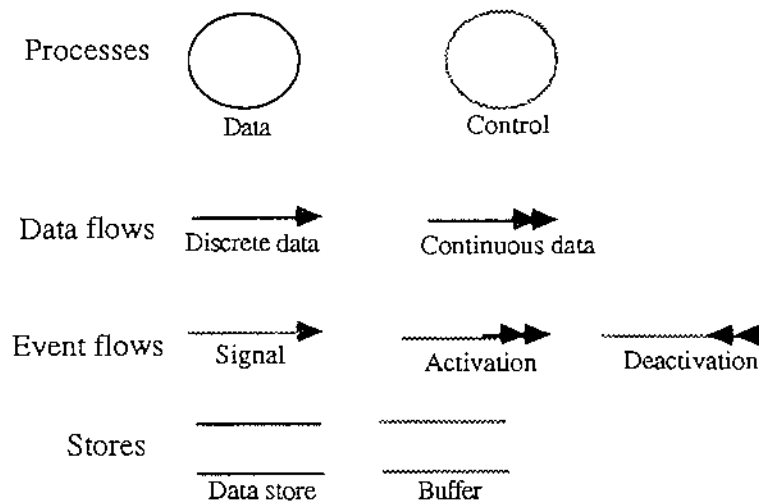


Figure 1.4 Transaction schema constructs

In TSs, flows crossing the boundary between the application and its environment are representations of events. Such events are changes in the environment which lead to sets of actions by the application called *responses*. Data flows depict events which are associated with data, while control flows depict events which are not associated with data. Data flows which depict events that occur at discrete points in time are called *discrete*, while those depicting events that occur frequently are called *continuous*. In a TS a data flows can be combined to form a single data flow representing the combined data flows, or a data flow may be split into other data flows, where the data flows resulting from the split represent constituent parts of the split data flow. Such data flows are said to be *composite*, and they eliminate the need for processes whose sole purposes are to combine or split data flows. Labeling conventions are used to distinguish split and combined data flows from branching data flows which carry the same data on each branch. Each branch in a split or combined data flow is uniquely named. If the branches of a branching data flow are not named then it means that the branches represent the same data flow, and thus carry the same data instances on all branches.

TSs utilize two kinds of processes: *data* and *control* processes. A data process transforms data inputs into outputs. A control process represents aspects of the

control logic associated with part of an application. Control flows are used to control data processes. They can affect the behaviour of processes in three ways:

- *Enable* - to enable a process means to allow it to be activated by a data flow.
- *Disable* - to disable a process means to prevent it from being activated.
- *Trigger* - to trigger a process is to activate a process in such a way that it deactivates itself when it has completed its task.

Control flows representing the above events are called *prompts*.

Control flows coming in from the environment are interpreted by control processes. The manner in which events depicted by control flows from the environment affect the application is specified by STDs. The effect the occurrence of an event in the environment has on the behaviour of an application is dependent on the current *state* of the application. The state of an application is a mode of operation that is externally observable. That is, if the application's behaviour was monitored, each state, or mode of operation, would be distinguishable. Information pertaining to the state of an application is kept by the control process. The occurrence of an event in the environment may cause a change in the current state of an application, which may in turn cause certain data processes to be enabled, disabled, and/or triggered. The events which may cause changes in the state of an application are depicted by control flow inputs of control processes, while the outputs of control processes depict the manner in which the control processes affect the behaviour of associated data processes. The behaviour of control processes is specified by *state transition diagrams* (STDs). An example of a STD is shown in Figure 1.5. The rectangular boxes represent the states of the application, while the labeled arrows represent state changes, where the labels specify the event causing the transition and the actions resulting from the occurrence of the event.

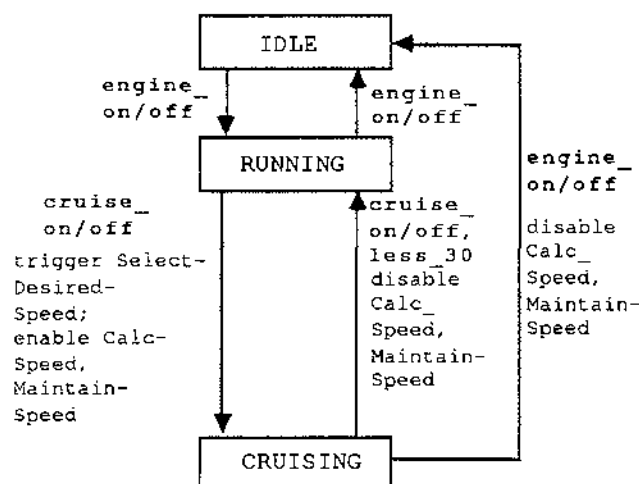


Figure 1.5 An example of a state transition diagram

Data stores in a TS, like data stores in traditional SA, are abstractions over files. TSs also use a special kind of data store called the *event store* or *buffer*. Buffers are abstractions over stacks and queues, and are used to represent delays between the occurrence and recognition of events by data processes.

The following summarizes the formation rules and interpretations associated with TSs:

Data processes

- A data process may have at most one input flow which arrives independently of any action carried out by the data process. Such a flow is called *active* and can be a discrete data flow from an external entity, another data or control process, or a buffer, or a control flow from a control process or buffer [War86]. Data processes with active inputs are not allowed to have continuous outputs. When the active input is a triggering prompt then the process must also be associated with an input from a data store. A data process's enabling and disabling input prompts, and flows from data stores are not considered to be active inputs.
- Data processes with only continuous input and output data flows accept inputs and produce outputs continuously. Enabling and disabling prompts may be associated with such processes, in which case the processes continuously produce outputs only when they are enabled. Data processes with continuous inputs and discrete outputs can only occur when there is also an active input to the process, in which case only the value on the continuous flow at the time an event occurs on the active flow is used to produce the values on the discrete outputs.
- A data process may have zero or more *active* outputs, where an active output is an output created by the process which can be an active input for another process [War86]. An active output can be a discrete data flow or a control flow to a control process, buffer, or external entity. A data process with two or more active outputs can produce output on only one in a single activation.
- A data process can have any number of continuous data flows, and discrete data flows to and from data stores.
- A data process with discrete output flows and no active inputs produces output via one of the following ways: forced by a triggering prompt; upon the occurrence of some significant value of a continuous input; or on the occurrence of a specific time retained and read in from a data store [War86].

Control processes

- Inputs and outputs to control processes are restricted to control flows. A control process must not have data flow inputs or output.

Flows

- Each discrete data flow must be connected to a data process at one end while at the other end may be a data store, buffer, external entity, or another data process.
- Each continuous data flow must be connected to a data process at one end while at the other end may be another data process, or an external entity.
- Control flows may connect any pair of processes, a data process and a buffer, or a process and an external entity. Control flows emanating from control processes to data processes are prompts.

Data stores

- A data store must be connected by a discrete data flow to at least one data process.

Buffers

- A buffer must be connected by a discrete input (data or control flow) to at least one data process, and by a discrete output (data or control flow) to at least one data process. Types of flows for a single buffer are not mixed [War86].
- Every buffer is associated with a capacity, which may be finite or not. This is interpreted as the number of units that can be stored in the buffer.

Essential modeling [MP84, Woo88] entails identifying the events in the environment that affect the application, and then developing a TS reflecting how the application responds to each identified event. In parallel, the control processes are specified by STDs, and data relationships by ERDs. The TS is then restructured into a leveled set of DFDs following the guidelines of the original SA approach. Implementation modeling extends the essential model with technology-dependent detail, such as performance and size constraints, and also provides specifications of the software structure, derived from the TS, in the form of Structure Charts (SCs) [YC79]. It is carried out in three phases:

- Model the physical processors associated with the application. This involves grouping processes according to the physical processors they are to be implemented on.
- Model the software environment in which the application is to exist, for example, some functions of an application may be carried out by operating systems and/or database management systems.
- Model the structure of the software to be produced. This involves translating the TS to an initial SC.

Limitations of YSM

YSM does tackle some of the problems associated with the use of SA tools and techniques for specifying requirements. It permits the specification of aspects other than the data flowing through an application, and provides a firmer conceptual basis for the tools and techniques than that provided in the original SA approach. Constructs are provided for representing control elements, together with a set of formation rules, which, if adhered to, permits a logical interpretation of the specifications. Such logical interpretation can be used as the basis for informal analysis of, and reasoning about, the specifications [Woo88].

The specifications generated by YSM are still descriptive in nature. The difference between SA and YSM in this respect is that YSM provides *descriptions* of other aspects of applications. The logical model implied by the descriptions is not consistent with any operational model thus formal verification of implementations is not possible. Furthermore, the approach is still reliant on the techniques for transforming DFDs to SCs, which, as discussed earlier, is problematic, and, when it can be done, is irreversible. Thus the specifications generated from YSM, like SA specifications, have limited use in other than the requirements analysis phase.

1.2.2 Hatley's Extensions

Hatley's extensions to SA [HP87], like YSM, were borne out of the need to model aspects other than the data flowing through an application. The extensions to the specification techniques mainly concern the modeling of control aspects in parallel with the processing aspects of applications. Hatley also extends SA by providing techniques for building an architectural design for the application. Such a design is similar in purpose to the implementation model of YSM.

Hatley provides techniques for building two types of models: the *requirements model* (RM) and the *architecture model* (AM). The RM specifies the required processing behaviour of an application. It is a description of the functional and control requirements of an application. The AM assigns the processes of the RM to physical modules that make up the application and establishes the relationships between them. The AM is thus an architectural design of the application whose required behaviour is specified by the corresponding RM. Below a description of these two models are given.

Requirements Model

The RM specifies *what* an application is to do in terms of its *functional* and *control* aspects. The functional aspects of an application are captured by the process model which consists of a leveled set of DFDs supported by a requirements

dictionary (RD), process specifications (PSPECs) and response time specifications for the primitive processes of the DFD. The leveled set of DFDs are constructed using the leveling and balancing principles associated with DFDs in the SA approach. Data flows, as in YSM, are events that are associated with data components which may also be split and combined as in YSM.

The control aspects of an application are captured by the control model, which consists of a leveled set of *control flow diagrams* (CFDs) supported by *control specifications* (CSPECs) and the RD. CFDs depict the flow of control signals (events not associated with data) while the CSPECs indicate how the signals affect the behaviour of the application. For each DFD in a leveled set of DFDs representing the functional aspects of an application, there is a corresponding CFD showing the control dependencies amongst the processes and external entities of the DFD. Thus, there is a direct correspondence between the levels in the leveled set of DFDs and the leveled set of CFDs for an application, where level 0 DFDs are associated with level 0 CFDs, level 1 DFDs with level 1 CFDs, and so on. CFDs consist of the processes, external entities, and data stores in their corresponding DFDs. CFDs, though, do not show data flows but control flows depicted by dashed directed arcs which obey the same routing rules as data flows. CFDs also utilize an additional construct, a bar, representing the control processing part of the application. A CFD may have a number of such bars with control flows going into and from them, all representing the single control processing unit of the CFD.

Control processing in a CFD is specified by a CSPEC. Control behaviour is modeled by viewing applications as finite state machines whose inputs and outputs are control flows. Diagrammatic and tabular representations of finite state machines are contained in CSPECs. State transition diagrams (STDs), as in YSM, are used to show states of the application and how they are influenced by control flows. Events and actions are shown on STDs as "Event/Action" labels on each arrow depicting a state transition. The events on these labels are the control flows directed towards the bars in the corresponding CFD. Process activation tables (PAT), give the conditions under which processes are activated. The actions shown in the STD are entered into the PAT and associated with the processes they activate or deactivate. Processes which are not controlled in this way are "data triggered", that is they are activated each time there is sufficient data on their inputs to perform the specified function.

The RD provides definitions for the data and control flows in the process model. Flows are classified as being either primitive or non-primitive, where non-primitive flows are groups of primitive flows. Primitive flows are defined in terms of their attributes. The process model for an application also consists of a

specification of the timing requirements, stating the required recomputation rates for interface outputs and the required input/output response times for the signals at the application's interface. Recomputation rates are specified in the RD, while response times can be given in a tabular form.

The process model is a logical model of an application's processing behaviour. Operationally, the process model can be viewed as an idealized, infinitely fast machine. Thus processes transform their inputs instantaneously, while control flows are interpreted instantaneously by the control processing components of the model. The process model, though, is not intended to represent an actual machine, rather, like SA specifications, it is merely a description of the processing requirements of an application.

Architecture Model (AM)

The AM shows the physical entities making up an application, defines the information flowing between these physical entities, and specifies the channels on which this information flows. The primary tool of the AM is the architecture flow diagram (AFD) which depicts the physical structure, or architecture, of the application in terms of its physical entities, called modules, and the information flow between them. The main purpose of the AFD is to allocate the processes given in the RM to physical units of the application. Additional processes may also be required in the AFD to support the new physical interfaces. Modules provide four additional perspectives to applications: *input processing*, *output processing*, *user interface*, and *maintenance or self-test processing*. The processes making up the input and output processing aspects represent the processes needed for the module to communicate with other modules, and to transform information to and from an internally usable form. Such processes are not shown in the RM. The user interface aspect is a special case of the input/output processing aspects. It is separated because of the special considerations, such as human factors, that affect the definition of the user interface, but have little to do with the interfaces between modules. The maintenance and self-test processing aspects concern the processes required to perform the self-monitoring, redundancy management, and data collection for maintenance purposes. The AFD itself is treated as a physical module, and depicts the modules modeling the above four aspects as well as the processing and control aspects specified in the RM. The modules in AFDs may also be "decomposed" into AFDs showing the modules modeling their six aspects. In this way hierarchies of AFDs are created.

The physical means, or channels, by which modules communicate with each other are depicted by the architecture interconnect diagram (AID), supported by

architecture interconnect specifications (AISs) which are textual characterizations of the channels. The AID and the AFD for an application may be combined in a single diagram if the result is not too complex.

An AFD is supported by architecture module specifications (AMSs), and an architecture dictionary (AD). AMSs define the inputs, outputs, and processes allocated from the RM for each module in the AFD. The AD contains the data and control flow definitions in the RD, plus the allocation of these flows to modules in the AFD.

Limitations of Hatley's extensions

Hatley's extensions, like YSM provide notation and concepts for modeling aspects other than the data flowing through an application. Furthermore, it provides tools and techniques for creating an initial architectural design from the specification of the required behaviour of an application, which is similar in form to a DFD. This means that while there may be a shift in emphasis in going from analysis to design, there is a straightforward relationship between the initial design specification and the requirements specification, thus facilitating traceability, and consistency checking.

Like YSM, Hatley's extensions suffer from their reliance on quasi-formal notation for definitions and the descriptive nature of the graphical specifications. Thus, like SA, the tools and techniques lack a formal basis for supporting rigorous validation and verification.

1.2.3 ADISSA

ADISSA (Architectural Design of Information Systems based on Structured Analysis) is an architectural design method that is compatible with and forming a direct continuum with SA [Sho88]. This is essentially achieved by viewing external entities as event triggers. Shoval argues that taking such a view does not require additional notation to represent control and timing detail as in YSM, which results in a change in the appearance of DFDs which may reduce their conceptual clarity [Sho88].

ADISSA takes a transaction-orientated view of applications, where a transaction consists of one or more processes performing specific functions in response to stimuli from the environment. The view of applications by ADISSA is based on Wasserman's and Stinson's view of interactive applications as consisting of: (1) a user interface, (2) operations on data, and (3) a database [WS79]. The related concepts in ADISSA are: (1) a menu tree describing the external architecture of the system; (2) transactions, describing the internal architecture; and (3) a database schema of normalized records.

A system of menus, organized as a hierarchy of menu screens, forms the *external architecture* of the system from the user's point of view. Menu screens consist of *selection lines* providing access to other menu screens in the system, and *terminal lines*, which invoke procedures in the application. The system of menus is called a *menu tree*, and is derived from a hierarchy of DFDs specifying the application, where the menu lines are generated from the processes connected to external entities by data flows. Primitive processes generate terminal lines, while other processes generate selection lines.

Transactions consist of primitive processes which form a data dependency chain, and of data stores and external entities which are connected to these processes. A hierarchy of DFDs, in general, consists of more than one transaction. These transactions make up the internal architecture of the system. ADISSA's design objectives as concerns transactions are given below.

For each transaction identified:

- identify what activates it;
- determine the order in which the component processes are executed; and
- determine the input/output operations carried out, and the data store accesses made by it.

Transactions are activated by *events*, and are classified by the types of their activation event, given below:

- *User event* - generated by a user (represented by an external entity) of the system, usually via the menu tree. Data flows between external entities, known as *user entities* (UEs), and primitive processes identify *user-transactions*: a data flow from an external entity to a primitive process signifies a user event that causes an application user to trigger a transaction which inputs data, while a data flow from a primitive process to an external entity signifies an user event which causes an application user to trigger a user-transaction which provides data on the data flow.
- *Time events* - generated by a special kind of external entity, called a *time entity* (TE). Time events are used to model events that activate a transaction at a predetermined point in time or time interval. TEs trigger transactions, called *time-transactions*, in much the same way as UEs trigger user-transactions.
- *Real-time events* - generated by a special kind of external entity, called a *real-time entity* (RTE), which is an abstraction of a sensor/detector device. The type of information generated by RTEs are represented by the flows connecting them and processes. RTEs trigger *real-time-transactions*.
- *Communication events* - generated by *communication entities* (CEs), which represent abstractions of communication mechanisms between the system being

modeled and other systems. Communication events occur when a message is received, and the message triggers a *communication-transaction*.

A "chain effect" occurs when two or more primitive processes within a transaction sequentially activate each other. The chain effect terminates when the data generated by a process is sent to a data store or an external entity.

The *trigger* of a transaction is the process connected to the external entity that activates it. The trigger is not necessarily the first process of a transaction to be executed, since it may be located anywhere in the chain of processes making up the transaction. If it is the first then the chain effect proceeds forward; if in any other position then it is necessary for the preceding processes to execute before the trigger can be executed. Thus, the event generated by the external entity is seen as activating the transaction from the start of the chain in all cases, even though the event entity may be at the end (or middle) of the transaction.

Structured descriptions of transactions replace the process specifications of SA. Shoval argues that it is more useful to specify the behaviour of transactions, rather than individual primitive processes, since the interrelationships among the processes, and data stores of the transaction can also be specified. Transaction specifications consist of a top- and a bottom-level description of a transaction. The top-level specification describes the externally observable behaviour of the processes making up the transaction. Specifically, the following four primitive functions are used in the top-level specification language:

- *execute* process, performs a primitive process, whose detailed specification is given in the bottom-level specification;
- *read/write* transfers data from data stores to processes;
- *input/output* transfer data between external entities and primitive processes; and
- *move* transfers data between primitive processes.

The above functions are used together with the control structures of structured programming in order to derive a top-level "skeleton" specification for transactions. The bottom-level description details the internal logic for each process in the transaction, and can be stated in the same manner as process specifications in SA, for example, using Structured English, or Decision trees.

Shoval provides a methodology for developing ADISSA specifications which consists of the following steps:

1. *Functional analysis*, producing hierarchical DFDs and a data dictionary; and includes analysis of events and external entity types.
2. *Menu tree design*, resulting in a menu tree for the application represented by the hierarchy of DFDs.
3. *Transaction design*, involving identification of transactions, finding their triggers, and determining their order of execution.
4. *Transaction specification*, resulting in structured descriptions of transactions.
5. *Database schema design*.
6. *Input/output schema design*, which associates input/output descriptions to the inputs and outputs described in the top-level transaction specification.
7. *Design of the ADISSA data dictionary*, which is an extension of the data dictionary derived in step 1, consisting of a menu tree dictionary containing details of all screens and their lines, and the transactions dictionary containing details of the transactions.

In [SP88] the use of ADISSA in a prototyping environment is described. The aim is to enhance user-analyst communication to enhance validation of the requirements specification by the user. The following are the types of prototypes that can be generated from ADISSA products:

- Interface prototype - a hierarchy of menu screens generated from the menu tree using a menu generator module. The user is allowed to navigate through these screens and make comments on the user interface of the application.
- Data prototype - a database schema created using the database management module of an application generator and its definition language.
- Process prototype - a collection of program modules based on the transaction specifications.
- Application prototype - a program based on the top-level descriptions of transactions, and on the previous prototypes.

ADISSA provides limited support for the specification of control requirements, in the form of specialized types of external entities. Control signals are not depicted, thus limiting the means for specifying conditions under which a process or transaction is can execute. Furthermore, the lack of a formal basis for ADISSA means that there is little support for rigorous validation or verification.

1.2.4 DARTS

DARTS [Gom84, Gom86] is a software design method for real-time systems which utilizes the DFD tool. The method can be viewed as an extension to the

SA/SD method which also provides mechanisms for structuring processes into tasks and for defining interfaces between them. The following phases are identified by the method:

- *Data flow analysis* - DFDs are used in DARTS to analyse the functional requirements of an application. This phase utilizes the tools and techniques of SA.
- *Decomposition into tasks* - The processes identified in the data flow analysis phase are structured into concurrent tasks in this phase. Tasks may consist of a single process or a group of processes. Criteria for deciding whether a process can act as a task or can be grouped with other processes to form a single task are provided by Gomaa [Gom84]. The result of this phase is a DFD whose processes are tasks.
- *Defining task interfaces* - Task interfaces determine how tasks communicate with each other, and are defined by two classes of interface modules: the *Task Communication Module* (TCM), which handles all communication between tasks and typically consists of a (concurrently accessed) data structure with access functions; and the *Task Synchronization Module* (TSM), which handles synchronization between tasks.
- *Structured design of tasks* - Each task represents a sequential program, and its design specification is derived by first representing it as a DFD, then transforming the DFD into an initial SC. The transition from DFDs to SCs is carried out in the same manner as in the SA/SD method. DARTS also provides a State Transition Manager (STM) for specifying transaction-oriented applications. The STM module maintains the current state of the application and a state transition table defining legal and illegal state transitions. A task that needs to process a transaction calls the STM with the desired action as input and the STM determines whether the action can be carried out or not, updating the current state if a valid transition is determined.

DARTS extends DFDs by expanding the notion of a data flow to include control signal flow, and allowing constraints on the interface between tasks to be specified. A data flow between two tasks is interpreted in one of the following ways:

1. A loosely coupled message queue containing synchronization mechanisms for suspending generators when the queue becomes full and receivers when the queue is empty. Such an interface is used when two tasks need to pass information to each other, and still proceed at possibly differing speeds.
2. A closely coupled message communication channel on which only one item of data can exist at any time. These channels are modeled as two uni-directional

channels with single-item queue structures. The two channels are orientated in opposite directions, representing the sending of messages and subsequent replies. Such an interface is used when information needs to be passed between two tasks, but the sending task cannot proceed until it has received a reply from the task it has sent information to.

3. an event signal used to notify tasks about event occurrences and does not involve transmission of data.

Interpretations 1 and 2 are defined in terms of a special TCM called a *Message Communication Module* (MCM), while interpretation 3 is defined in terms of the TSM. Interactions with data stores are defined by a special TCM called an *information hiding module* (IHM), which is a data structure with access functions which can be concurrently accessed. DARTS provides special notation for the above types of interfaces, enabling them to be depicted on DFDs.

State dependent behaviour of an application is described in DARTS by a module called a state transition manager (STM). The STM maintains both the current state of the application, as well as a state transition table which defines all legal and illegal state transitions. A task that needs to carry out an action calls the STM with the desired action. The STM then determines whether the action can be carried out given the current state of the system. If the action can be carried out the STM changes the state of the application, if required, and notifies the task that it can carry out the action, otherwise the STM notifies the task that the action cannot be carried out. DARTS provides a central state model for the application. The STM is a data structure.

DARTS, as its name implies, is intended as a design tool, but its use of SA and the techniques used for converting DFDs into diagrams depicting types of communication interfaces, and partitions of processes solves some of the problems associated with the representation of control in DFDs. In particular, the method is especially useful for the specification of applications with complex interactions. Furthermore, the transition to specifications of program structure has the potential to be less contentious than the SA/SD approach, since structured charts are associated with individual tasks, rather than with the entire DFD.

DARTS though seems to lack a formal basis, thus suffers from the problems associated with the lack of such a basis.

1.2.5 Tse's extensions: Formal DFDs (FDFDs)

Tse recognizes the need for a formal framework for the tools of SA, and has carried out a series of studies into possible formalisms ranging from the very abstract (initial algebras) to the more operational (Petri nets) [Tse85a, Tse85b,

Tse86, Tse87, TP86b]. The work of Tse on the abstract representation of the syntactic structure of DFDs [Tse86], in terms of algebraic specifications, is the inspiration for the part of the formal framework presented in this thesis that formalizes the syntactic aspects of DFDs.

Formal DFDs (FDFDs) [Pong86, TP86b] is a language created by Tse and Pong which provides DFDs with a theoretical framework in the form of extended Petri nets. In recognition of the need to preserve understandability, the language has both graphic and symbolic aspects, which allow for the creation of graphical and formal symbolic representations in one-to-one correspondence. The formal symbolic description is needed since graphical descriptions cannot be analysed by computers. The one-to-one correspondence between the two descriptions enables traceability and consistency between the two.

The graphical descriptions take the form of DFDs consisting of only two constructs: data flows, and processes. The symbolic representations take the form of algebraic expressions of Petri nets acting as the formal operational models of the DFDs. A Petri net ([Pete81]) interpretation of a DFD is obtained by viewing processes as transitions, and data flows as places. Tokens placed on data flows mean that data on the data flows are available to the processes needing them. The firing of a process causes the removal of tokens on some of its input data flows and the addition of tokens on some of its outputs data flows. In order to avoid ambiguities that may arise in deciding which inputs may fire and which outputs to place tokens on as a result of a process being fired, the Petri net model is extended with input and output and functions, which are derived from explicit relationships amongst the inputs and outputs of processes expressed in the DFD.

Analysis of FDFDs is carried out using analysis techniques based on the Petri net interpretation of DFDs. Tse identifies three types of analysis:

1. *Global consistency analysis*, which concerns the consistency of the hierarchical structure of DFDs, for example, the directed graph derived as a result of the decomposition of processes should contain no cycles (i.e. recursive decomposition is not allowed);
2. *Structural consistency analysis*, which concerns the input/output relationship between parent and child processes; and
3. *Behavioural consistency analysis*, which concerns the preservation of behavioural properties, as modeled by Petri nets, during decomposition.

Tse and Pong provides algorithms for carrying out the above types of analysis above [TP86b].

While Tse's objectives are similar to the objectives of the formal framework presented in this thesis, his work is carried out at the syntactic level. He does not

provide formal definitions for the data objects in the DFD (in fact data stores are ignored), nor does his work provide support for the formal specification of a process's logic, except in terms of other DFDs. While Petri nets do provide an operational basis from which executable specifications can be derived, one has to be careful about what aspects of the application are actually being made executable. A Petri net essentially provides a simulation of the *control flow* of an application, and are useful tools for representing applications with synchronous interactions. Data objects, and their relationships are not modeled explicitly, while the internal structure of processes are invisible.

Petri nets provide formal operational models, but lack an associated mathematical basis. This limits the use of tools and techniques based on Petri nets in a formal development method.

1.2.6 Extended DFDs (EXT-DFDs)

Petri nets are also used as an operational basis for the visual language extensions to DFDs provided by *extended data flow diagram* (EXT-DFD) [CTL87]. The primary objectives of EXT-DFD are to provide a non-procedural, easy to use, graphical environment, with high processing power, for creating and validating DFDs. The visual aspects of EXT-DFD consist of DFDs, and *entity-relation* (ER) graphs which are transformed into user-interface forms for specifying database manipulation. The symbols of EXT-DFD are icons, associated with properties, which may be composed of other (sub) icons. Icons communicate by message passing interpreted as data flow. There are four types of icons in EXT-DFDs:

1. *Object icons* represent entities with an associated set of operations or actions. Data store icons are classed as object icons.
2. *Action icons* represent a specific operation (action) of the system. Process icons are classed as action icons. Forms are also action icons which manipulate the database. An action icon is associated with input and output data icons (see 3. below). It acts on the input data icons to produce the output data icons. The behaviour of an action icon acts as its semantics.
3. *Data icons* represent data flow elements. They define the data type of a data flow, and may contain a value during execution of the EXT-DFD.
4. *External icons* represent components lying outside the scope of an application, that is the external entities of a DFD. They are used to initiate and terminate the execution of EXT-DFDs. No processing is carried out by these icons.

DFDs are viewed, in EXT-DFDs, as being composed of separate, interacting components, where each component has its own state which may change over time. The components can exhibit concurrency, and may require synchronization of

constituent actions. Petri nets are used as the operational basis because of their demonstrated usefulness in the modelling, analysing, and simulation of such concurrent components.

When viewed as Petri nets the data flows, data stores and external entities of an EXT-DFD are treated as places, while the processes (action icons) are treated as transitions. Three types of Petri net places are distinguished: *initial/terminal*, *store*, and *data* places, corresponding to external entities, data stores, and data flows. Tokens, representing the presence or absence of data, are stored in places. Initial places initiate the execution of EXT-DFDs, given an external command to "run" the DFD. Terminal places terminate the execution of a DFD in an execution path. Rules governing the firing of transitions, and their effects on the tokens in places are also provided by the operational model [CTL87].

In order to support the decomposition of DFDs *grouped transitions* are used. A *grouped transition* is an abstraction of a Petri net model. Grouped transitions correspond to non-primitive processes, while *single transitions* correspond to primitive processes.

EXT-DFDs are validated by analysing their components. Each component must satisfy a set of rules, called *integrity constraints*, which determine how components can be related.

EXT-DFDs, improve upon FDFDs by incorporating abstraction concepts for structuring complexity in their modified Petri nets, and by considering data stores. EXT-DFDs also have the potential of further enhancing communication amongst developers and users through their use of a graphical language. The executable nature of EXT-DFDs also makes them potentially useful for validating behaviour with users, though, as pointed out, only the aspects related to the types of interactions between components are actually demonstrated.

EXT-DFDs, while having a formal operational model, lack an associated mathematical basis thus limiting their use in formal development methods.

1.3 Conclusion

The methods reviewed above provide extensions to DFDs to alleviate problems associated with various aspects of their use. Some of the methods provide extensions to the descriptive power of the tools while few seek to provide formal operational frameworks for them. The need to provide a formal basis for the tools of SA is evident in the lack of automated tools for checking not only the syntactic aspects of the generated specifications, but the semantic, or behavioural aspects captured by the specifications. Formal frameworks which associate formal operational models interpretations for DFDs in terms of Petri nets can be seen as a

first step towards providing a basis for DFDs. Such operational models are useful for rigorously validating specifications with users. Petri nets, though, are not associated with any mathematical foundations, and this limits their use in formal development methods in which it is required that implementations be proven against the requirements specification.

CHAPTER 2

Syntactic and Semantic Aspects of Data Flow Diagrams

2.0 Introduction

This chapter serves as an informal introduction to the syntactic and semantic aspects of DFDs which are formalized in later chapters of this thesis. The syntactic aspects of DFDs are concerned with the building of their syntactic objects while the semantic aspects are concerned with the behavioural interpretations associated with the syntactic objects.

2.1 A computer-based library application example

This section introduces the example which will be used to illustrate the concepts and techniques used in this and other chapters. It is based on a problem set for the Fourth International Workshop on Software Specification and Design [SSD87].

The requirements for a university computer-based library application, in terms of the basic actions it is required to support, are as follows¹ :

- A1. Add and remove copies of books to and from the library.
- A2. Borrow and return books.
- A3. Update borrower's record on full or part payment of fines.
- A4. Add and delete borrowers.

The above actions can only be invoked by the library staff. There are three types of borrowers: *undergraduates*, *postgraduates*, and *academic staff*. There are also three types of books: *references*, *general (access) books*, and *periodicals*. References cannot be checked out. Periodicals can be checked out only by postgraduates and academic staff, and only for a period of two weeks. General books can be checked out by any type of borrower for the following periods: undergraduates - 2 weeks, postgraduates - 4 weeks, academic staff - 6 weeks.

A fine of 20c per day is incurred for overdue books. The fine is accumulated starting from the day after the book is to be returned until the book is actually returned. If a borrower's accumulated fines for all books exceeds \$30.00 the borrower is barred from checking out books until the fine is reduced to \$30.00 or less.

¹References to the library are actually references to the library data base.

The following constraints must also be satisfied by the library application:

- A. All borrowable copies in the library must be available for checkout, or be checked out.
- B. No copy of a book may be both checked out and available at the same time.

2.2 Syntactic aspects of DFDs

The syntactic aspects of DFDs are characterized by the formal framework in terms of abstract syntactic objects. Abstract here means *representation independent*, that is, no particular concrete representations are implied by the definitions of the objects. This is to allow various graphical and textual representations to be used in conjunction with the formal framework. The abstract objects are defined in terms of their attributes and are associated with formation rules which characterize their structure. Syntactic objects which adhere to their formation rules are said to be *structurally correct*. Figure 2.1 shows the graphical representations of the basic syntactic structures (constructs) of DFDs used in this thesis.

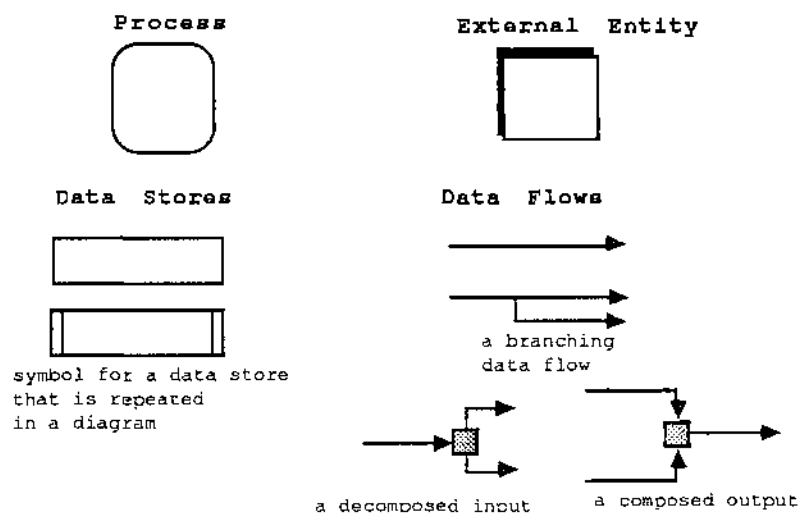
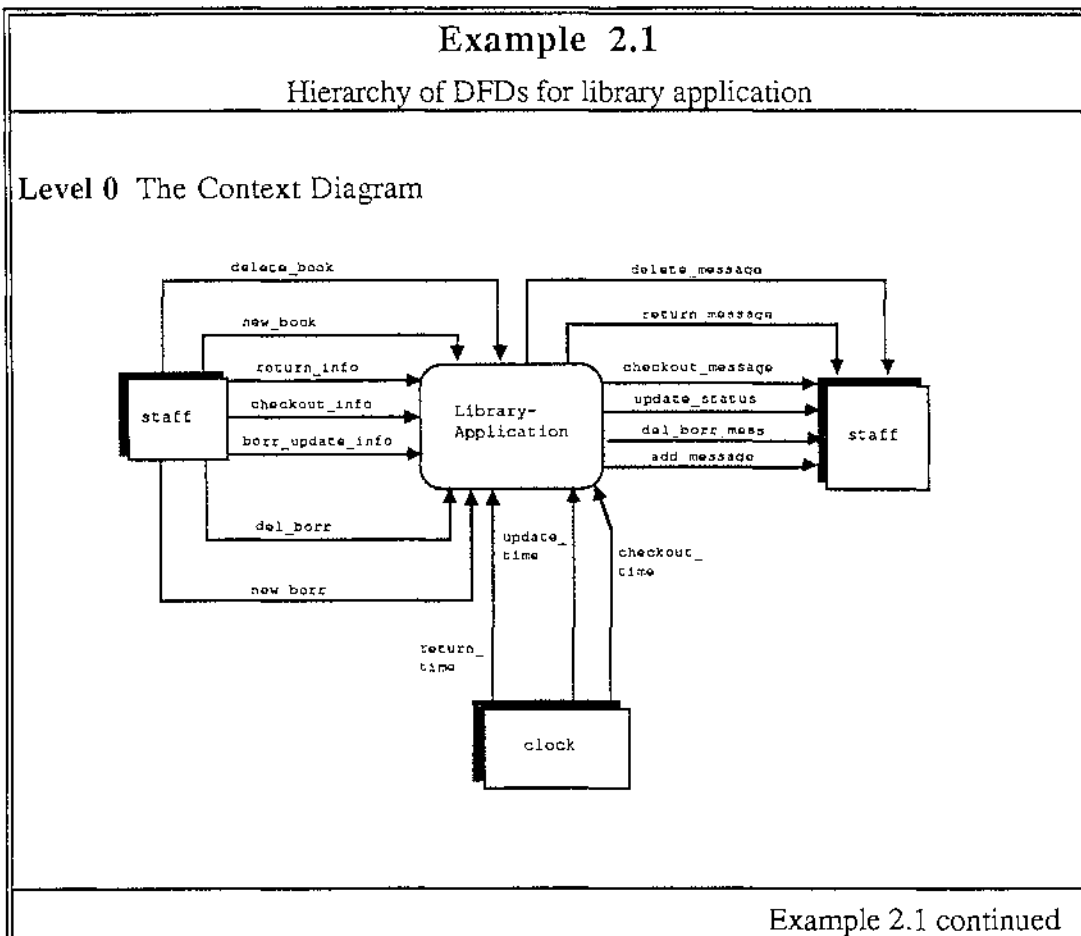


Figure 2.1 Graphical representations of DFD constructs

In the SA approach, applications are represented by a hierarchy of DFDs, made up of a top level DFD called the *context diagram* consisting of a single process, a bottom level consisting of DFDs of primitive processes, and intermediary levels consisting of DFDs describing the processes and data flows at the higher levels in more detail ([DeM78], see also Chapter 1). Such a hierarchy of DFDs can alternatively be viewed in terms of a *process hierarchy*, where the top level consists of the single process in the context diagram, and the bottom and intermediary levels are the bottom and intermediary levels of the hierarchy of DFDs. Processes at the intermediary levels of a hierarchy of DFDs can similarly be viewed

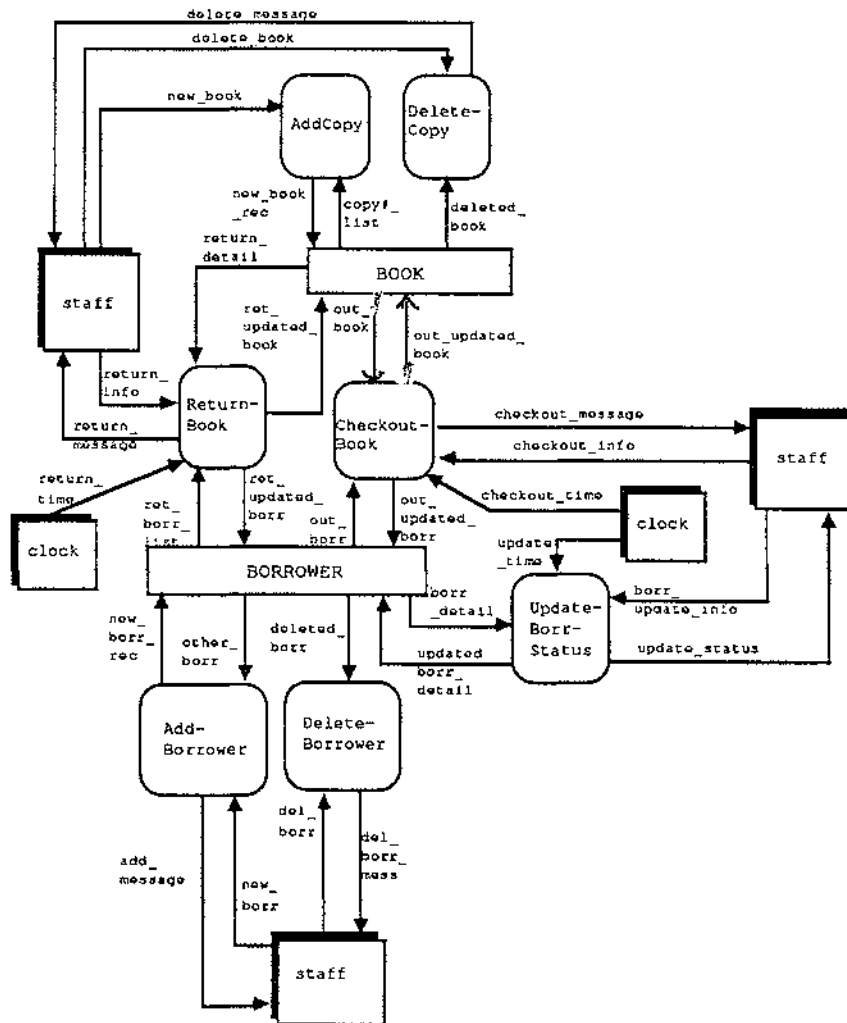
as hierarchical structures. Also, data flow hierarchies can be associated with a hierarchy of DFDs in which process decomposition involves parallel data flow decomposition. The structures for decomposed inputs and composed outputs shown in Figure 2.1 are used to depict the decomposition of data flows. The processes and data flows in a hierarchy of DFDs are syntactically treated as hierarchical objects with structures conforming to rules governing the decomposition of processes and data flows provided by the formal framework. When the hierarchical nature of processes and data flows in a DFD are to be ignored, the DFD and its components are referred to as *flat*.

Example 2.1 shows a hierarchy of DFDs representing the library application.



Example 2.1 (continued)
Hierarchy of DFDs for library application

Level 1



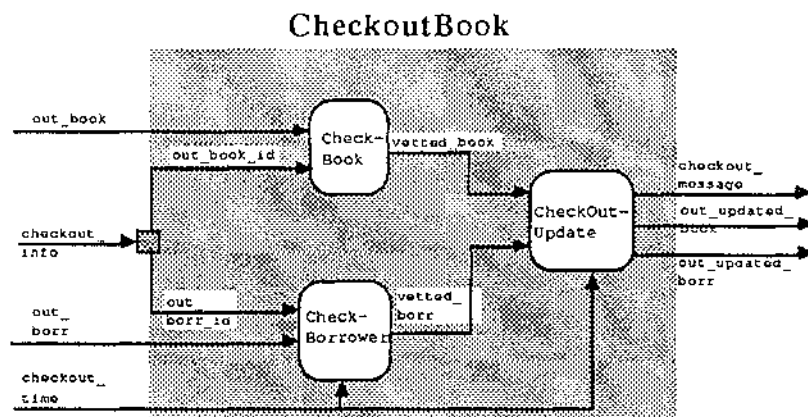
In the Level 1 DFD the data store **BORROWER** contains information about borrowers for example, personal details and fines paid, and information accessed via the borrower, for example, details of books borrowed. Similarly the data store **BOOK** contains information about copies of books, such as copy details (authors, title, etc.), and a borrower flag indicating whether a book is available or not. The clock external entity provides the current time to processes needing it.

Example 2.1 continued

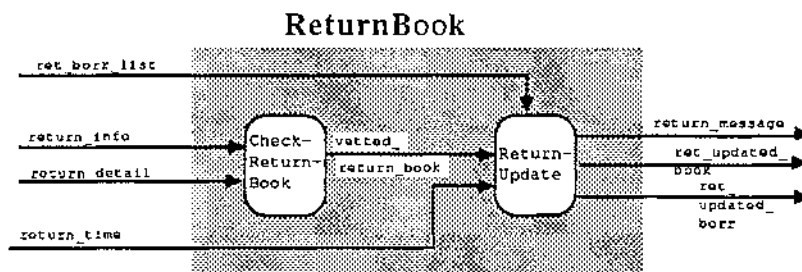
Example 2.1 (continued)

Hierarchy of DFDs for library application

Level 2



CheckoutBook is decomposed into three processes: CheckBook determines whether the book to be checked out can be checked out; CheckBorrower determines whether the borrower is permitted to borrow any books; and CheckOutUpdate updates the BOOK and BORROWER, provided that the check out is possible, and generates a check out message indicating the status of the check out action.

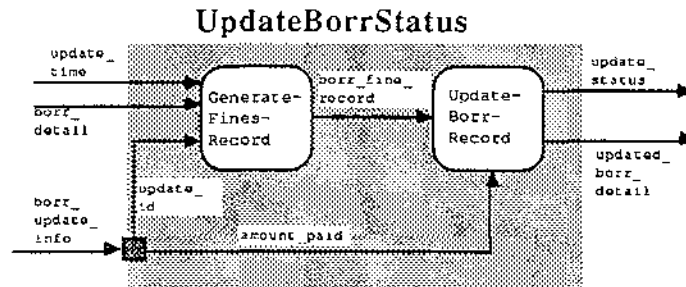


ReturnBook is decomposed into two processes: CheckReturnBook determines whether the book is a library copy; ReturnUpdate updates BORROWER and BOOK, if the book is a copy of the library, and generates a message indicating the status of the return book action.

Example 2.1 continued

Example 2.1 (continued)

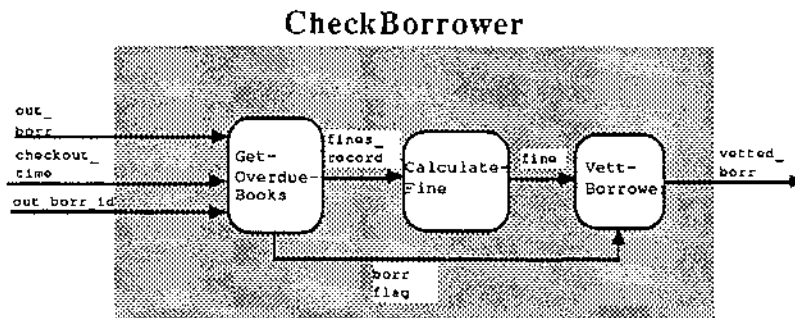
Hierarchy of DFDs for library application



UpdateBorrStatus is decomposed into two processes:

GenerateFinesRecord calculates the fine on each overdue book possessed by a borrower; UpdateBorrRecord updates BORROWER with the amount paid.

Level 3



The level 2 process CheckBorrower consists of three processes:

GetOverdueBooks calculates the fines for each overdue book possessed by the borrower; CalculateFine sums the fines; VettBorrower determines whether the borrower is permitted to borrow books.

In what follows the syntactic aspects of flat DFDs are described and then extended to incorporate concepts related to the decomposition of processes and data flows. The result is a set of abstract syntactic objects which encapsulate the syntactic aspects of hierarchies of DFDs. Formalization of the syntactic aspects of hierarchies of DFDs is thus achieved by providing formal definitions of the abstract objects introduced here, which is done in Chapter 4.

2.2.1 Syntactic aspects of flat DFDs

Syntactically, a flat DFD is an abstract object consisting of an *external environment* (EE) and a *process structure* (PS). The EE of a flat DFD consists of external entities with their associated data flows, while the PS of a flat DFD consists of processes and data stores with their associated data flows. Syntactically, data flows have two attributes: a name and a direction. In what follows a data flow directed towards a construct is called an *input* of the construct, while a data flow directed away from a construct is called an *output* of the construct. Processes, data stores and external entities have the following syntactic attributes: a set of inputs, and a set of outputs.

The following are the formation rules characterizing the syntactic aspects of flat DFDs.

Definition 2.1

Characterizing the syntactic aspects of DFDs

Processes

F1. A structurally correct process has a non-empty set of inputs, and a non-empty set of outputs. Furthermore, the set of inputs and the set of outputs are disjoint, that is, an input of a process cannot be an output of the same process.

Data stores

F2. A structurally correct data store has a non-empty set of inputs or a non-empty set of outputs. Its set of inputs and set of outputs are also disjoint.

Process structures

F3. A structurally correct process structure has at least one process. All processes in a structurally correct process structure are structurally correct and are uniquely identified by their inputs and outputs.

F4. All data stores in a structurally correct process structure are structurally correct. All the inputs of a data store in a structurally correct process structure are also outputs of processes in the process structure. Also all the outputs of a data store must also be inputs of processes in the process structure.

Furthermore, the set of data flows (inputs and outputs) of a data store in a structurally correct process structure is disjoint from the set of data flows of any other data store in the process structure. This means that data stores are not directly connected by data flows in a structurally correct process structure. Data stores in a structurally correct process structure are uniquely identified by their inputs and outputs.

Definition 2.1 continued

Definition 2.1 (continued)

Characterizing the syntactic aspects of DFDs

- F5. An output of a process in a structurally correct process structure is either associated with another process and/or data store in the process structure as an input, or is not associated with any process or data store in the process structure. An input of a process in a structurally correct process structure, on the other hand, may be associated with more than one process and/or data store in a process structure as an input.
- F6. The *net* or *boundary inputs* of a process structure are the inputs associated with the processes and data stores in the process structure which are not also outputs of processes and data stores in the process structure. A structurally correct process structure has at least one net input.

External entities

- F7. A structurally correct external entity has a non-empty set of inputs or a non-empty set of outputs. Its set of inputs and the set of outputs are also disjoint.

External environments (EEs)

- F8. A structurally correct EE consists only of structurally correct external entities. Furthermore, there is at least one external entity in the EE with a non-empty set of inputs, and at least one external entity with a non-empty set of outputs. All external entities in a structurally correct EE are uniquely identified by the set of their data flows.
- F9. An input of an external entity in a structurally correct EE is never an output of another external entity in the EE. An input, on the other hand may be associated with more than one external entity in the EE as an input, provided that it is not also an output of an external entity in the EE. The sets of outputs associated with the external entities in a structurally correct EE are all disjoint.

Flat DFDs

A flat DFD consists of a structurally correct process structure and a structurally correct EE (possibly empty) satisfying the following rule:

- F10. The set of all outputs in the EE is equal to the set of the net inputs of the process structure. Also, the set of all inputs in the EE is a subset of the set of all outputs in the process structure. For a DFD with a non-empty EE the result is that each data flow in the DFD is associated with a unique generator, and a non-empty set of receivers.

Level 1 of the hierarchy of DFDs shown in Example 2.1 can be viewed as a flat DFD by ignoring the lower levels of the hierarchy. The EE of this DFD consists of the structurally correct external entities `staff` and `clock`, while the process structure consists of the structurally correct processes `AddCopy`, `DeleteCopy`, `ReturnBook`, `CheckoutBook`, `UpdateBorrStatus`, `AddBorrower`, and `DeleteBorrower`. In the EE, no two external entities are directly connected to each other, and all outputs are unique. Furthermore, the set of inputs to, and the set of outputs from the EE are non-empty. The EE is thus structurally correct. Within the process structure, the data store inputs (outputs) are all associated with process outputs (inputs). Furthermore, all outputs in the process structure are associated with unique generators. The net inputs of the process structure, `new_book`, `delete_book`, `checkout_info`, `checkout_time`, `return_time`, `return_info`, `new_borr`, `del_borr`, `borr_update_info`, and `update_time`, are exactly the outputs of the EE, also the set of inputs of the EE is a subset of the set of outputs of the process structure. The process structure of the DFD and the DFD itself are thus structurally correct.

2.2.2 Syntactic aspects of hierarchies of DFDs

Hierarchies of DFDs are syntactically treated as objects consisting of hierarchical representations of processes and data flows. The objects can be viewed as extensions of the syntactic objects representing flat DFDs, characterized in the previous section, where the extensions take the form of additional attributes reflecting the hierarchical nature of processes and data flows. Processes and data flows so extended are referred to as *hierarchical*. Formation rules for hierarchical data flows and processes are given below.

Hierarchical data flows

Decomposition of data flows in a DFD results in the revelation of their component data flows. The component data flows so obtained either *fully define* the data flow, in which case they are all necessary and sufficient components of the data flow, or they may *partially define* the data flow, in which case they may not be sufficient to fully define the data flow. Decomposition of data flows whose structures consist of alternative components is not permitted here. Also, recursively defined data flows are not permitted, since such definitions are not consistent with the view of decomposition as an activity which results in the revelation of detail not provided at the previous level. These restrictions enable the representation of hierarchies resulting from successive decomposition of data flows, as tree structures, in which nodes are data flows and edges represent the "is a component

of" relationship between data flows. Such trees are called *data flow trees*. The data flow at the root of a data flow tree is called the *root data flow*, while the data flows with no components (i.e. the leaf, or bottom level, nodes of the tree) are called *primitive* data flows. Semantically, primitive data flows are associated with data types (with possibly alternative structures), which can then be used to generate the composite data types for the higher level data flows. An example of a data flow tree is given in Example 2.2.

The syntactic aspects associated with the decomposition of data flows are captured by an object called a *hierarchical data flow*, with the following attributes:

- a name, and
- a set of hierarchical data flows, called the *child decomposition set* of the hierarchical data flow, representing the structure of the hierarchical data flow.

Data flow trees are concrete representations of hierarchical data flows. The name of a hierarchical data flow is the name of the data flow at the root of its data flow tree representation, while its child decomposition set is the set of sub trees whose roots have an edge connecting them to the root data flow. For example the child decomposition set of the hierarchical data flow shown in Example 2.2 are the sub trees with roots `book_id` and `borrow_id`. The nodes of a data flow tree are called the *sub data flows* of the hierarchical data flow, while the sub trees whose set of leaf nodes are a subset of the set of leaf nodes of the data flow tree are called the *sub hierarchical data flows* of the hierarchical data flow. The rule characterizing structurally correct hierarchical data flows, given in Definition 2.2, is essentially a rule for building data flow tree structures.

Definition 2.2

Characterizing structurally correct hierarchical data flows

Each sub data flow of a structurally correct hierarchical data flow is unique.

In order to express the relationship between a data flow at a particular level of a hierarchy of DFDs and its decomposed data flows at the next level, the notions of *full* and *partial decomposition sets* are needed.

Definition 2.3

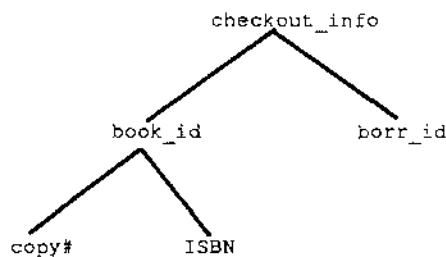
Full and partial decomposition sets

A *full decomposition set*, F , of a hierarchical data flow D , is a set of sub hierarchical data flows of D , satisfying the following conditions:

1. no two hierarchical data flows in F have common sub data flows; and
2. the set of all primitive data flows in the hierarchical data flows in F is equal to the set of primitive data flows in D .

A *partial decomposition set* of a hierarchical data flow is simply a subset of its sub hierarchical data flows.

A full decomposition set of a data flow fully defines the data flow. Examples of full and partial decomposition sets can be found in Example 2.2.

Example 2.2Data flow tree for the hierarchical data flow `checkout_info`

The above data flow tree is a concrete representation of the hierarchical data flow `checkout_info`. The child decomposition set of `checkout_info` is the set consisting of the hierarchical data flows `book_id` and `borr_id`. The set consisting only of the hierarchical data flows `book_id` and `borr_id` is a *full decomposition set* of `checkout_info`, so also is the set consisting of the hierarchical data flows `copy#`, `ISBN`, and `borr_id`. The set consisting only of the data flows `book_id` and `copy#` is a *partial decomposition set* of `book_id` as well as of `checkout_info`.

In describing the syntactic aspects of hierarchies of DFDs, the use of hierarchical data flows parallels the use of flat data flows in describing flat DFDs. External entities and data stores associated with hierarchical data flows as inputs and outputs are referred to as *hierarchical*. Processes are also associated with hierarchical data flows in a manner described in the next sub section.

In a hierarchy of DFDs all data flows are uniquely named (see Definition 2.2). To express this uniqueness property, the notions of a *distinguished pair* and *set* of hierarchical data flows are used.

Definition 2.4

Distinguished sets of hierarchical data flows

Two hierarchical data flows are said to be *distinguished* if they do not have any common sub data flows. A set of hierarchical data flows in which every pair is distinguished is called a *distinguished set*.

Hierarchical processes

The syntactic aspects of a hierarchy of processes, resulting from successive process decomposition, are concerned mainly with the relationships between high level processes and their more detailed description at the lower levels. Such aspects are captured by an abstract syntactic object called a *hierarchical process*, with the following attributes.

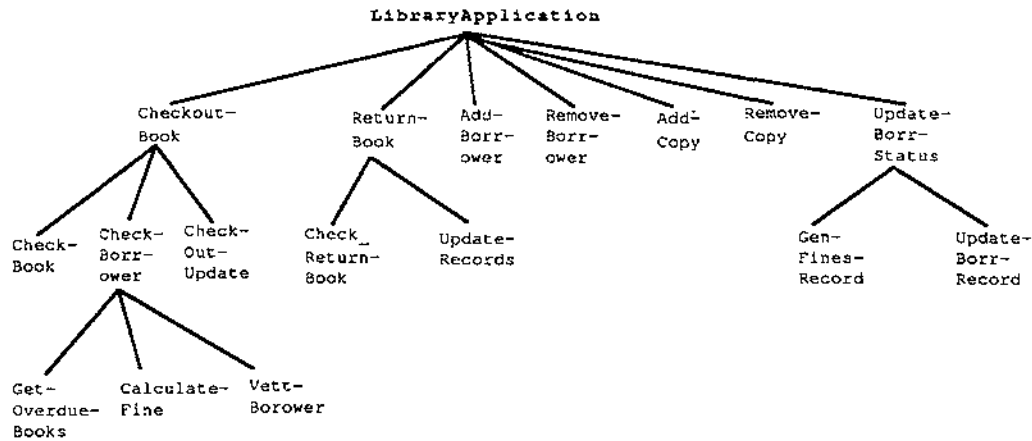
- a set of hierarchical data flows, called the *inputs* of the hierarchical process;
- a set of hierarchical data flows, called the *outputs* of the hierarchical process;
- and
- a structure of (sub) hierarchical processes and hierarchical data stores, called the *body* of the hierarchical process.

The body of a hierarchical process represents the hierarchical structure of the process arising from successive decompositions. As in hierarchies of data flows, recursive descriptions of processes are not permitted. Successive decomposition of a process can be concretely represented by a tree. In such a tree, here called a *process tree*, the nodes are flat processes (with their inputs and outputs), and the edges represent the "is a sub process of" relationship between processes. Each level of the tree is associated with a set of data stores, where the set of data stores at a particular level is disjoint from any other set associated with the other levels of the tree. The root of a process tree is called the *root process*, while the processes at the bottom level are called *primitive*. Primitive processes have empty bodies. The nodes of a process tree are called the *sub processes* of the corresponding hierarchical process, while the sub trees of a process tree whose leaf nodes are subsets of the set of leaf nodes of the process tree represent sub hierarchical processes of the hierarchical process corresponding to the process tree. An example of a process tree is given in Example 2.3 (data stores are not shown in the example).

Example 2.3

Process tree for the library application

The hierarchy of DFDs shown in Example 2.1 can be viewed as a tree of processes as is illustrated below:



The hierarchical process corresponding to CheckoutBook is the sub tree of the above tree with CheckoutBook as its root. The *body* of CheckoutBook, thus consists of the hierarchical processes CheckBook, CheckBorrower, and CheckOutUpdate, where both CheckBook and CheckOutUpdate are *primitive*, and their associated data stores (not shown in the tree diagram).

The following definitions are needed to express the rules characterizing structurally correct hierarchical processes. The *net inputs* of a hierarchical process are the inputs in its body which are not also outputs in the body. For example, the net inputs of the hierarchical process CheckoutBook are the data flows `out_book`, `out_book_id`, `out_borr_id`, `checkout_time`, and `out_borr`. The set of all inputs (outputs) in the body of a hierarchical process is called the *internal input (output) set* of the body. For example, the internal output set of the hierarchical process CheckoutBook is `{vetted_book, vetted_borr, checkout_message, out_updated_book, out_updated_borr}`

Definition 2.5 gives the rules characterizing structurally correct hierarchical processes.

Definition 2.5

Characterizing structurally correct hierarchical processes

Hierarchical data stores

P1. A structurally correct (hierarchical) data store has a non-empty set of hierarchical inputs or a non-empty set of hierarchical outputs. The union of inputs and outputs of a data store is a distinguished set.

The body

P2. A structurally correct body is either empty or contains at least one structurally correct (sub) hierarchical process. All data stores in a body are structurally correct.

P3. No two hierarchical processes in a structurally correct body must have common sub processes.

P4. The set of all data store inputs in a structurally correct body is a subset of the internal output set of the body, and the set of all data store outputs is a subset of the internal input set of the body. Furthermore, the receiver of a hierarchical data flow whose generator is a data store is never a data store.

P5. Each hierarchical data flow in the internal output set has a unique generator in the body. The internal output set of a structurally correct body is a distinguished set.

P6. There is at least one net input in a non-empty structurally correct body.

Hierarchical processes

P7. The set of inputs and the set of outputs of a structurally correct hierarchical process are both non-empty. Furthermore, the union of the inputs and the outputs of a hierarchical process is a distinguished set.

P8. The body of a structurally correct hierarchical process is structurally correct. In a structurally correct hierarchical process with a non-empty body, an input corresponds to a subset of the net inputs in the body, called its *decomposition set*, which is a partial decomposition set of the input. The decomposition sets of any two hierarchical data flows in the input interface are disjoint, and the union of the decomposition sets associated with the inputs of the hierarchical process is exactly the set of the net inputs of the body.

Definition 2.5 continued

Definition 2.5 (continued)

Characterizing structurally correct hierarchical processes

- P9. For a structurally correct hierarchical process with a non-empty body, an output corresponds to a subset of the internal output set, called its *decomposition set*, which is a full decomposition set of the output. The decomposition sets of any two outputs is disjoint. If a hierarchical data flow in the internal output set of the body of a structurally correct hierarchical process is not in any decomposition set then it is directed towards hierarchical processes in the body.

The rules determining the structure of the body of a hierarchical process can be viewed as extensions of the rules characterizing the PSs of flat DFDs, which take into consideration the hierarchical nature of data flows and processes. Similarly, the rules for hierarchical processes can be viewed as extensions to the rules characterizing structurally correct flat processes.

Hierarchical DFDs

The syntactic aspects of a hierarchy of DFDs are captured by an abstract syntactic object called a *hierarchical DFD* (H_DFD). A structurally correct H_DFD consists of a structurally correct *hierarchical process* and a structurally correct *external environment* (EE), where the EE of a H_DFD is a set of external entities with hierarchical inputs and outputs. The hierarchical process of a H_DFD represents the hierarchy of DFDs resulting from successive process decompositions, and can be viewed as a hierarchical representation of the single process in the context diagram of the corresponding hierarchy of DFDs. The rules characterizing structurally correct H_DFDs, given in Definition 2.6, can be viewed as extensions of the rules characterizing structurally correct flat DFDs.

The abstract syntactic objects introduced above capture the desired syntactic aspects associated with hierarchies of DFDs. Formalizing the above definitions for the abstract objects results in the formalization of the syntactic aspects they capture, thus providing a basis for validating the syntactic structure of (hierarchies of) DFDs. Chapter 4 provides the formal counterparts of the rules characterizing structurally correct objects stated in this chapter.

Definition 2.6

Characterizing structurally correct hierarchical DFDs

Hierarchical external entities

H1. A structurally correct external entity has a non-empty set of inputs or a non-empty set of outputs. Also, the sets of inputs and outputs are disjoint, and their union is a distinguished set.

External environments

H2. A structurally correct EE consists of structurally correct external entities or no external entities. Furthermore, there is at least one external entity in the EE with a non-empty set of inputs, and at least one external entity in the EE with a non-empty set of outputs. All external entities in a structurally correct EE are uniquely identified by their inputs and outputs.

H3. An input of an external entity in a structurally correct EE is never an output of another external entity in the EE. The sets of outputs of any two external entities in a structurally correct EE are disjoint and are distinguished sets.

H_DFDs

H4. A structurally correct H_DFD consists of a structurally correct EE and a structurally correct hierarchical process. The set of all inputs (outputs) in the EE of a structurally correct H_DFD is equal to the set of inputs (outputs) of the hierarchical process of the H_DFD.

In the remainder of this chapter the semantic aspects of DFDs are introduced in an informal setting. Concrete representations of the abstract objects described above will be used for illustration purposes in what follows.

2.3 Semantic aspects of DFDs

In the SA/SD approach a data dictionary contains definitions of the structure and content of the data flows and data stores in a hierarchy of DFDs, while process specifications describe the functional behaviour of its primitive processes. The definitions provided by the data dictionary and process specifications are quasi-formal, and thus provide little support for the rigorous validation and verification of behaviour. Furthermore, the transition from SA specifications to an initial design is problematic ([Sho88, Pet88], see also Chapter 1). Formal interpretations of the syntactic structures in DFDs facilitate the derivation of formal specifications of behaviour for DFDs, which can be viewed as initial designs of the applications described by the DFDs.

The semantic aspects of hierarchical DFDs concern the interpretations associated with their syntactic structures. Two types of semantic aspects are

specifiable in the formal framework: *behavioural* and *data*. Behaviour refers to the manner in which components interact with each other. Two aspects of behaviour are emphasized: *functional* and *control*. Functional aspects concern the relationship between the input values and output values of a process while control aspects concern the time-related interactions between processes. Specifications of the functional aspects of a DFD are supported by specifications of its *data aspects*, which are concerned with the structure of the data objects in the DFD.

To support the specification of the behavioural aspects of applications additional constructs for describing interactions in an application which cannot be described using the traditional DFD constructs are introduced. Diagrams created using these additional constructs are called *Extended DFDs* (ExtDFDs). In the formal framework, the behaviour of an ExtDFD is characterized by all the possible interactions that can take place amongst its components. Such interactions are determined by the occurrences of events which may or may not have data associated with them. The characterization is expressed as a formal specification derivable from the ExtDFD. The derivation of the formal specification of behaviour from a hierarchy of DFDs goes through the following steps:

1. Generating a flat representation of the hierarchy of DFDs. Such a representation, called the *primitive DFD*, consists of the primitive processes, and all the data stores and external entities in the hierarchy of DFDs.
2. Introducing notation for describing state dependent behaviour into the primitive DFD, specifying the state dependent behaviour, and identifying actions, and state and asynchronous data flows to and from the external environment (EE). The result of this step is an ExtDFD.
3. Specifying the data types associated with the ExtDFD's data flows and data stores.
4. Specifying the behaviour of the ExtDFD's primitive processes and data stores.
5. Deriving the specifications of behaviours of the ExtDFD's actions from the specifications of behaviours of their constituent processes.
6. Deriving the specification of behaviour of the ExtDFD from the specifications of behaviour of its actions, data stores, and asynchronous data flows, and the specification of its state dependent behaviour. The resulting specification is called the Behavioural Specification (BS).

An overview of these steps is given in the following sections.

2.3.1 Flattening hierarchies of DFDs

In deriving a formal specification of behaviour from a hierarchy of DFDs it is sufficient to consider the interactions amongst its primitive processes and data

stores, and their interactions with the EE. The structure consisting only of the primitive processes, data stores and EE of a hierarchy of DFDs is called the *primitive DFD* of the hierarchy, and can be viewed as a flat representation of the hierarchy.

The data flow relationships between structures in a primitive DFD are not simple, since data flows associated with structures in one part of the DFD may be decomposed in other parts of it. The relationships between such data flows are depicted by *splitter* and *binder* symbols, shown in Figure 2.2. A splitter takes an incoming data flow, called its *input*, and generates a subset of its sub data flows, called the *outputs* of the splitter. A binder takes a set of incoming data flows, called its *inputs*, and combines them to form a single outgoing data flow, called its *output*. The input of a splitter may emanate from a binder, data store, external entity, or a process, while its outputs can be directed to processes and or binders. The inputs of binders may emanate from splitters and/or processes while its output may be directed towards processes, data stores, external entities, and/or splitters. Later it will be shown how splitters and binders force their associated processes to synchronize their receive and send events.

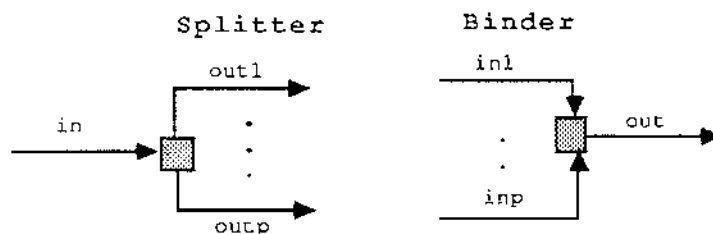
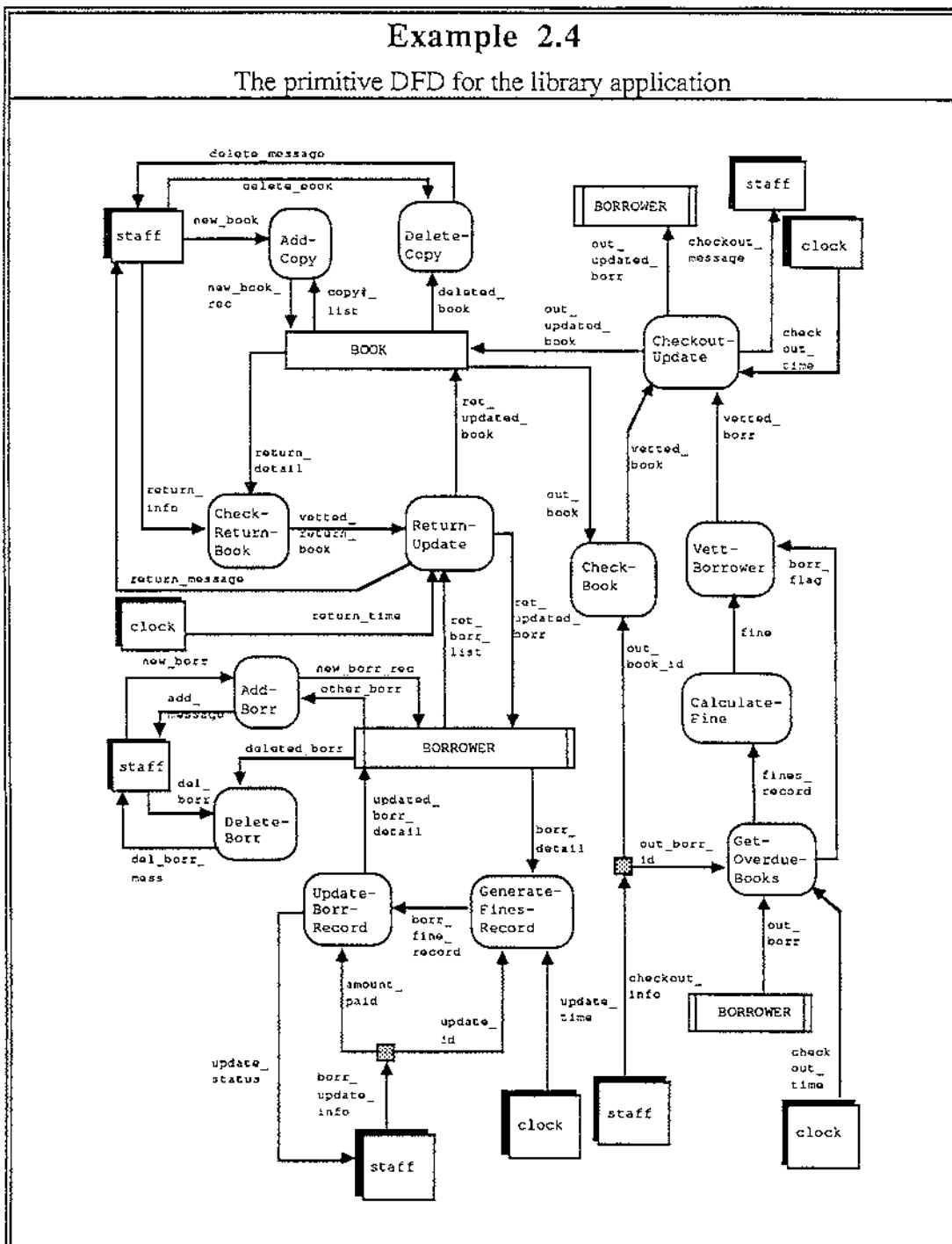


Figure 2.2 Splitter and binder symbols

Example 2.4 shows the primitive DFD for the H_DFD for the library application depicted in Example 2.1.



Henceforth, the term process will refer to a primitive process, and the term DFD refers to the primitive DFD.

2.3.2 Describing the control aspects of applications

Situations in which the required behaviour of an application is dependent on the current state, or mode of operation, of the application often occur in certain types of applications, for example real-time applications. Additional notation is

required in order to describe state dependent behaviour with DFDs. Two types of constructs are used here in this respect: a *state entity* and *control flows*. The state entity encapsulates information about the current state of the DFD affecting the behaviour of the application, while control flows represent the events which cause changes in the mode of operation. Syntactically, a state entity has the following attributes: a name, a set of control flows called the *inputs* of the state entity, and another set of control flows called the *outputs* of the state entity. Semantically, a state entity can be viewed as an *interpreter* of events represented by its inputs, which may *generate* other events, represented by its outputs, as a result of interpretations. A state entity can affect the behaviour of processes via its outputs, in three ways:

- It can *enable* processes. An enabled process is permitted to transform its inputs to outputs when required to do so.
- It can *disable* processes. A disabled process is not allowed to transform its inputs to outputs.
- It can *initiate* processes. A process that is initiated is enabled for only a single transformation after which it disables itself.

Control flows are either directed from external entities or processes to a state entity or to other processes, or are directed from a state entity to processes. A control flow, like a data flow, may be directed towards more than one construct, called the *receivers* of the control flow, but emanate only from a single construct, called the *generator* of the control flow. Control flows differ from data flows in that they represent events which are not associated with data. Control flows generated by external entities and processes are called *signals*. Control flows from the state entity to processes are associated with one of the following type of events, reflecting the manner in which the state entity can affect the behaviour of processes.

- *Enablers*: events which enable processes.
- *Disablers*: events which disable processes.
- *Initiators*: events which initiate processes.

Enablers, disablers, and initiators are events extended in time, whose occurrences last for the periods of time that the associated processes are enabled, disabled, and initiated respectively. Control flows emanating from other than the state entity can also be input to processes. Such control flows depict events that affect only the behaviour of the process, and behave as initiators.

The constructs depicting control information are shown in Figure 2.3.

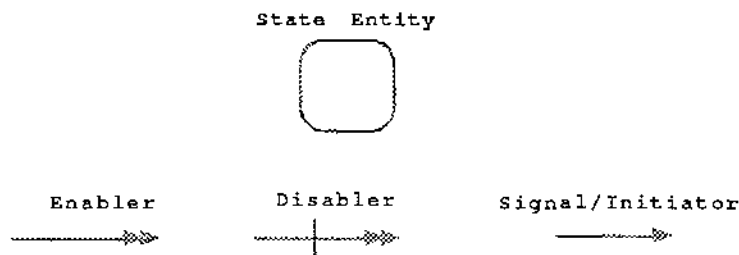


Figure 2.3 State entity and control flow symbols

The approach described above is similar to that used in YSM for describing state dependent behaviour [War86, Woo88]. The state entity corresponds to the control process in YSM, while control flows play similar roles in both approaches. As in YSM, the state dependent behaviour of an application can be described by a state diagram, associated with the state entity, defining the manner in which the state entity interprets its inputs. The differences between YSM and the SL lie in their use of the control extended DFDs. Here, such DFDs are associated with a theoretical basis enabling the derivation of formal specifications characterizing the class of behavioural models for the DFDs, while in YSM, the diagrams, together with their associated state diagrams, are used as *descriptions* of an application's behaviour. The control extended DFDs used in this thesis can be viewed as informal, pictorial representations of the derived formal specifications. An example of a control extended DFD is given in Example 2.5.

Example 2.5

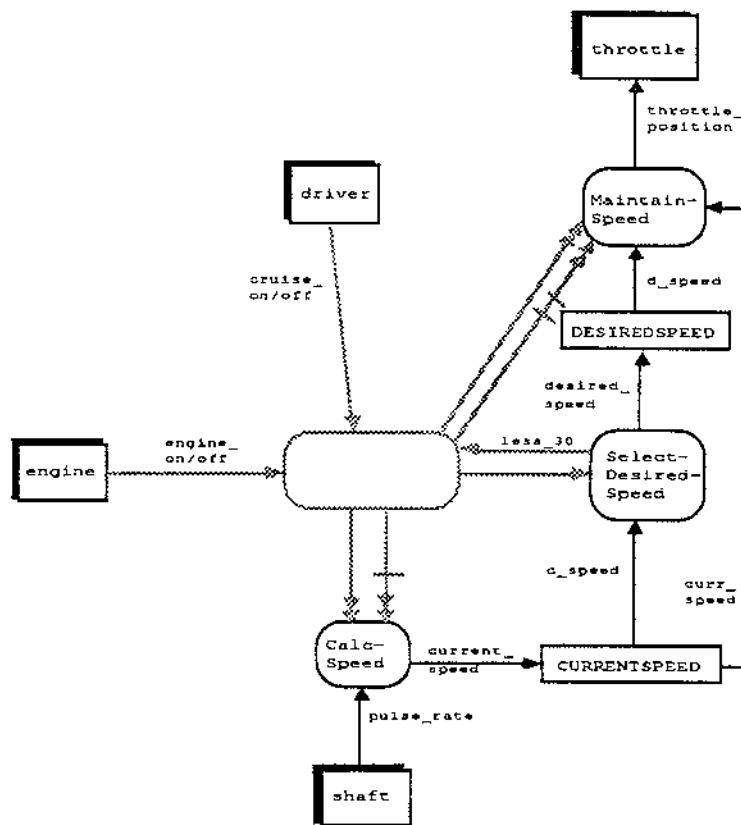
A control-extended DFD for a cruise-control system

A cruise-control system, when active, maintains the speed of a vehicle at a constant level. In the system, depicted below, the driver sends signals to the system which activates and deactivates it. The system can only be activated when the engine is running. When activated the system maintains the current speed of the vehicle, if it is greater than 30 miles per hour, until the system is deactivated. A more detailed version of this example is presented in Chapter 6.

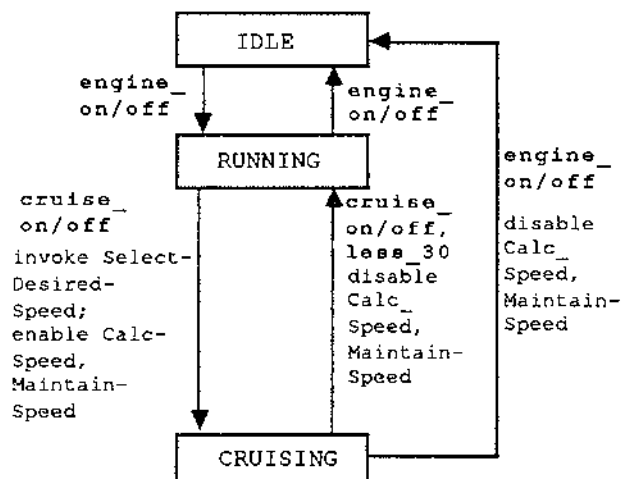
Example 2.5 continued

Example 2.5 (continued)

A control-extended DFD for a cruise-control system



The state transition diagram for the state entity in the above diagram.



2.3.3 Semantic aspects of data flows and data stores

There are two aspects to the semantics associated with data flows and data stores: *static* and *dynamic*. The static aspects are concerned with the specification of the structure and data content of data flows and data stores, while the dynamic aspects are concerned with the manner in which the data stores and data flows interact with other DFD constructs. Below an overview of the static and dynamic aspects of data flows and data stores is given.

Data flows

Data flows represent *data interfaces* between processes, or between a process and a data store or external entity. The static aspects of a data flow concerns the definition of the data type associated with the interface it represents, and the structure of the interface, where the structure of an interface is determined by the relationships between the data present in the interface. The data types and structures associated with data flows are treated as abstract data types (ADTs) to avoid premature consideration of representation issues. Instances of the types associated with data flows (and data stores) will be called objects, or simply data where it does not cause confusion.

Whether a data flow is associated with a structure or not is dependent on the type of interface it provides, which in turn is determined by the dynamic aspects of data flows. Dynamically, a data flow is associated with either a *state variable* or *data communication events*. A state variable is an entity that is *persistently present*, that is, it is always associated with a value representing its current state. An event, on the other hand, is *intermittently present*, thus one speaks about an *event occurrence*. An occurrence may be instantaneous or may be extended in time.

A data flow associated with a state variable is called a *state flow*. State flows always have external entities as their generators. For example, the data flow `checkout_time` is associated with a state variable with a value representing the current state of the external entity clock (i.e. the current time). State flows have simple dynamic interpretations: state values are simply read by their receivers whenever they are required to do so. Such flows are not associated with a structure since only one value (representing the current state) is associated with it at any time.

A data communication event is an event which is associated with data. The occurrence of a data communication event signifies the transmission of the data associated with the event. A data flow associated with data communication events is called a *data event flow*. A data event flow can either be *synchronous* or *asynchronous*. A synchronous data event flow is one which requires its generator and receivers to cooperate in order for data communication to take place. That is the

generator cannot proceed after sending data on the data flow until the receiver has acknowledged receipt of the data sent. When no such cooperation is required in order to transmit data, the data event flow is said to be asynchronous. In such cases the generators do not require acknowledgement from receivers in order to proceed after sending data on the data flow. A synchronous data event flow is associated with a single communication event, representing the synchronized generation and receipt of data. Synchronized data event flows have no structure associated with them since at most a single item of data (i.e. the data being transmitted) is associated with the flow at any time.

The uncooperative interaction associated with an asynchronous data event flow is obtained by associating with it a data structure and two communication events: *send* and *receive*. The send event of an asynchronous data flow passes on a single item of data from the data flow to all its receivers, while the receive event accepts a single item of data from its generator and 'stores' it in the the data structure awaiting transmission to its receivers. The data structures associated with asynchronous flows are queues. The send event of an asynchronous flow thus takes a data object from the top of its queue, while the receive event puts a data value at the end of the queue. The symbols used for depicting state and data event flows on a DFD are shown in Figure 2.4.



Figure 2.4 Symbols for asynchronous, synchronous and state flows

Data stores

The static aspects of data stores concern the specification of the type of data held in the data store, and the data store's structure, and are treated in the same way as the static aspects of data flows in the formal framework.

Dynamically, data stores are associated with *access* events which observe and/or modify the data store. The following are the classes of access events associated with data stores:

- *Read accesses*, for example the access events associated with the data flows `copy#_list`, `out_book`, `return_detail` of the data store `BOOK`. A read access event returns either a data object (or a sub structure of a data object), or a structure of (sub structures of) data objects in a data store. Some access schemes may require that the access event be supplied with data which identify the particular object in the data store to be accessed. The object returned by a read

access is of the type associated with its data flow. For example, the read access associated with `copy#_list` returns a list of the `copy#` attributes of the data objects in `BOOK`.

- *Updates*, for example the access events associated with the data flows `ret_updated_book`, and `out_updated_book` of the data store `BOOK`. An update changes the values of a subset of the attributes of a select set of data objects in a data store. The objects to be updated are identified by data supplied to the event.
- *Additions*, for example the access event associated with the data flow `new_book_rec` of the data store `BOOK`. Additions simply add new objects to a data store.
- *Deletions*, for example the access event associated with the data flow `deleted_book` of the data store `BOOK`. Deletions remove objects from a data store. Some access schemes may require information on the the objects to be deleted to be supplied to the event.

In the formal treatment, data stores are treated as ADTs on which concurrent accesses can be carried out. Such a treatment of data stores provides flexibility in the type of interactions possible between processes and data stores.

Example 2.6 gives type definitions for the data objects in the library application. Such definitions can be viewed as an informal front to the formal specifications characterizing the data objects. *Base types* are predefined classes of indivisible objects, or list or set structures of such objects, while *non-base types* are classes of composite objects based on the base types. The type definitions are expressed in the form `typename ::= typedefinition`, where `typename` is a name, and `typedefinition` is either another name or a *structure* of names enclosed within `<`, `>`. Structures consist of mandatory types identified by names separated by commas, and/or alternative sub structures separated by `!`. Particular instances of a type may also be included in a structure in place of type names, for example, the message/flag types of the library application are defined in terms of their instances which are text strings of the form "message", reflecting the condition which the message/flag reports on. In the definitions of the non-base types, base components are written in bold. The indivisible base types used for the library application are:

number - the class of floating point numbers,

time - the class of time points,

character - the class of characters, and

message/flag types within the application.

Aliases for base types, reflecting their use in the composite objects, are used to aid readability.

Example 2.6

Type definitions for the library application

List structures are enclosed within [,], for example, [number] is a list of objects of type number.

Non-base data types

```

bb_status      ::= <time_returned | "Not returned">
book           ::= <book_id, title, subject, author,
copy_type, borrower_indicator>
book_id        ::= <ISBN, copy#>
borr_detail    ::= <[borrower_book_detail], number>
borr_fine_record ::= <<number, borrower_id> | "Not in file">
borr_flag      ::= <"Not in file" | <out_borr,
borrower_id>>
borr_update_info ::= <borrower_id, number>
borrower       ::= <borrower_id, borrower_name,
borrower_addr, borrower_type,
[borrower_book_detail],
payment_to_date>
borrower_book_detail ::= <book_id, due_time, bb_status>
borrower_id    ::= <[character]>
borrower_indicator ::= <"Available" | borrower_id>
checkout_info  ::= <book_id, borrower_id>
checkout_message ::= <vetted_borr, vetted_book>
del_borr       ::= borrower_id
delete_book    ::= book_id
deleted_borr   ::= [borrower_book_detail]
deleted_book   ::= borrower_indicator
ISBN           ::= <[integer]>
new_book       ::= <ISBN, title, subject, author,
copy_type>
new_book_rec   ::= book
new_borr       ::= <borrower_id, borrower_name,
borrower_addr>
new_borr_rec   ::= borrower
other_borr     ::= borrower_id
out_book       ::= <borrower_indicator, copy_type>
out_book_id    ::= book_id
out_borr       ::= <[borrower_book_detail],
borrower_type, payment_to_date>
out_borr_id    ::= borrower_id
out_updated_book ::= borrower_indicator
out_updated_borr ::= [borrower_book_detail]
ret_borr_list  ::= [borrower_book_detail]
ret_updated_book ::= borrower_indicator
ret_updated_borr ::= [borrower_book_detail]
return_detail  ::= borrower_indicator
return_info    ::= book_id
update_id      ::= borrower_id
update_status  ::= <outstanding_fine | excess_number |
"Not in file" | "No fines" | "Cleared">
vetted_book    ::= <<book_id, copy_type> | "book not in
file" | "book already checked out"
| "not borrowable">

```

Example 2.6 continued

Example 2.6 (continued)

Type definitions for the library application

```

vetted_borr          ::= <<"Fines over limit", number> |
                        "borrower not in file" |
                        <out_borr, borrower_id>
vetted_return_book  ::= <<"Not in file" | "Already returned" |
                        <book_id, borrower_id> >

Base data types
add_message         ::= <"OK" | "Borrower already in file">
amount_paid        ::= number
author             ::= [character]
borrower_addr      ::= [character]
borrower_name      ::= [character]
borrower_type      ::= <"undergrad" | postgrad" | "staff">
checkout_time      ::= time
copy#              ::= integer
copy#_list         ::= [integer]
copy_type          ::= <"book" | "reference" | "periodical">
del_borr_mess      ::= <"OK" | "Not in file" | "Has books out">
delete_message     ::= <"delete-OK" | "Not in file" |
                        "Not available">
due_time           ::= time
excess_number      ::= number
fine               ::= number
fines_record       ::= [number]
new_copy#          ::= integer
outstanding_fine   ::= number
paidup_amount      ::= number
payment_to_date    ::= number
return_message     ::= <"Already in" | "Not in file" |
                        "Ok return">
return_time        ::= time
subject            ::= [character]
time_returned      ::= time
title              ::= [character]
update_time        ::= time
updated_borr_detail ::= number

```

2.3.4 Semantic aspects of processes

Unlike the usual logical approaches to interpreting process behaviour in DFDs (eg. see [War86, Woo88, Hat88]), the transformation from inputs to outputs is not assumed, within the formal framework, to be instantaneous. Such a logical view may be helpful as a first approximation of behaviour, but is of little use to further development since no operational view can be consistent with it [KK88]. The behaviour of a process is characterized by its class of *invocations* (or *p-invocations*), where an invocation represents a particular transformation of single instances of the types associated with some of the inputs of the process, to single instances of the types associated with some of its outputs. Formally, an invocation is a labeled sequence of *states*, where the labels represent the effects of events

occurring within the transformation represented by the invocation. Such a labeled sequence can be depicted as follows: $s_0-l_0-s_1-l_1-s_2-\dots-s_n-l_n-s_{n+1}$, where s_i ($1 \leq i \leq n$) is a state, and l_i ($1 \leq i \leq n$) is its associated label. The states of an invocation reflect the observable effects of events thus far in an invocation. Such states are said to be *observable*. An event represented by a label in an invocation causes a change from its associated state in the sequence to the next state in the sequence. The first state of a process invocation is called its *idle* state, and represents the situation where no inputs of the process are being transformed into outputs. The event which causes a change from an idle state to another state is called an *invocation event* of the process, and a process is said to be *invoked* when it occurs.

Operational models of behaviour can be associated with DFD processes, since transformations are interpreted as sequences of observable states rather than instantaneous conversions of inputs to outputs. The class of operational models associated with a DFD process is abstractly characterized by an algebraic specification of the labeled state transitions that can take place in its invocations. The class of invocations characterizing a process's behaviour can be pictorially represented by a *state transition tree* (STT), with *classes* of states as nodes and *classes* of labels as edges. Example 2.7 shows the STT for the process `CheckBook`. Conditions under which certain transitions can take place can be included in STTs by associating such conditions, expressed in an appropriate language, with the respective edges. STTs can thus be made to show all information necessary for characterizing the invocation class of a process.

Example 2.7

State transition tree for the process `CheckBook`

A state of `CheckBook` is of the form $\langle \text{Book_id}, \text{Status}, \text{Vett_Book} \rangle$, where `Book_id` is either an object communicated via the data flow `out_book_id`, or a null object, `Null1`, representing the situation where no such communication has occurred, `Status` is either an object communicated via the data flow `out_status`, or a null object, `Null2`, representing the situation where no such communication has occurred, and `Vett_Book` is either an object to be sent for communication via the data flow `vetted_book`, or a null value, `Null3`, representing the situation where no object is available for communication on `vetted book`. $\langle \text{Null1}, \text{Null2}, \text{Null3} \rangle$ is thus the idle state of the process.

Example 2.7 continued

Example 2.7 (continued)

State transition tree for the process CheckBook

The classes of event labels associated with CheckBook are the following:

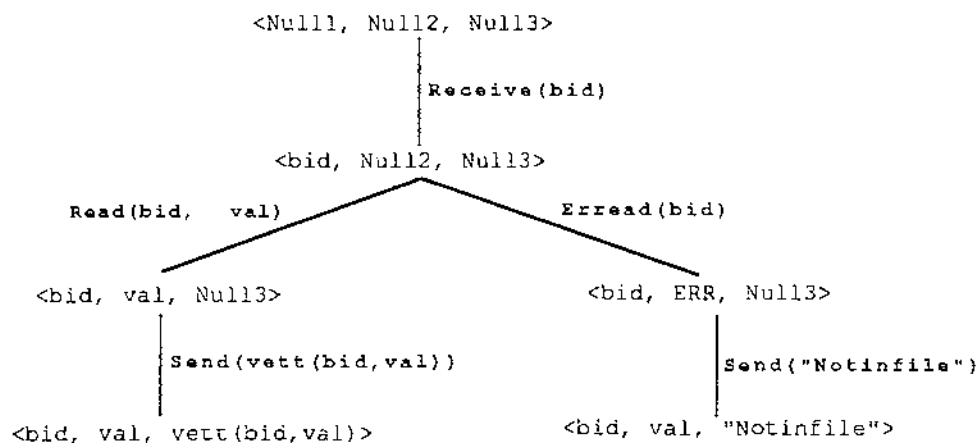
Receive (bid) - representing a receive communication event which receives a book_id object, bid, from the data flow out_book_id;

Read (bid, val) - representing a read event to the data store BOOK, which accesses a book object identified by a book_id object bid, and retrieves the out_book object, val, associated with the book object.

Erread (bid) - representing an unsuccessful read access to BOOK. This may occur, for example, when the book object identified by bid is not in BOOK.

Send (vbook) - representing a send communication event, which sends a vetted_book object, vbook, on the data flow vetted_book. The function vett returns a vetted_book object given a book_id object and an out_book object. This function checks whether the book can be borrowed.

The STT for CheckBook is shown below:



The above STT can be intuitively interpreted as follows: A particular transformation of CheckBook would first receive data from the data flow out_book_id, represented by the occurrence of the receive event (the invocation event) whose effect is labeled by Receive (bid), and then attempt a read access to the data store BOOK. The effect of a successful read attempt is represented by a label of the form Read (bid, val), while an unsuccessful read attempt is represented by the label Erread (bid). The next externally observable event is the send event which sends data on the data flow vetted_book, the value of which is dependent on whether a successful or unsuccessful read attempt was made.

Example 2.7 continued

Example 2.7 (continued)

State transition tree for the process CheckBook

```
An example of a CheckBook invocation is the labeled sequence: <Null1,
Null2, Null3>-Receive (bid) -<bid, Null2, Null3>-
Erread (bid) -<bid, ERR, Null3>-Send ("Notinfile") -<bid,
val, "Notinfile">.
```

The quasi-formal specifications associated with processes in the SA approach are replaced by formal specifications of behaviour created using the techniques of the formal framework. The formal specifications are algebraic characterizations of all the possible state transitions that can occur as a result of the occurrences of events.

2.3.5 Describing the interactions in a DFD

A DFD is interpreted as a system of processes and data stores interacting with an external environment. The environment interacts with the system in an uncooperative manner, thus allowing the system and the environment to proceed at different speeds, without the need to synchronize for communication. Such interaction often occurs in real-time applications and is sometimes a desirable feature of some non real-time applications [KK88]. Uncooperative interaction between the environment and the application is represented by asynchronous data flows or state flows in extended DFDs, thus data flows between external entities and processes are either state flows or asynchronous data flows.

In describing the interactions in the system of processes and data stores of a DFD the processes of the DFD are partitioned into *actions*, allowing a modular description of interactions. An action is a system of related processes in which certain processes are designated as *invoker*, and in which each process which is not an invoker:

- depends only on the other processes in the action for its data inputs; and
- is not associated with control inputs.

The invokers of an action are the processes that must be invoked before any of the other processes in the action can be invoked. The invocation events of invokers are synchronized with each other, thus an action can be thought of as being *invoked* by a single synchronization event. Only the invokers of a process can be associated with input control flows. Since the invocation events of the invokers of an action are synchronized, an initiator associated with a particular

invoker of an action must also be associated with all the invokers of the action. An action whose invokers have disablers as inputs cannot be invoked when at least one of its invokers is disabled, in which case the action itself is said to be *disabled*. Similarly, an action whose invokers have enablers as inputs can only be invoked when *all* of its invokers are enabled, in which case the action itself is said to be *enabled*.

The *terminators* of an action are the processes in the action which have outputs to external entities, data stores, and/or invokers of other actions. Once invoked an action transforms the data inputs of its invokers to data outputs on its terminators. An action can thus be viewed as a high level process, where the inputs of its invokers are referred to as the inputs of the action, and the outputs of the terminators directed towards data stores, external entities and other actions are referred to as the outputs of the action, and each disabler to its invokers is referred to as an action disabler, while an enabler representing the conjunction of all the enablers associated with its invokers is called the enabler of the action.

In Figure 2.5, a DFD is partitioned into the actions A1, A2 and A3. A1 and A2 together cannot form an action since the process p2 has an input from p4, which is not in A1 or A2.

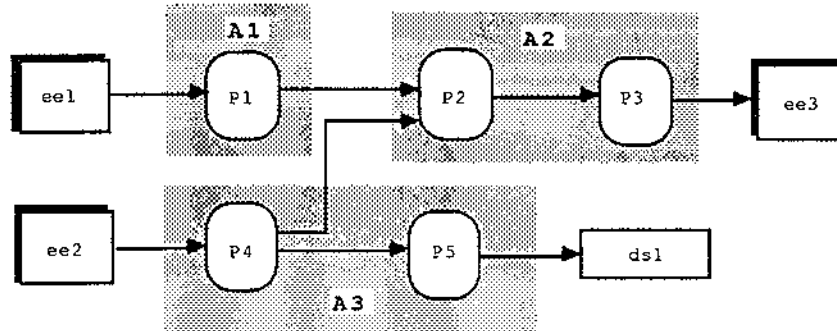


Figure 2.5 Actions in a DFD

Actions are also associated with a *termination* event which causes all its constituent processes to revert to the idle state. Such an event occurs when an action has transformed its inputs to outputs. An action is said to be *terminated*, or in an idle state, when all its processes are in the idle state. Notice that the behavioural semantics associated with processes implies that an action, once invoked, cannot be invoked again until it has terminated.

The behaviour of an invoked action is determined by the behaviour of its processes. Within an action, all data flows which are not also inputs to or outputs from the action, are *synchronous*. Intuitively, actions are systems of processes

which cooperate, via synchronization, to transform single instances of its inputs to outputs.

Splitters and binders occur only in actions, and force their associated processes to synchronize. The processes *associated with the outputs of a splitter* are the processes to which the outgoing data flows are directed to, while the processes *associated with the inputs of a binder* are the processes from which the incoming data flows emanate. The processes associated with the outputs of a splitter, and the processes associated with the inputs of a binder, are forced to synchronize the receipt and generation of data on the respective data flows. Figure 2.6 illustrates the different situations in which binders and splitters may occur. In Figure 2.6(a) a binder takes p inputs, all of which must be synchronous data flows, from processes, and generates a synchronous data flow, called its *output*, directed towards other processes and/or splitters. This situation is interpreted as a synchronization of the send events of the processes associated with the inputs of the binder and the receive events of the processes associated with the output of the binder, either directly or indirectly via splitters. In Figure 2.6(b) the output of the binder is an asynchronous data flow. This situation is interpreted as a synchronization of the send events of the processes associated with the inputs of the binder, and the receive event of the asynchronous data flow. Figure 2.6(c) shows a splitter with p outputs directed towards processes, and an incoming synchronous data flow, called its *input*, emanating from a process or a binder. This situation is interpreted as a synchronization of the send event of the process from which the data flow emanates, or in the case that the input emanates from a binder, the send events of the processes associated with the inputs of the binder, and the receive events of the processes associated with the outputs of the splitter. In Figure 2.6(d), the input to the splitter is an asynchronous data flow. This situation is interpreted as a synchronization of the send event of the asynchronous data flow and the receive event of the processes associated with the outputs of the splitter.

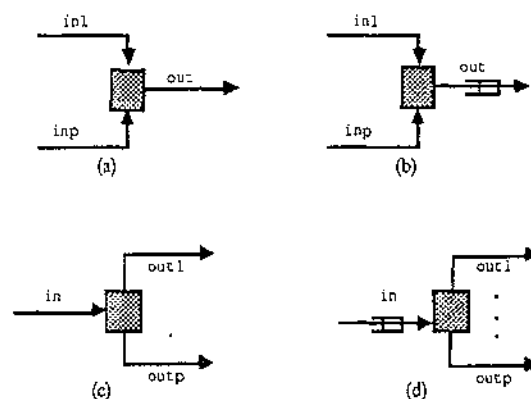


Figure 2.6 Communication situations involving binders and splitters

The specification of behaviour of an action is derived from the specifications of its processes. An action is viewed as a system of synchronously interacting processes, thus, provided the specification of the processes are given, together with specifications of the data transmitted by the processes, the action's specification can be generated.

A class of labeled sequences of states can be associated with actions, in the same way they can be associated with processes. Such sequences, called *a-invocations*, represent the sequence of states an action passes through when transforming a particular data on a subset of its inputs to data on some of its outputs. The states of an action is a tuple of states of its DFD processes, while the events represented by the labels of an a-invocation are occurrences of action events arising from the interactions of its DFD processes.

All communication between actions are uncooperative (represented by asynchronous data flows), while communication between actions and data stores are always cooperative (represented by synchronous data flows). The output flows of a data store may be associated with splitters, representing the situation where the decomposed parts of the data flows are needed in different parts of a action. In such a situation, the read events of the processes associated with the outputs of the splitter are synchronized. Similarly an input to a data store may be the output of a binder, in which case the write events of the processes associated with the inputs of the binder must be synchronized. These situations are illustrated in Figure 2.7.

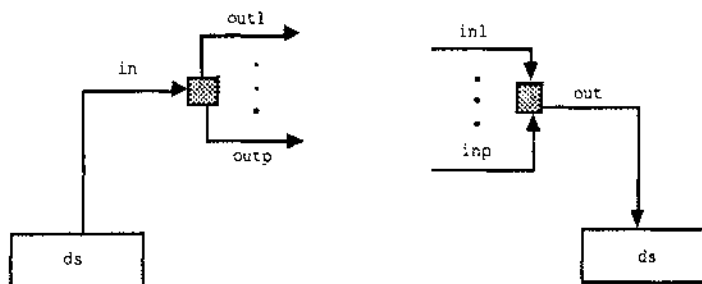


Figure 2.7 Relationships between the extractors and data stores, and between binders and data stores

Actions interact with asynchronous data flows in a synchronized manner, where the send event of an action which is a generator of the asynchronous data flow is synchronized with the receive event of the asynchronous data flow, while the receive event of an action which is a receiver of an asynchronous data flow is synchronized with the send event of the asynchronous data flow.

The DFD resulting from the identification of actions, synchronous and asynchronous data flows, and state flows, in a possibly control-extended DFD is called an *Extended DFD* (ExtDFD). An ExtDFD is interpreted as a system of interacting actions. The view of actions as high-level processes permits the technique used for specifying the behaviour of processes to be used to specify the behaviour of actions. The specification of an ExtDFD's behaviour is derived from the specifications of its actions and the specifications of the dynamic and static aspects of its data stores and asynchronous data flows, and from a specification of the effects of events on the mode of operation (depicted by control flows directed towards the state entity).

The BS characterizes the behaviour of ExtDFDs in the same way as processes and actions are characterized. The state of an ExtDFD consists of the states of its actions and data stores, as well as a flag indicating the current mode of operation the ExtDFD is in (this can be omitted when the ExtDFD has only one mode). The set of events associated with an ExtDFD consists of action events, and the events arising from the interaction amongst actions, data stores and external entities, and events associated with control flows.

2.4 Summary

This chapter presented, in an informal setting, the syntactic and semantic aspects of DFDs on which the formal framework developed in this thesis is founded. The syntactic aspects of DFDs are concerned with the building of correct syntactic structures, and hierarchies of such structures, and are encapsulated by abstract objects. The part of the formal framework concerned with formalizing the syntactic aspects of DFDs is called the *Picture Level* (PL), and is described in Chapter 4.

The semantic aspects concern the building of a specification of behaviour for suitably extended DFDs called ExtDFDs. Dynamically, an ExtDFD is a system of interacting processes with an uncooperative interface to its environment. The formal specification of behaviour for an ExtDFD characterizes what the ExtDFD is allowed to do in terms of the possible interactions amongst its components. The building of such a specification requires that the syntactic structures are associated with dynamic, as well as static interpretations. The part of the formal framework which provides support for specifying the semantic aspects of DFDs is called the *Specification Level* (SL), and is described in Chapter 5.

CHAPTER 3

Positive-Negative Relational Specifications: An Algebraic Approach to Specification

3.0 Introduction

Specification techniques based on data abstraction have been developed by many researchers (see, for example [GTW78, GHM78, LZ75, LZ77]), and the use of such techniques for specifying applications has shown promising results. The data abstraction approach to specifying applications entails viewing applications as consisting of groups of related functions, acting upon particular classes of objects, with the constraint that the behaviour of the objects can only be observed through application of the functions [LZ75].

Algebraic specification techniques are a class of techniques based on the data abstraction approach, which have firm mathematical foundations based on concepts from universal algebra and mathematical logic [GTW78, WB82]. Such techniques provide implicit definitions of classes of objects and their functions in terms of algebraic relations. The resulting (algebraic) specifications, are syntactic entities, consisting of a declaration part, called the *signature*, and a set of relations, called *laws*, between terms formed using the symbols declared in the signature. An algebraic specification is associated with a model semantics in the form of a class of *algebras*. The mathematical foundations for algebraic specification techniques enables the generated specifications to be used in the investigation of formal properties of the objects they characterize. Also, some researchers provide formal criteria for establishing whether an algebraic specification implements another.

Considerable research has also gone into providing an operational semantics for algebraic specifications based on *term rewriting systems* [Hue80, Kap84, Kap87, Jou87]. Under suitable conditions, such operational semantics provide effective deduction systems, which can be used to investigate properties of the specifications in a computational manner. The conditions under which decidable deduction systems can be obtained can place serious restrictions on the form of laws which, inevitably, affects their expressiveness. Research in this area has progressed from algebraic specifications consisting only of unconditional equational laws (see, for example [Hue80, Der87, HO80]) to specifications consisting of conditional equational laws (see for example [Kap84, Jou87, Dro84, RZ84, BK82, CTRS87]). Conditional laws are more expressive than their unconditional

counterparts, but are still not expressive enough to specify some objects 'naturally'. Recent research, in this area, which introduce inequalities into the condition parts of conditional laws, look promising in this respect [Kap87, MS87].

Current algebraic specification techniques are well supported by firm mathematical and operational foundations, but no single such technique provides the expressive power needed to support for range of specifications required by the formal framework developed in this thesis. In this chapter a specialized algebraic technique, together with its mathematical foundations, is introduced. The technique unifies and extends techniques based on partial functions [WB82], relations [ARW86], and conditional term rewriting with inequalities [MS87]. In what follows, concepts and notations from the works of Goguen et al [GTW78], and Wirsing and Broy [WB82] are freely used.

3.1 Positive-Negative Relational Specifications (RSs)

A *positive-negative relational specification*, or simply called a relational specification (RS), is a partial conditional algebraic specification with relations. In this section, the concepts and notation used for building RSs are discussed. In particular, it is shown how the notions of hierarchy and schema help reduce the complexity in building and understanding large RSs.

3.1.1 Specifications and algebras

A RS consists of a *signature* and a set of *laws*. The signature is the declaration part of the RS while the laws are relations between terms formed by the symbols declared in the signature. The formal definition of a RS signature given in Definition 3.1, utilizes the following notion of an indexed set: a *S-indexed set*, A , is a family of component sets A_s for each index s in S .

Function symbols can be partitioned into two sets:

- The set of function symbols called *constructors*, C , representing functions which create new objects of a sort.
- The set of all other function symbols, called *non-constructors*.

The signature of a RS is associated with a class of algebraic models, called Σ -RS algebras, or simply Σ -algebras. A Σ -algebra, defined in Definition 3.2, provides representations for the objects of each sort, and interpretations for the function and relation symbols in the signature.

Definition 3.1

RS Signature

A RS signature $\Sigma = \langle S, F, R \rangle$ consists of:

- a non-empty set S of sorts (S^* denotes the set of all finite strings from S , including the empty string denoted by λ);
- a non-empty $S^* \times S$ indexed set, $F = \{F_{w,s} \mid w \in S^*, s \in S\}$, where $F_{w,s}$ is the set of function symbols with *arity* w , a string made up of the domain sorts of the functions, and *sort* s , the sort of the object returned by the functions (a function symbol f in the set $F_{w,s}$ where $w = s_1 \dots s_n$, will be written as $f: s_1, \dots, s_n \rightarrow s$); and
- a non-empty S^* -indexed set $R = \{R_w \mid w \in S^*\}$, where R_w is a set of relation symbols of *sort* w (a relation symbol r in R_w , where $w = s_1 \dots s_n$, will be written as $r: s_1, \dots, s_n$).

Definition 3.2 Σ -RS Algebra

For a RS signature $\Sigma = \langle S, F, R \rangle$, an *algebra with relations*, $A = \langle \{A_s \mid s \in S\}, A_F, A_R \rangle$, is called a Σ -RS algebra if it consists of:

- a S -indexed set, $\{A_s \mid s \in S\}$, called the *carrier sets*;
- a $S^* \times S$ -indexed family of functions $A_F = \{A_{w,s} : F_{w,s} \rightarrow [A_w^1 \rightarrow A_s] \mid w \in S^*, s \in S\}$, which consists of functions mapping function symbols in F to *partial* functions, where $[A_w \rightarrow A_s]$ denotes the set of all partial functions from A_w to A_s ; and
- an S^* -indexed family of functions $A_R = \{A_{Rw} : R_w \rightarrow \text{Atup}_w \mid w \in S^*\}$, which consists of functions mapping the relation symbols in R to elements in Atup_w , where Atup_w is the set of all sets whose elements are A_w tuples.

The interpretation of a function or relation symbol, t , in an algebra, A , is denoted by t_A . Relation symbols are interpreted as sets of tuples, where each tuple of the set signifies that the relation represented by the symbol holds amongst the objects in the tuple.

Example 3.1 gives an example of a signature and an algebra for the signature.

¹ A_w , where $w=(s_1 \dots s_n) \in S^*$, represents the cartesian product $A_{s_1} \times \dots \times A_{s_n}$.

Example 3.1

A signature for natural numbers

```

 $\Sigma(\text{Natnum}) \equiv$ 
  Signature
    sorts nat
    constructors
      0:  $\rightarrow$  nat
      succ: nat  $\rightarrow$  nat
    auxiliary functions
      +: nat, nat  $\rightarrow$  nat
    relations
      <: nat, nat

```

An algebra, A , for $\Sigma(\text{Natnum})$ can be defined where A_{nat} is the set of natural numbers, $+$ is mapped to the addition function on natural numbers, and $<$ is mapped to the relation "is less than" defined on natural numbers.

Every signature, Σ , defines a set of syntactically correct expressions, called *well-formed terms*, built using the function and relation symbols of Σ . Such terms can be partitioned into two sets: a set of function terms, called *F-terms*, and a set of relation terms, called *R-terms*.

Definition 3.3

Well-formed F-terms

For a signature $\Sigma = \langle S, F, R \rangle$, and an S -sorted set $\{X_s \mid s \in S\}$ of symbols called *variables*, the set of *well-formed function terms*, called *F-terms*, of sort s in S with variables is defined as the least set, $T(F, X)_s$, having the following properties :

- all variables x in X_s are F-terms of sort s ,
- all constant symbols, $f : \rightarrow s$ (i.e. function symbols with arity λ , and sort s), in F are F-terms of sort s ,
- for all function symbols $f : s_1, \dots, s_n \rightarrow s$ in F ($n > 0$) and all F-terms t_1, \dots, t_n of sorts s_1, \dots, s_n respectively, $f(t_1, \dots, t_n)$ is a F-term of sort s .

A *ground F-term* is a F-term containing no variables (i.e. elements in X). The set of ground F-terms of sort s is denoted by $T(F)_s$. A *constructor term* is a term which consists only of constructor symbols and variables. Thus a constructor term is of the form $c(c_1, \dots, c_n)$ where c is a constructor and c_1, \dots, c_n are constructor terms. The set of all ground constructor terms of sort s is denoted by $Tc(F)_s$.

Definition 3.4

Well-formed R-terms

Given a signature $\Sigma = \langle S, F, R \rangle$, the set of *well-formed relation terms*, called *R-terms*, of type $w \in S^*$, with free variables from an S -sorted set $\{X_s \mid s \in S\}$ of symbols called *variables*, is defined as the least set, $T(R, X)_w$, having the following property:

- for all relation symbols $r : s_1, \dots, s_n$ in R ($n > 0$), and all F-terms t_1, \dots, t_n of sorts s_1, \dots, s_n respectively, $r(t_1, \dots, t_n)$ is a R-term of type $w = s_1 \dots s_n$.

A *ground* relational term is a R-term containing no variables. The set of ground R-terms of type $w \in S^*$, is denoted by $T(R)_w$. The union of the set of F- and R-terms will be denoted by $T(\Sigma, X)$, and the terms are collectively called Σ -terms. The set of ground Σ -terms is denoted by $T(\Sigma)$.

Example 3.2

Examples of F- and R-terms

Examples of F-terms from the signature in Example 3.1 are $\text{succ}(\text{succ}(0))$, $\text{succ}(x) + \text{succ}(\text{succ}(\text{succ}(x)))$, where $\text{succ}(\text{succ}(0))$ is a ground constructor term. Examples of R-terms from the same signature are $\text{succ}(x) + \text{succ}(\text{succ}(x)) < \text{succ}(0)$ and $\text{succ}(0) < 0$, where $\text{succ}(0) < 0$ is a ground relation term.

The 'evaluation' of a Σ -term in a Σ -algebra is intuitively captured by the notion of an interpretation. For a Σ -algebra, $A = \langle \{A_s \mid s \in S\}, A_F, A_R \rangle$, a S -indexed set of variables $\{X_s \mid s \in S\}$, and a S -indexed family of partial functions $V = \{v_s \mid v_s : X_s \rightarrow A_s\}$, an *interpretation with respect to V* , of a Σ -term t in A , denoted by $V_A(t)$, is defined as follows :

- (1) $V_A(x_s) = v_s(x_s)$ for $x_s \in X_s$.
- (2) $V_A(f(t_1, \dots, t_n)) = f_A(V_A(t_1), \dots, V_A(t_n))$ for $f \in F$, provided that every $V_A(t_i)$, $1 \leq i \leq n$, is defined and the n -tuple $(V_A(t_1) \dots V_A(t_n))$ is in the domain of f_A . Otherwise $V_A(f(t_1, \dots, t_n))$ is undefined.
- (3) $V_A(r(t_1, \dots, t_n)) = (V_A(t_1) \dots V_A(t_n))$ for $r \in \text{Rel}$ (the set of relation symbols in Σ), provided that every $V_A(t_i)$, $1 \leq i \leq n$, is defined, and $(V_A(t_1) \dots V_A(t_n)) \in r_A$. Otherwise $V_A(r(t_1, \dots, t_n))$ is undefined.

The interpretation of a ground term t' in an algebra A does not depend on V , and its unique interpretation is denoted by t'_A .

Every Σ -algebra, A , contains a least sub algebra A' which is finitely generated by the constants in Σ . If A does not contain a *proper* sub algebra (i.e. $A = A'$) then A is called a *finitely generated algebra* [WB82]. Ground terms define a

special finitely generated Σ -algebra called the Σ -term algebra, denoted by T_Σ , with the carrier sets $T(F)_s$ for $s \in S$, functions $f: T(F)_{s_1}, \dots, T(F)_{s_n} \rightarrow T(F)_s$ mapping (t_1, \dots, t_n) to the term $f(t_1, \dots, t_n)$, for $f \in F$, and a set $\{r(t_1, \dots, t_n) \mid r(t_1, \dots, t_n) \in T(R)_w\}$ for each relation symbol, $r \in R$ with type $w = s_1, \dots, s_n$.

An interpretation, V_A , from $T(\Sigma, X)$ into a Σ -algebra A induces a congruence on the F-terms, $=_A$, called the *strong equality* of A , defined as follows:

$t =_A t'$ if and only if $t_A = t'_A$

that is the F-terms t and t' are congruent with respect to the algebra A if and only if either both F-terms are undefined in A or both F-terms are defined in A and their interpretations are equal.

The definedness of Σ -terms in an algebra A is determined by associating a predicate, called an *ok-predicate*, with each sort in Σ , defined over the terms of the sort as follows:

$D_A(t) = \text{true}$ if $V_A(t)$ is defined, and

$D_A(t) = \text{false}$ if $V_A(t)$ is not defined, where t is a term of sort s , and D is the ok-predicate associated with the sort.

Properties of the objects declared in a Σ -signature are implicitly expressed by statements, called *laws*, in a first-order language of Σ -terms. Such laws characterize the behaviour of the functions and relations on the objects by establishing relationships between them. Well-formed Σ -laws are defined in Definition 3.5.

Definition 3.5

Well-formed positive-negative conditional Σ -laws

A well-formed positive-negative conditional Σ -law has the following form :

$$\bullet (\bigwedge_{i=1..j} \text{ok}_{c_i}(t_i)) \wedge (\bigwedge_{i=1..l} (\text{ok}_{a_i}(u_i) \wedge \text{ok}_{a_i}(v_i) \wedge u_i = v_i)) \wedge (\bigwedge_{i=1..n} (\text{ok}_{b_i}(u'_i) \wedge \text{ok}_{b_i}(v'_i) \wedge u'_i \neq v'_i)) \wedge (\bigwedge_{i=1..o} r_{d_i}(w)) \wedge (\bigwedge_{i=1..p} \sim r'_{e_i}(w')) \Rightarrow C,$$

where $t_i, u_i, v_i, u'_i, v'_i$ are F-terms in $T(\Sigma, X)$, $\text{ok}_{c_i}, \text{ok}_{a_i}$ and ok_{b_i} are ok-predicates, a_i, b_i, c_i, d_i , and e_i are sorts, and $r_{d_i}(w)$ and $r'_{e_i}(w')$ are R-terms in $T(\Sigma, X)$. C is either of the form $\text{ok}(t)$, z , or $x = y$, where z is a R-term and t, x , and y are F-terms in $T(\Sigma, X)$. C is called the *consequence*, while the expression to the left of the implication symbol, \Rightarrow , is called the *antecedent* of the law. A literal of the form $\sim r$, where r is a R-term, is called a *negated relation* (n-relation).

A *closed law* is a formula having no free variables, while a *ground law* is one which has no variables.

Well-formed Σ -laws are assumed to be universally quantified on defined terms only.

A Σ -algebra, A , satisfies a Σ -law, ζ of the form given in Definition 3.5, denoted by $A \models \zeta$, if and only if for all interpretations V_A :

- $ok_{ciA}(t_i)$, $i = 1$ to j , and
- $ok_{aiA}(u_i) \wedge ok_{aiA}(v_i) \wedge u_i =_A v_i$, $i = 1$ to l , and
- $ok_{biA}(u'_i) \wedge ok_{biA}(v'_i) \wedge u'_i \neq_A v'_i$, $i = 1$ to n (i.e. u'_i and v'_i do not have equal interpretations), and
- $r_{di}(w) \in r_{diA}$, $i = 1$ to o , and
- $r'_{ei}(w') \notin r'_{eiA}$, $i = 1$ to p ,

implies that $ok_A(t)$, or $x =_A y$, or, for $z = r(t)$, $z \in r_A$, depending on the form of the consequence.

A formal definition of the structure of a RS can now be given.

Definition 3.6

Positive-negative relational specification (RS)

A positive-negative relational specification (RS) $PR = \langle \Sigma + OK, E \rangle$ consists of a signature, $\Sigma + OK$, where OK is a set of ok -predicate symbols for each sort in Σ , and a set E of well-formed Σ -laws.

The class of algebras satisfying the laws of a RS is denoted by $Alg_{\Sigma, E}$. In presenting the laws of an RS a comma is used in place of the symbol \wedge , and the following short form is used:

- A law $u \Rightarrow v$, where t_i are the free variables occurring in $u \Rightarrow v$, is the short form for $(\bigwedge_{i=1..j} ok_{ci}(t_i)) \wedge u \Rightarrow v$, for example a law $f(x1, x2) = g(x3)$ is the short form for $ok1(x1) \wedge ok2(x2) \wedge ok3(x3) \Rightarrow f(x1, x2) = g(x3)$, where oki is the ok -predicate associated with the sort of x_i , $1 \leq i \leq 3$.

RS laws are derived and presented in a modular fashion, with each non-constructor and relation symbol, of the RS being associated with a unique set of laws, called its *characterizing set*, which characterizes the function or relation. Characterizing sets are presented so that they are distinguishable: the characterizing set for a function symbol, f , consists of all laws in which f appears in the consequence as the outermost symbol on the left hand side of the equality, while the characterizing set of a relation symbol, r , consists of all the laws in which r appears as the outermost symbol of the consequence. Example 3.3 has examples of characterizing sets.

The defined objects of a sort are characterized in an RS by the set of laws whose consequences have ok -predicate symbols of the sort as the outermost symbols (the characterizing set of the ok -predicate).

Example 3.3

An RS characterizing natural numbers

```

Natnum ≡
  Signature
  sorts nat
  constructors
    0: → nat
    succ: nat → nat
  auxiliary functions
    _+_ : nat, nat → nat
  ok-predicates
    oknat: nat
  relations
    _<_ : nat, nat
  Laws  $\forall x, x1, x2: \text{nat}$ 
  Characterizing set for oknat
  1. oknat(0)
  2. oknat(succ(x))
  Characterizing set for +
  3. x+0 = x
  4. x1+succ(x2) = succ(x1+x2)
  Characterizing set for <
  5. 0<succ(x) = true
  6. x1<x2 = true  $\Rightarrow$  succ(x1)<succ(x2) = true

```

The modular approach for presenting laws is not enough to control the complexity in large RSs. Two syntactic concepts which have proved useful in this respect are hierarchy [WB82] and schemas. The aim is to control complexity by permitting complex specifications to be built up from simpler and/or generic specifications.

3.1.2 Hierarchical RSs

A hierarchical RS provides a leveled view of a specification, where each lower level contains RSs that are simpler than those at the higher levels. Thus an understanding of the RS is based on an understanding of its simpler components. Hierarchical RSs are defined in Definition 3.7.

Note that every primitive term is of primitive sort but a term of primitive sort is not necessarily primitive. A hierarchical RS, HS, based on primitive hierarchical RSs, HS1, ..., HS_n, is presented in the following manner: HS ≡ HS1 + ... + HS_n + **Signature Sig Laws E**, where Sig is the signature declaring the non-primitive sorts and symbols, and E is a set of laws.

Definition 3.7

Hierarchical RSs

A *hierarchical RS* HS is a triple $\langle \Sigma, E, \{P_1, \dots, P_n\} \rangle$, where the hierarchical RS, P_i ($1 \leq i \leq n$), with signature Σ_{pi} and a set of laws E_{pi} , is contained in HS, i.e. Σ_{pi} is a subset of Σ , and E_{pi} is a subset of E . P_i is called a *primitive RS* of HS, and HS is said to be *based* on the RSs in $\{P_1, \dots, P_n\}$. A ground term t is called *primitive* if t is built solely from symbols declared in the primitive RSs (primitive symbols). A F-term t' is said to be of *primitive sort* if t' is of sort s and s is a sort declared in a primitive RS (primitive sort).

In order to preserve the algebraic interpretations of primitive RSs within the context of hierarchical RSs the notion of *hierarchy-constraints* is used. An algebra *satisfies the hierarchy-constraints* if the primitive carrier sets are built only by interpretations on the primitive ground terms [WB82]. Intuitively, this means that non-primitive constructors cannot create new objects of a primitive sort.

Definition 3.8

Reducts and hierarchy constraints

Let A be a Σ -RS algebra, and let Σ' be a sub signature of Σ (i.e. the set of sorts and symbols of Σ' is a subset of the set of sorts and symbols of Σ). The Σ' *reduct* of A , denoted by $A|\Sigma'$, is the Σ' -algebra whose carriers, functions and relations are those of A named in Σ' . The Σ' -sub algebra of A generated by the relation, and function symbols in Σ' , is denoted by $\langle A \rangle_{\Sigma'}$. An algebra A satisfies the *hierarchy constraints with respect to Σ'* if and only if $A|\Sigma' = \langle A \rangle_{\Sigma'}$, that is the Σ' -reduct of A is a finitely generated algebra.

The above can be extended to hierarchy-constraints with respect to a set of signatures by considering all hierarchy constraints with respect to the signatures in the set. For any hierarchical RS $\langle \Sigma, E, \{P_1, \dots, P_n\} \rangle$, the class of all finitely-generated Σ -algebras which satisfy the hierarchy constraints with respect to $\{\Sigma_{pi} \mid 1 \leq i \leq n\}$, and the laws of E , is denoted by $HAlg(\Sigma, E, SP)$, where $SP = \{P_1, \dots, P_n\}$. In Section 3.4 it is shown that a sufficient completeness condition, derived from an operational interpretation of RSs, ensures the existence of such algebras.

The hierarchical and modular approach to presenting RS functions determines a relationship on the function symbols, \prec_h , defined as follows:

Definition 3.9

The relation $<_h$ on function symbols

For a hierarchical RS, $\langle \Sigma, E, SP \rangle$, $f <_h g$, where $f, g \in \Sigma$, if and only if:

- f is a primitive function symbol and g is a non-primitive function symbol; or
- f is a constructor and g is a non-constructor at the same level as f ; or
- f and g are function symbols at the same level and f appears as the outermost symbol of a sub term in the characterizing set of g , and g does not appear as the outermost symbol of any sub term in the characterizing set of f .

The above relation is used as part of a syntactic check on the sufficient completeness property mentioned above, given later in this chapter.

Example 3.4

Characterizing the natural numbers by a hierarchical RS

```

Setnum = Boolean + Natnum +
Signature
  sorts setnum
  constructors
    ∅ : → setnum
    --- constant symbol for an empty set ---

    insert : nat, setnum → setnum
    --- symbol for the function which adds a natural number to a
    set -
  auxiliary functions
    isempty : setnum → boolean
    --- symbol for the function which returns the value true if
    and only if the set is empty ---

    isin : nat, setnum → boolean
    --- symbol for the function which returns the value true if
    and only if the natural number is in the set ---

```

Example 3.4 continued

Example 3.4 (continued)

Characterizing the natural numbers by a hierarchical RS

```

issubset : setnum, setnum → boolean
--- symbol for the function which returns the value true if
and only if the leftmost set of the argument is a subset of
the rightmost set ---

-int-, +_ : setnum, setnum → set
--- int is the symbol for the function which returns set
which is an intersection of the two sets, and + is the symbol
for the function which returns the union of the two sets ---
ok-predicate
okset : setnum
Laws  $\forall n, n1, n2 : \text{nat}; s, s1, s2 : \text{setnum}$ 
Laws characterizing isempty
S1 isempty( $\emptyset$ ) = true
S2 isempty(insert( $n, s$ )) = false
Laws characterizing isin
S3  $n1 = n2 \Rightarrow \text{isin}(n1, \text{insert}(n2, s)) = \text{true}$ 
S4  $n1 \neq n2 \Rightarrow \text{isin}(n1, \text{insert}(n2, s)) = \text{isin}(n1, s)$ 
S5  $\text{isin}(n, \emptyset) = \text{false}$ 
Laws characterizing issubset
S6  $\text{issubset}(\emptyset, s) = \text{true}$ 
S7  $\text{isin}(n, s2) = \text{true} \Rightarrow$ 
     $\text{issubset}(\text{insert}(n, s1), s2) = \text{issubset}(s1, s2)$ 
S8  $\text{isin}(n, s2) = \text{false} \Rightarrow \text{issubset}(\text{insert}(n, s1), s2) = \text{false}$ 
Laws characterizing +
S9  $\text{insert}(n, s1) + s2 = \text{insert}(n, (s1 + s2))$ 
S10  $\emptyset + s = s$ 
Laws characterizing int
S11  $\emptyset - \text{int} - s = \emptyset$ 
S12  $\text{isin}(n, s2) = \text{true} \Rightarrow$ 
     $\text{insert}(n, s1) - \text{int} - s2 = \text{insert}(n, s1 - \text{int} - s2)$ 
S13  $\text{isin}(n, s2) = \text{false} \Rightarrow \text{insert}(n, s1) - \text{int} - s2 = s1 - \text{int} - s2$ 
Laws characterizing okset
S14 okset( $\emptyset$ )
S15  $\text{isin}(n, s) = \text{false} \Rightarrow \text{okset}(\text{insert}(n, s))$ 

```

The non-primitive sort is setnum, representing sets of natural numbers, and the primitive sorts are the sorts of Boolean and Natnum (i.e. nat for Natnum, and boolean for Boolean). The symbols isempty, isin, and issubset are non-primitive symbols of primitive sort boolean. Characterizing sets are preceded by headings (eg. the characterizing set for -int- is {S11, S12, S13}). The following are examples of relations in \langle_h : $\text{succ} \langle_h f$ and $\ll \langle_h f$, where f is a non-primitive symbol, $\text{isin} \langle_h \text{issubset}$, $\text{isin} \langle_h \text{isempty}$, and $\text{isin} \langle_h -\text{int}-$. The following points about the above RS are made briefly here, but will be extended upon in later sections of this chapter.

- The left hand side of the consequences of each law are of the form $f(c1, \dots, cn)$ where c_i ($1 \leq i \leq n$) is a constructor term. The form of the laws is in keeping with the notion of constructors as the only creators of new objects of the sort, implying that all terms of a sort should be expressible as a constructor term. This notion is formalized when the model and operational semantics for RSs are discussed.
- Note that the RS does not contain any laws expressing the commutativity of the functions + and -int-. Such laws are left implicit in the RS. How such laws are made explicit is described in Section 3.2

3.1.3 RS Schemas

Example 3.4 illustrates a RS characterizing the set of natural numbers. A similar characterization for sets of other objects can be made. Rather than build separate RSs for each such characterization, the similarity in the structure of the RSs can be used to derive a generic specification, from which particular RSs can be generated when provided with parameters. A RS schema is such a generic specification, and can have one of the following structures:

- $PS(P_1 \text{ with } \{\Sigma_1; L_1\}, \dots, P_n \text{ with } \{\Sigma_n; L_n\}) \equiv \text{Prim}_1 + \dots + \text{Prim}_m + \text{Signature Sig Laws } E$, where P_i is a RS name, called a *parameter name*, Σ_i ($1 \leq i \leq n$) is a signature, and L_i ($1 \leq i \leq n$) is a set of laws, called *constraints*, Prim_i is a primitive RS, Sig is the signature declaring the non-primitive sorts and symbols, E is a set of laws. An RS is generated from a schema of the above form by providing an RS, $\text{Par}_i = \langle \Sigma_{pi}, E_{pi}, SP_{pi} \rangle$ for each P_i , such that Σ_i is a subset of Σ_{pi} , and L_i is a subset of E_{pi} , for $1 \leq i \leq n$. Such RSs are called RS parameters of the schema. The result is an RS which is the smallest extension of the RS parameters and the hierarchical specification on the right of the \equiv symbol.
- $PS(\text{Par}_1, \dots, \text{Par}_n \text{ where } \text{Par}_1 \text{ is } [P_{11}, \dots, P_{1p}], \dots, \text{Par}_n \text{ is } [P_{n1}, \dots, P_{nq}]) \equiv \text{Prim}_1 + \dots + \text{Prim}_m + \text{Signature Sig Laws } E$, where Par_i ($1 \leq i \leq n$) is a parameter name, and P_{ij} ($1 \leq i \leq n$) is a hierarchical RS. This is a more restrictive form of a schema than the one given above since only the hierarchical RSs associated with the parameter names by is can be used as RS parameters for the names. The result, as in the previous case, is an RS which is the smallest extension of the RS parameters and the hierarchical specification on the right of the \equiv symbol.

Example 3.5

A RS schema for sets, based on the primitive RS, Boolean, characterizing a two-valued boolean algebra

```

Set(Element with (Signature sorts elem))  $\equiv$  Boolean +
Signature
  sorts set
  constructors
     $\emptyset$  :  $\rightarrow$  set
    insert : elem, set  $\rightarrow$  set
  auxiliary functions
    isempty : set  $\rightarrow$  boolean
    isin : elem, set  $\rightarrow$  boolean
    issubset : set, set  $\rightarrow$  boolean
    -int-, _+ : set, set  $\rightarrow$  set
Laws  $\forall e, e1, e2:elem; s, s1, s2:set$ 
  Laws characterizing isempty
  S1 isempty( $\emptyset$ ) = true
  S2 isempty(insert(e, s)) = false

```

Example 3.5 continued

Example 3.5 (continued)

A RS schema for sets, based on the primitive RS, Boolean, characterizing a two-valued boolean algebra

Laws characterizing isin

S3 $\text{isin}(e, \text{insert}(e, s)) = \text{true}$

S4 $e_1 \neq e_2 \Rightarrow \text{isin}(e_1, \text{insert}(e_2, s)) = \text{isin}(e_1, s)$

S5 $\text{isin}(e, \emptyset) = \text{false}$

Laws characterizing issubset

S6 $\text{issubset}(\emptyset, s) = \text{true}$

S7 $\text{isin}(e, s_2) = \text{true} \Rightarrow$

$\text{issubset}(\text{insert}(e, s_1), s_2) = \text{issubset}(s_1, s_2)$

S8 $\text{isin}(e, s_2) = \text{false} \Rightarrow \text{issubset}(\text{insert}(e, s_1), s_2) = \text{false}$

Laws characterizing +

S9 $\text{insert}(e, s_1) + s_2 = \text{insert}(e, (s_1 + s_2))$

S10 $\emptyset + s = s$

Laws characterizing int

S11 $\emptyset - \text{int} - s = \emptyset$

S12 $\text{isin}(e, s_2) = \text{true} \Rightarrow$

$\text{insert}(e, s_1) - \text{int} - s_2 = \text{insert}(e, s_1 - \text{int} - s_2)$

S13 $\text{isin}(e, s_2) = \text{false} \Rightarrow \text{insert}(e, s_1) - \text{int} - s_2 = s_1 - \text{int} - s_2$

Laws characterizing okset

S14 $\text{okset}(\emptyset)$

S15 $\text{isin}(e, s) = \text{false} \Rightarrow \text{okset}(\text{insert}(n, s))$

3.2 Model-theoretic interpretation of RSs

In formulating the laws of a RS, certain information, in the form of *assumed* inequalities and equalities between ground constructor terms, and n-relations, is left implicit. The set of inequations, equations, and n-relations that are left implicit in RS laws are called *assumptions*. Of the models which satisfy the laws of an RS, only those that also satisfy the assumptions of the RS are of interest. Of these algebras, the finitely generated algebras which satisfy the hierarchy-constraints are considered as useful semantic models of RSs. Specifically, algebras whose elements are all generated by constructors only, are desirable, since they provide a formalization of the intuitive notion of a constructor as the sole creators of objects.

For a hierarchical RS, $\langle \Sigma, E, SP \rangle$, one is thus interested in the algebras in $\text{HAlg}(\Sigma, E, SP)$ which also satisfy the assumptions. Let $\alpha = \mu + \mu' + \eta$, where μ represents the inequality assumptions, μ' represents the equality assumptions, and η represents the n-relation assumptions. $M_{\Sigma, E + \alpha}$ represents the subclass of algebras in $\text{HAlg}(\Sigma, E, SP)$ that also satisfy the assumptions α . The algebras in $M_{\Sigma, E + \alpha}$ are called the *models* of the associated RS.

3.2.1 Equality and inequality assumptions

The approach to specifying inequality and equality assumptions is adapted from the work of Mohan et al [MS87]. In their work, inequality assumptions are

inequalities between all ground constructor terms, that is, $\mu = \{ \langle x, y \rangle \mid x = c_1(x_1, \dots, x_n); y = c_2(y_1, \dots, y_n); c_1, c_2 \in C_s; x_1, \dots, x_n, y_1, \dots, y_n \in Tc(F) \}$, where C_s is the set of constructors of sort s , and $Tc(F)$ is the set of ground constructor terms derived from the symbols in F . This approach assumes a sub language of constructor terms which is 'free' in the sense that all constructor terms are assumed distinct.

The approach used here assumes a sub language of defined ground constructor terms² which are not all distinct. In such a sub language the constructor terms are not all distinct, for example, in a specification of sets, not all set constructor terms should be assumed distinct. In order to determine which ground constructor terms are equal, and which are distinct, a well-defined mapping, called a *normalizing function*, is associated with each sort of a RS. A normalizing function for a sort s takes a ground constructor term of sort s and returns a ground constructor term of sort s , called its *normal term*.

All normal terms of a sort are considered distinct. This provides the basis for generating equality and inequality assumptions as follows:

- ground constructor terms which map into the same normal term are equal, while
- ground constructor terms which map into different normal terms are distinct.

Normalizing functions are defined below.

Definition 3.10

Normalizing functions

For any hierarchical RS, $HS = \langle \Sigma, E, SP \rangle$, where $\Sigma = \langle S, F \rangle$, there is associated a S -indexed family of *normalizing functions*, $N = \{ N_s : Tc(F)_s \rightarrow Tc(F)_s \mid s \in S \}$. N_s is a mapping from defined ground constructor terms of sort s to defined ground constructor terms of sort s , such that for any ground constructor term, $c(x_1, \dots, x_n)$, of sort s , where x_1, \dots, x_n are defined ground constructor terms (primitive or non-primitive) of sorts s_1, \dots, s_n respectively, $N_s(c(x_1, \dots, x_n)) = N_s(c(N_{s_1}(x_1), \dots, N_{s_n}(x_n)))$.

Normalizing functions can be based on a total ordering on defined constructor terms, in which case they simply order the sub terms of their ground constructor term arguments in order to derive a unique normal term. An example of a normalizing functions is given in Example 3.6.

² All subsequent references to constructor terms in this section are actually references to defined constructor terms. Undefined constructor terms are considered equal.

Example 3.6

Normalizing functions for sets

Consider two defined constructor terms of sort `setnum` given in Example 3.4: `insert(e,insert(d,insert(f,empty)))` and `insert(f,insert(d,insert(e,empty)))`, where `e`, `d`, and `f` are of sort `natnum`. They should have equal interpretations in any model for `Setnum`, since they represent sets with the same elements, thus they should be mapped to the same normal term by the normalizing function for the sort `setnum`. The ordering $<$ characterized in `Natnum` can be used as a basis for the normalizing function for `set`. Thus, if $d < e < f$, and a normal term of `setnum` is defined as a set of natural numbers in ascending order, the two ground constructor terms are mapped into the normal term `insert(d,insert(e,insert(f,empty)))` by the normalizing function for `setnum`.

The equality and inequality assumptions for a RS are made explicit in the manner defined in Definition 3.11.

Definition 3.11

Inequality and equality assumptions

Given a hierarchical RS, $\langle \Sigma, E, SP \rangle$, where $\Sigma = \langle S, F \rangle$, the *equality assumption set* associated with the RS is the set $Q = \mu'_{p_1} + \dots + \mu'_{p_n} + \mu'$, where $\mu'_{p_1} + \dots + \mu'_{p_n}$ is the union of the equality assumption sets of the primitive RSs P_1, \dots, P_n in SP , and μ' is the union of a family of S -indexed sets consisting of sets $\mu'_s = \{ \langle x, y \rangle \mid N_s(x) = N_s(y); x, y \in s \}$, where N_s is the normalizing function for $s \in S$. The *inequality assumption set* associated with the RS is the set $I = \mu_{p_1} + \dots + \mu_{p_n} + \mu$, where $\mu_{p_1} + \dots + \mu_{p_n}$ is the union of the inequality assumption sets of the primitive RSs P_1, \dots, P_n in SP , and μ is the union of a S -indexed family of sets consisting of the sets $\mu_s = \{ \langle x, y \rangle \mid N_s(x) \neq N_s(y); x, y \in s \}$, where N_s is the normalizing function for $s \in S$.

To summarize, RSs are formulated partially based on *assumptions* made on the equality and inequality of terms in a sub language of ground constructor terms. To make such assumptions explicit, each RS is associated with a set of *normalizing functions* which generate *normal terms*. A pair of defined ground constructor terms which map to the same normal term is called an *equality assumption*, while a pair of defined ground constructor terms of the same sort mapping to different normal terms is called an *inequality assumption*. The set of all equality (inequality) assumptions of a RS is called the *equality (inequality) assumption set* of the RS.

3.2.2 Negated relation assumptions

N-relation assumptions concern the relationships that must hold amongst defined ground terms. They are based on an implicit operational interpretation for relations, where relations on ground terms whose truth cannot be 'deduced' are false. Deduction is based on an operational semantics for RSs detailed in the next section. The generation of these assumptions thus depends on the operational semantics of RSs.

3.3 An operational semantics for RSs

A useful operational semantics, in terms of conditional term rewriting systems (CTRSs), can be associated with RSs, provided the RSs satisfy certain syntactic conditions. The operational semantics is useful in the sense that it provides computationally effective representations of the objects abstractly characterized by the laws and assumptions of the RS. In this section it is shown that for any RS, $\langle \Sigma, E \rangle$, satisfying the syntactic conditions referred to above, rewriting in the derived CTRS is sound and complete with respect to $M_{\Sigma, E+\alpha}$ where α are the assumptions associated with the RS. Furthermore, such rewriting determines a canonical algebra in $M_{\Sigma, E+\alpha}$ consisting exactly of the normal terms generated by the normalizing functions (the "effective representation"). When such an operational semantics can be associated with a RS, the canonical algebra is taken as the semantic model for the RS. Note, however, that if an operational semantics cannot be associated with the RS, then a model-theoretic interpretation, in terms of $M_{\Sigma, E+\alpha}$ is not possible, since the n-relation assumptions in α cannot be made explicit.

The particular type of CTRSs and the sufficient conditions ensuring soundness and completeness, are detailed in the following sections. Some preliminary concepts and definitions are introduced here. A more detailed account of the following definitions, together with an introduction to CTRSs, can be found in Appendix I.

Let $T(\Sigma, X)$ be the set of all well-formed terms formed from the symbols in the signature $\Sigma = \langle S, F \rangle$, and the elements in X (called *variables*).

- The function *Var* takes a term in $T(\Sigma, X)$ and returns the set of variables occurring in it.
- A *substitution*, σ , is a mapping from X to $T(F, X)$, with $\sigma(x) = x$ almost everywhere. A *defined* substitution is a mapping from X to defined terms in $T(F, X)$ only. Substitutions are extended to morphisms of $T(\Sigma, X)$ as follows: $\sigma(f(t_1, \dots, t_n)) = f(\sigma t_1, \dots, \sigma t_n)$.

- The relative occurrence (or position) of a sub term in a term is represented by a sequence of positive integers, where the empty sequence is denoted by Λ . Let $O(t)$ denote the set of all sub term occurrences in the term t . The sub term of t at π , where π is a sequence of positive integers, denoted by $t|\pi$ is defined as follows:
 - If $t = x \in X$, the $O(t) = \Lambda$, and $t|\Lambda = t$.
 - If $t = f(t_1, \dots, t_n)$, then $O(t) = \{\Lambda\} + \{i\pi \mid 1 \leq i \leq n, \pi \in O(t_i)\}$, $t|\Lambda = t$, and $t|i\pi = t_i|\pi$.
- The *replacement* of a sub term at an occurrence π in t , by another term t' , is denoted by $t[\pi \leftarrow t']$.

3.3.1 Relational conditional term rewriting systems (R-CTRSs)

A relational conditional term rewriting system (R-CTRS), that can be associated with RSs is defined in Definition 3.12.

Normalizing functions and ok-predicates play the same roles in R-CTRSs as they do in RSs. A RS law can be transformed into a R-CTRS rule if the consequences of the laws can be oriented and the antecedents satisfy the conditions on R-CTRS rules given in Definition 3.12.

\rightarrow_R induces a relationship, called a *rewrite relation*, on the terms in $T(\Sigma, X)$, which can be informally described as follows: t rewrites to t' , or $t \rightarrow_R t'$ if there is a rule in the R-CTRS, with a consequence whose left hand side matches a sub term of t , after suitable substitution, and whose premises hold under the substitutions resulting from the match, such that t' is the result of replacing the matching sub term in t by the substituted rhs of the rule. Before a formal definition of rewriting is given, some notation is introduced.

Rewriting in zero or more steps, or the transitive closure of \rightarrow_R , is denoted by \rightarrow_R^* . If a term t is minimal with respect to \rightarrow_R^* (i.e. there is no t' such that $t \rightarrow_R^* t'$) then t is called a *normal form*. The set of all normal forms is denoted by \mathcal{N} . $t \rightarrow_N t'$ if and only if t' is a normal form and $t \rightarrow_R^* t'$. The following relationships are derived from \rightarrow_R :

- $t \Downarrow t'$ if and only if $\exists t_1$ such that $t \rightarrow_R^* t_1$ and $t' \rightarrow_R^* t_1$.
- $t \Uparrow t'$ if and only if $\exists t_1$ such that $t_1 \rightarrow_R^* t$ and $t_1 \rightarrow_R^* t'$.

Rewriting in a R-CTRS is formally defined in Definition 3.13.

Definition 3.12

Relational conditional term rewriting systems

A *relational conditional term rewriting system* (R-CTRS) is a triple $\langle \Sigma, RR, N \rangle$ consisting of:

- a signature, $\Sigma = \langle S, F, R, OK \rangle$, where S is a set of sorts, F is a set of function symbols, with a special subset of symbols called constructors, R is a set of relation symbols, and OK is a set of predicate symbols, one each for the sorts in S , called ok-predicates;
- a set of rules, RR , of the form $(u_i = v_i)_{i=1..l}, (u_i \neq v_i)_{i=1..n}, (r_i)_{i=1..o}, (\sim r'_i)_{i=1..p} \Rightarrow C$, where C (called the *consequence*) is one of the following forms:
 - (1) an oriented pair of F -terms, $t \rightarrow_R t'$, $t, t' \in T(F, X)$, where t is called the left hand side (lhs) and t' is called the right hand side (rhs), such that $(\text{Var}(u_i), \text{Var}(v_i))_{i=1..l}, (\text{Var}(u'_i), \text{Var}(v'_i))_{i=1..n}, \text{Var}(r_i)_{i=1..o}, \text{Var}(r'_i)_{i=1..p}, \text{Var}(t)$ are all subsets of $\text{Var}(t)$;
 - (2) an oriented pair, $r \rightarrow_R TT$, where $r \in T(R, X)$, such that $(\text{Var}(u_i), \text{Var}(v_i))_{i=1..l}, (\text{Var}(u'_i), \text{Var}(v'_i))_{i=1..n}, \text{Var}(r_i)_{i=1..o}, \text{Var}(r'_i)_{i=1..p}$ are all subsets of $\text{Var}(r)$; or
 - (3) an oriented pair $ok(t) \rightarrow_R TT$, where $ok \in OK$ and $t \in T(F, X)$, such that $(\text{Var}(u_i), \text{Var}(v_i))_{i=1..l}, (\text{Var}(u'_i), \text{Var}(v'_i))_{i=1..n}, \text{Var}(r_i)_{i=1..o}, \text{Var}(r'_i)_{i=1..p}$ are all subsets of $\text{Var}(t)$
- a set of partial functions, N , called *normalizing functions*, which map ground constructor terms to ground constructor terms. The set contains exactly one function for each sort in S .

Definition 3.13

Rewrite relation

A ground term t is said to *rewrite* to a ground term t' under a relational conditional term rewriting system (R-CTRS), R , denoted by $t \rightarrow_R t'$, if and only if there is a rule in R : $(u_i = v_i)_{i=1..l}, (u_i \neq v_i)_{i=1..n}, (r_i)_{i=1..o}, (\sim r'_i)_{i=1..p} \Rightarrow \text{lhs} \rightarrow_R \text{rhs}$ in R such that:

Match and replace

- there exists a *defined* substitution σ , and an occurrence π in t such that $t|\pi = \sigma\text{lhs}$ and $t' = t[\pi \leftarrow \sigma\text{rhs}]$,

Convergence of R-terms

- $(\sigma r_i \rightarrow_R^* TT)_{i=1..o}$,

Definition 3.13 continued

Definition 3.13 (continued)

Rewrite relation

Non-convergence of R-terms

- $(\text{NOT}(\sigma'_i \rightarrow^*_R \text{TT}))_{i=1..p}$,

Convergence of F-terms

- $(\text{ok}_{a_i}(\sigma u_i) \rightarrow^*_R \text{TT}, \text{ok}_{a_i}(\sigma v_i) \rightarrow^*_R \text{TT})_{i=1..i}$, and $((\sigma u_i \downarrow \sigma v_i), \text{or } (\sigma u_i \rightarrow^*_R c1, \sigma v_i \rightarrow^*_R c2), \text{ and } N_{a_i}(c1) = N_{a_i}(c2); c1, c2 \in \text{Tc}(F)_{a_i})_{i=1..i}$, where N_{a_i} is the normalizing function for the sort a_i , and

Non-convergence of F-terms

- $(\text{ok}_{b_i}(\sigma u'_i) \rightarrow^*_R \text{TT}, \text{ok}_{b_i}(\sigma v'_i) \rightarrow^*_R \text{TT})_{i=1..n}$ and $(\text{NOT}(\sigma u'_i \downarrow \sigma v'_i))_{i=1..n}$, and $(\sigma u'_i \rightarrow^*_R c1, \sigma v'_i \rightarrow^*_R c2 \Rightarrow N_{b_i}(c1) \neq N_{b_i}(c2); c1, c2 \in \text{Tc}(F)_{b_i})_{i=1..n}$, where N_{b_i} is the normalizing function for the sort b_i .

A ground term t , of sort s , is said to be *defined* in R , if and only if $\text{ok}_s(t) \rightarrow^*_R \text{TT}$, where ok_s is the ok-predicate for s .

Note that rewriting under a R-CTRS is defined only on ground terms. It can be extended to terms with variables by treating the variables as constants in the rewriting relationship. In such a situation the rewriting relationship is not closed since $t \rightarrow_R t'$ does not imply $\sigma t \rightarrow_R \sigma t'$ [Kap87]. For this reason, only term rewriting on ground terms is considered here.

To show the non-convergence of a R-term it is necessary to have a finite number of reduction steps starting from the R-term. Non-convergence then occurs when the final term in the reduction sequence is not TT. Similarly, to show the non-convergence of a pair of F-terms, t and t' , it is necessary to have a finite number of reduction steps starting each from t and t' . Non-convergence then occurs when the reducts generated by t are all distinct from the reducts generated by t' , and for any ground constructor term reducts $c1$ generated by t and $c2$ generated by t' , the normal terms corresponding to them are distinct. Thus there may be cases where it cannot be determined that $t \rightarrow_R t'$ in a R-CTRS, R , as illustrated in Example 3.7.

Example 3.7

Examples of situations where convergence and non-convergence of rewriting cannot be determined

The example CTRS given below is taken from [MS87]. In the R-CTRS $R = \{f(x) \neq g(y) \Rightarrow h(x,y) \rightarrow c; f(a) \rightarrow g(a); g(a) \rightarrow f(a)\}$, it is not possible to conclude that $h(a,b) \rightarrow c$ since there is an infinite reduction sequence starting from $f(a)$ thus reduction from $h(a,b)$ does not terminate. It is not also possible to conclude that $\text{NOT}(h(a,b) \rightarrow c)$ since $f(a)$ and $g(b)$ cannot be shown to converge.

\rightarrow_R is said to be *noetherian*, or terminating, if and only if there is no infinite sequence $t_1 \rightarrow_R t_2 \rightarrow_R \dots \rightarrow_R t_n \rightarrow_R \dots$ (i.e. when \rightarrow_R is well-founded). An important property of term rewriting systems is confluence. \rightarrow_R is said to be *confluent* if and only if, $\forall t, t' \ t \downarrow t' = t \uparrow t'$. In a confluent and terminating CTRS every term is rewritten to a unique normal form.

3.3.2 Sufficient conditions for termination and confluence of R-CTRSs

Termination and confluence are properties R-CTRSs need to have in order to be sound and complete, as is shown later. In what follows, sufficient conditions for ensuring termination, and confluence are given.

In unconditional term rewriting systems, one-step rewriting obviously terminates. This is not the case for CTRSs, since one-step rewriting also involves recursive calls to the evaluation procedure for evaluating the premises of the rule. Infinite calls to the evaluating procedure are thus possible. In order to avoid such infinite calls Kaplan suggests the use of a *simplification ordering* on terms, which makes the literals in the antecedents of rules, in some sense, "simpler" than their consequences [Kap84]. A similar type of ordering, based on the relationship $<_h$ on function symbols, defined in Definition 3.8, is used here. The following proposition states how such an ordering can be used to check for termination of rewriting in a R-CTRS.

Proposition 3.1 An R-CTRS, R , is terminating if:

- (1) $<_h$ is a partial ordering on $T(F)$
- (2) for every rule with consequent $f(s) \rightarrow rhs$, every sub term of rhs and every sub term of the terms appearing in the premises, $g(t)$, is either:
 - $g <_h f$, or
 - $\text{NOT}(f <_h g)$, and $t <_h s$, where \ll_h is the multi-set ordering on terms based on $<_h$. (See Appendix I)

The proof of Proposition 3.1 can be found in Appendix II.

Example 3.8

An example of termination

The set of rules $\{0 < \text{succ}(x) = \text{true}; x < y = \text{true} \Rightarrow \text{succ}(x) < \text{succ}(y) = \text{true}\}$, taken from the RS Natnum in Example 3.3, is terminating since $\text{true} <_h <$, and $\{x, y\} \ll_h \{\text{succ}(x), \text{succ}(y)\}$

A very strong sufficient condition for confluence is used here, somewhat similar to the one used in [MS87]. The condition ensures that two rules of a R-CTRS cannot be simultaneously applied to the same occurrence of a ground term resulting in two distinct terms. The severity of the above condition has the advantage that implementations of R-CTRSs are easier to design since, at any particular time, the rule to be used for reducing a given ground term is decidable. Furthermore, as will be seen later, the restriction provides a useful guideline for writing R-CTRSs (hence RSs) which are sufficiently complete. The above condition requires that arguments of the left hand sides of consequences must all be constructor terms only. This prevents overlaps between non-unifiable left hand sides. The condition also requires that if the left hand sides of two rules are unifiable and the corresponding instantiated right hand sides are distinct, then the antecedents of the rules cannot hold simultaneously. The conditions for confluence are formally stated below.

Proposition 3.2 A R-CTRS is confluent if the following conditions hold:

- (1) The consequences of a rule must have left hand sides with only constructor terms as *proper* sub terms (i.e. a lhs must be of the form $f(c_1, \dots, c_n)$ where c_1, \dots, c_n are constructor terms); and
- (2) Let $A1 \Rightarrow \text{lhs1} \rightarrow_{\text{R}} \text{rhs1}$ and $A2 \Rightarrow \text{lhs2} \rightarrow_{\text{R}} \text{rhs2}$, be any two rules in a R-CTRS such that there is a defined substitution, σ , which unifies lhs1 and lhs2 (i.e. $\sigma \text{lhs1} =_{\text{s}} \sigma \text{lhs2}$ where $=_{\text{s}}$ symbolizes syntactic equality). Then either:
 - $\sigma \text{rhs1} =_{\text{s}} \sigma \text{rhs2}$; or
 - there exists $u \neq v \in A1 + A2$ such that σu and σv have a common reduct, or $\sigma u \rightarrow_{\text{RC}}^* c1$, $\text{ok}_{\text{S}'}(c1)$, $\sigma v \rightarrow_{\text{RC}}^* c2$, $\text{ok}_{\text{S}'}(c2)$, and $N_{\text{S}'}(c1) = N_{\text{S}'}(c2)$, where $c1$ and $c2$ are ground constructor terms of sort s' , and $N_{\text{S}'}$ is the normalizing function for the sort; or
 - there exists $u = v \in A1 + A2$ such that $\text{NOT}(\sigma u \downarrow \sigma v)$, and if $\sigma u \rightarrow_{\text{RC}}^* c1$, $\text{ok}_{\text{S}}(c1)$, $\sigma v \rightarrow_{\text{RC}}^* c2$, $\text{ok}_{\text{S}}(c2)$, then $N_{\text{S}}(c1) \neq N_{\text{S}}(c2)$, where $c1$ and $c2$ are ground constructor terms of sort s , and N_{S} is the normalizing function for the sort; or
 - there exists $r \in A1 + A2$ such that $\text{NOT}(\sigma r \rightarrow_{\text{R}}^* \text{TT})$; or
 - there exists $\sim r' \in A1 + A2$ such that $\sigma r' \rightarrow_{\text{R}}^* \text{TT}$.

The proof of Proposition 3.2 can be found in Appendix II.

Example 3.9

Example of ground confluence

The RS Natnum, repeated below from Example 3.3, is ground confluent since the left hand sides of each rule contain only constructor terms or variables as proper sub terms, and at any time only one rule can be applied to a ground term.

```

Natnum ≡
Signature
  sorts nat
  constructors
    0: → nat
    succ: nat → nat
  auxiliary functions
    +_: nat, nat → nat
  ok-predicates
    oknat: nat
  relations
    <_: nat, nat
Laws  $\forall x, x1, x2: \text{nat}$ 
  1. oknat(0)
  2. oknat(succ(x))
  3. x+0 = x
  4. x1+succ(x2) = succ(x1+x2)
  5. 0<succ(x) = true
  6. x<y = true  $\Rightarrow$  succ(x)<succ(y) = true

```

3.3.3 Correctness of R-CTRSs

A R-CTRS determines a RS where the normalizing functions of the R-CTRS become the normalizing functions of the derived RS. The n -relation assumption set, η , of the RS is defined as follows: $\eta = \{\tau_i(t_{1i}, \dots, t_{ni}) \mid \forall \sigma, \text{NOT}(\sigma\tau_i(t_{1i}, \dots, t_{ni}) \rightarrow^*_R \text{TT}); (\text{ok}_j(t_{ji}) \rightarrow^*_R \text{TT}, t_{ji} \in T(F)_{j=1, \dots, n})\}$, where ok_j is the ok -predicate for the sort of the ground constructor term c_{ji} . The conditions under which rewriting in a R-CTRS is sound and complete, with respect to the derived RS, or correctness criteria, are now given.

An R-CTRSs, R , are said to be *correct* if rewriting with R is sound and complete in the following sense.

Definition 3.14

Soundness and completeness of rewriting with R-CTRSs

Rewriting with a R-CTRS, R is *sound and complete with respect to a set of assumptions* α , if the following conditions hold:

- *Soundness* : for all relational terms $r \in T(R)$, $r \rightarrow^*_R \text{TT} \Rightarrow M_{\Sigma, E+\alpha} \models r$, and for all F-terms $t, t' \in T(F)$, $\text{ok}(t)$ and $\text{ok}(t')$ and $t \rightarrow^*_R t' \Rightarrow M_{\Sigma, E+\alpha} \models t = t'$;
- *Completeness* : for all relational terms $r \in T(R)$, $M_{\Sigma, E+\alpha} \models r \Rightarrow r \rightarrow^*_R \text{TT}$, and for all F-terms $t, t' \in T(F)$, $M_{\Sigma, E+\alpha} \models \text{ok}(t) \Rightarrow (M_{\Sigma, E+\alpha} \models t = t' \Rightarrow t \rightarrow^*_R t')$.

In general termination and confluence are not sufficient conditions to ensure the correctness of a R-CTRS. An additional criteria, which supports the intuitive interpretation of constructors as generators of carrier sets, is that every defined non-constructor term in a R-CTRS, R , is reducible to a unique defined ground constructor term.

Definition 3.15

Sufficient completeness of R-CTRSs

A R-CTRS, R , is said to be *sufficiently complete* if and only if the following conditions hold:

- R is terminating and confluent; and
- for every non-constructor term, $f \in T(F)$, $\text{ok}(f) \rightarrow^*_R \text{TT} \Rightarrow f \rightarrow^*_R c$, where c is a constructor term and $\text{ok}(c) \rightarrow^*_R \text{TT}$.

Proposition 3.3 An R-CTRS is correct if it is sufficiently complete.

The proof of Proposition 3.3 can be found in Appendix II. A sufficiently complete R-CTRS generates a canonical algebra in $M_{\Sigma, E+\alpha}$, whose carrier sets are exactly the defined ground constructor terms. A hierarchical RS which can be transformed into a sufficiently complete R-CTRS is called a *reducing* RS, and, has models which reflect its hierarchical structure (i.e. the models satisfy the hierarchy-constraints) since all defined terms are reducible to ground constructor terms, thus terms of a primitive sort, s , are reducible to constructor terms of sort s . This means that non-primitive functions do not create objects of primitive sorts.

3.4 Summary

This chapter detailed a new algebraic specification technique which forms the cornerstone of the formal framework developed in this thesis. The specifications generated by the technique are called positive-negative relational conditional specifications (RSs), since they involve conditional equations with relations, inequalities, and negated relations (n-relations). Inequalities and n-relations are provided with operational interpretations, from which a model interpretation can be derived based on the notion of an assumption. In the model-theoretic sense, assumptions are equations that are not explicitly stated in the presented form, but are implicit in the formulation of the RS. Inequality and equality assumptions are based on a sub language of defined ground constructor terms generated from normalizing functions associated with the RS, while n-relation assumptions are derived from the operational interpretation of RSs. Only the algebraic models in which the RS laws and the assumptions are true, and whose structure matches the hierarchical structure of the RS (i.e. they satisfy the hierarchy-constraints) are considered as useful models of a RS. The class of such algebras, for a RS $\langle \Sigma, E \rangle$, is denoted by $M_{\Sigma, E+\alpha}$, where α is the set of assumptions.

The operational semantics is given in terms of a conditional term rewriting system (CTRS) called a relational CTRS (R-CTRS). Inequalities and n-relations are interpreted as the non-convergence of the terms involved. The sufficient completeness condition ensures that rewriting in a R-CTRS is sound and complete. The algebra of defined ground constructor terms generated by the R-CTRS can be taken as the model semantics for the RS.

CHAPTER 4

The Picture Level: Characterizing The Syntactic Aspects of DFDs

4.0 Introduction

The *Picture Level* (PL) is an algebraic theory characterizing the syntactic aspects of hierarchies of DFDs. The algebraic treatment of the syntactic aspects of DFDs entails viewing syntactic DFD structures as objects. The objects capturing the syntactic aspects of hierarchies of DFDs are called a *hierarchical data flow diagrams* (H_DFDs). H_DFDs and their components were introduced in Chapter 2, together with rules characterizing their structure. Such objects are said to be *structurally correct* if they satisfy their associated rules. The PL is a formalization of the rules characterizing structurally correct objects, stated in Chapter 2.

Specifically, the PL is a (positive-negative) relational specification (RS) named H_PLapplic, characterizing functions that construct, modify, or carry out observations on the syntactic objects. The laws of H_PLapplic are formal expressions of the rules characterizing H_DFDs and their components given in Chapter 2. The term representation of a syntactic object is called its *PL representation* and is taken as the formal textual representation of the object. Such formal representations, unlike graphical representations, are capable of being automatically analysed, for example, by term rewriting systems.

In this chapter, the structure of H_PLapplic and its model and operational semantics are described. Section 4.1 details the building of a RS, called SimpleApplic, which characterizes the syntactic aspects of flat DFDs. This section provides the essential flavour of the approach to characterizing the syntactic aspects of DFDs without the complexity introduced by hierarchies. In Section 4.2, SimpleApplic is extended to H_PLapplic by incorporating RSs characterizing hierarchies of data flows and processes into SimpleApplic.

H_PLapplic has an operational semantics in the form of a relational conditional rewriting system (R-CTRS) which is sufficiently complete (see Chapter 3). The canonical model generated by the operational interpretation provides the model semantics for H_PLapplic. The model consists of the PL representations of structurally correct DFD structures. Section 4.3 describes the operational and model semantics for H_PLapplic. Section 4.4 discusses the limitations of the PL.

The flat DFD at level 1 of the hierarchy of DFDs representing the library application shown in Example 2.1 in Chapter 2, is repeated here as Figure 4.1, and will be used for illustration purposes in this chapter.

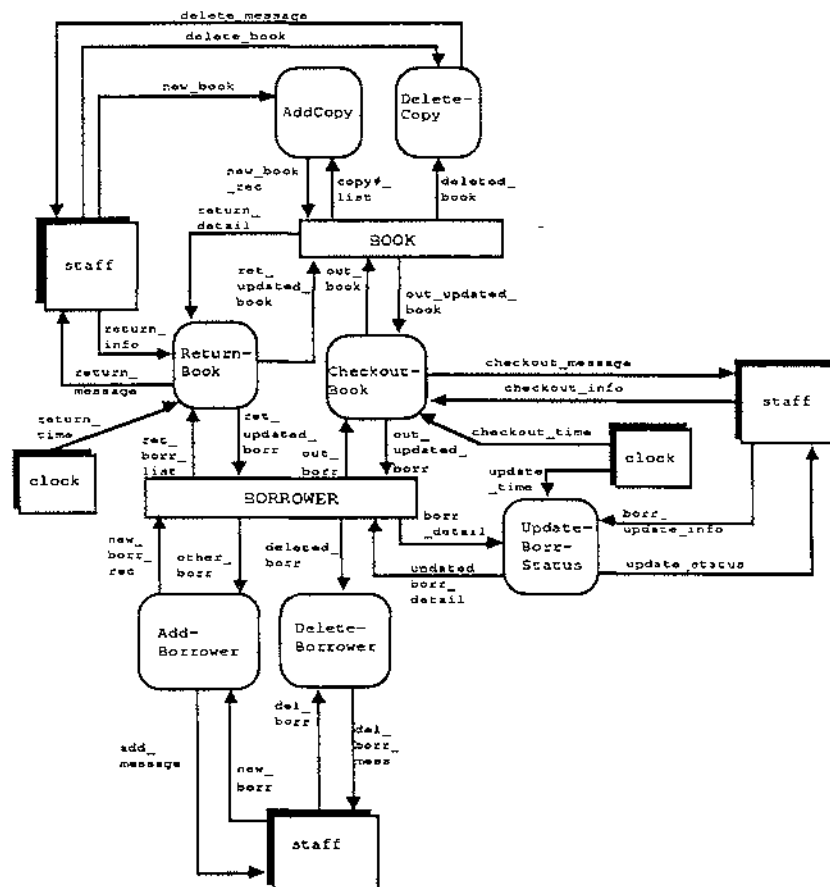


Figure 4.1 Level 1 of the hierarchy of DFDs describing the library application

4.1 Characterizing the syntactic aspects of flat DFDs

The RS characterizing structurally correct flat DFDs is based on primitive RSs characterizing structurally correct external environments (EEs) and process structures. In turn, the RS characterizing structurally correct EEs is based on a RS characterizing structurally correct external entities, while the RS characterizing structurally correct process structures is based on RSs characterizing structurally correct processes and data stores.

The following RSs are assumed available in what follows:

- **Boolean :** A RS characterizing truth values, with sort boolean, and constant functions true and false, where true \neq false.
- **Flowname:** A RS defining a finite set of data flow names of sort flowname.

- **Set :** A RS schema characterizing sets with the functions, `insert`, `+_+` (set union)¹, `_-diff-` (set difference), `_-int-` (set intersection), `isempty`, `isin`, and the constant function \emptyset (see Example 3.5 in Chapter 3).

The non-constructors of the RSs used by the PL can be categorized as *observation* and *auxiliary* functions. Observation functions are functions which return sub structures of their arguments, that is, they carry out observations on the objects in the arguments. All other functions are called auxiliary functions.

4.1.1 Characterizing structurally correct flat data flows

The syntactic aspects of data flows are characterized by the RS `PLflow`, shown in Figure 4.2. `PLflow` characterizes the type `plflow`² whose objects capture the syntactic aspects of data flows. The single law of the RS states that all `plflows` created from defined flownames are defined. For example, the PL representation of the structurally correct data flow `new_book` in Figure 4.1 is `mkflow("new_book")`, where "new_book" is a structurally correct flowname.

```

PLflow ≡ Flowname +
  Signature
    sort plflow
    constructor
      mkflow: flowname → plflow
    ok-predicate
      okflow : plflow
  Laws ∇ f:flowname
    F1.    okflow(mkflow(f))

```

Figure 4.2 The RS characterizing structurally correct data flows

4.1.2 Characterizing structurally correct flat processes

The syntactic aspects of processes are characterized by the RS `PLprocess`, shown in Figure 4.3. `PLprocess` characterizes the type `plprocess` whose objects encapsulate the syntactic aspects of processes. The constructor, `mkplprocess`, creates a `plprocess` from two sets of `plflows` representing the inputs and outputs of the process. For example, the PL representation of the process `ReturnBook` in Figure 4.1 is given in Example 4.1.

The law characterizing the ok-predicate, `okproc`, formally states the rule that a process is structurally correct if and only if its sets of inputs and outputs are disjoint and are both non-empty. The observation functions `getpinputs` and `getpoutputs` respectively return the set of input `plflows` and the set of output

¹ `_` marks the position of the argument for infix functions

² the "pl" stands for picture level and indicates that only the syntactic aspects of the data flow is of interest.

plflows of a pprocess. The effects of these observation functions on the PL representation of ReturnBook are given in Example 4.1.

```

PLprocess = Set(PLflow) +
  Signature
    sort pprocess
    constructor
      mkplprocess: set(plflow), set(plflow) → pprocess
    observation functions
      getpinputs, getpoutputs: pprocess → set(plflow)
    ok-predicate
      okproc : pprocess
  Laws  $\forall$  in,out:set(plflow)
    Law characterizing the ok-predicate
    PR1. isempty(in-int-out) = true, isempty(in) = false, isempty(out) = false  $\Rightarrow$ 
      okproc(mkplprocess(in,out))
    Laws characterizing the observation functions
    PR2. getpinputs(mkplprocess(in,out)) = in
    PR3. getpoutputs(mkplprocess(in,out)) = out

```

Figure 4.3 The RS, PLprocess, characterizing structurally correct processes

Example 4.1

PL representation for the process ReturnBook

```

ReturnBook =
  mkplprocess( {mkflow("return_info"), mkflow("return_time"), mkflow("return_detail"),
               mkflow("ret_borr_list)},
               {mkflow("return_message"), mkflow("ret_updated_book"),
               mkflow("ret_updated_borr")})

```

The effects of the observation functions on ReturnBook are as follows:

```

getpinputs(ReturnBook) = {mkflow("return_info"), mkflow("return_time"),
                          mkflow("return_detail"), mkflow("ret_borr_list")}

```

```

getpoutputs(P1) = {{mkflow("return_message"), mkflow("ret_updated_book"),
                    mkflow("ret_updated_borr")}}

```

4.1.3 Characterizing structurally correct flat external entities and data stores

The syntactic aspects of external entities and data stores are characterized by the RSs `PLentity` and `PLstore` shown in Figure 4.4 and Figure 4.5. `PLentity` characterizes the type `plentity` whose objects encapsulate the syntactic aspects of external entities, while `PLstore` characterizes the type `plstore` whose objects encapsulate the syntactic aspects of data stores. The constructors of these types create objects consisting of a set of input `plflows` and a set of output `plflows` respectively representing the inputs and outputs of the corresponding external entities and data stores. Like `PLprocess`, both `PLentity` and `PLstore` have observation functions which return the input `plflows` and output `plflows` of `plentities` and `plstores`. Example 4.2 gives the PL representations for the external entity `clock` and the data store `BOOK`.

The laws characterizing the `ok`-predicates for the `PLentity` and `PLstore` classes formally state the rules characterizing structurally correct external entities and data stores given in Chapter 2. These rules are repeated below.

- F1. A structurally correct data store has a non-empty set of inputs or a non-empty set of outputs. Its set of inputs and set of outputs are also disjoint.
- F7. A structurally correct external entity has a non-empty set of inputs or a non-empty set of outputs. Its set of inputs and the set of outputs are also disjoint.

F1 is expressed by the laws `EE1` and `EE2`, while F7 is expressed by the laws `DS1` and `DS2`.

```

PLentity ≡ Set(PLflow) +
  Signature
    sort plentity
    constructor
      mkplentity: set(plflow), set(plflow) → plentity
    observation functions
      geteinputs, geteoutputs: plentity → set(plflow)
    ok-predicate
      okentity : plentity
  Laws ∀ in,out:set(plflow)
    Laws characterizing the ok-predicate
    EE1. isempty(in-int-out) = true, isempty(in) = false ⇒ okproc(mkplentity(in,out))
    EE2. isempty(in-int-out) = true, isempty(out) = false ⇒ okproc(mkplentity(in,out))
    Laws characterizing the observation functions
    EE3. geteinputs(mkplentity(in,out)) = in
    EE4. geteoutputs(mkplentity(in,out)) = out

```

Figure 4.4 The RS `PLentity` characterizing structurally correct external entities

PLstore \equiv Set(PLflow) +

Signature

sort plstore

constructor

mkplstore: set(plflow), set(plflow) \rightarrow plstore

observation functions

getsinputs, getsoutputs: plstore \rightarrow set(plflow)

ok-predicate

okstore : plstore

Laws \forall in,out:set(plflow)

Law characterizing the ok-predicate

DS1. isempty(in-int-out), isempty(in) = false \Rightarrow okproc(mkplstore(in,out))

DS2. isempty(in-int-out), isempty(out) = false \Rightarrow okproc(mkplstore(in,out))

Laws characterizing the observation functions

DS3. getsinputs(mkplstore(in,out)) = in

DS4. getsoutputs(mkplstore(in,out)) = out

Figure 4.5 The RS PLstore characterizing structurally correct data stores

Example 4.2	
PL representations for clock and BOOK	
clock =	mkplentity(\emptyset , {mkflow("update_time"), mkflow("checkout_time"), mkflow("return_time")})
book =	mkplstore({mkflow("new_book_rec"), mkflow("out_updated_book"), mkflow("ret_updated_book")}, {mkflow("copy#_list"), mkflow("deleted_book"), mkflow("out_book"), mkflow("return_detail")})

4.1.4 Characterizing structurally correct process structures

Process structures are characterized by the RS Struct, shown in Figure 4.6. The objects of sort struct are process structures, and are built using three constructors: initstruct builds the simplest process structure consisting of exactly one plprocess; mkstruct1 builds a new process structure by adding a plprocess to a given process structure; and mkstruct2 builds a new process structure by adding a plstore to a given process structure. An example of the PL representation of a process structure is given in Example 4.3.

The laws, ST1, ST2, and ST3, characterizing the ok-predicate okstruct, are formal expressions of the rules governing the construction of structurally correct process structures given in Chapter 2 and repeated below:

- F3. A structurally correct process structure has at least one process. All processes in a structurally correct process structure are structurally correct and are uniquely identified by their inputs and outputs.
- F4. All data stores in a structurally correct process structure are structurally correct. All the inputs of a data store in a structurally correct process structure are also outputs of processes in the process structure, and all the outputs of a data store are also inputs of processes in the process structure. Furthermore, the set of data flows (inputs and outputs) of a data store in a structurally correct process structure is disjoint from the set of data flows of any other data store in the process structure. Data stores in a structurally correct process structure are uniquely identified by their inputs and outputs.
- F5. An output of a process in a structurally correct process structure is either associated with another process and/or data store in the process structure as an input, or is not associated with any process or data store in the process structure. An input of a process in a structurally correct process structure, on the other hand, may be associated with more than one process and/or data store in a process structure as an input.
- F6. A *net* or *boundary inputs* of a structurally correct process structure is an input associated with processes and data stores in the process structure that is not an output of process or data store in the process structure. A structurally correct process structure has at least one net input.

The observation functions `getinflows` and `getoutflows` return the set of inputs and the set of outputs, respectively, in a process structure, while `getininterface` returns the set of net inputs of a process structure. The function `getprocs` returns the set of `plprocesses` in a process structure, while `getstores` returns the set of `plstores` in a process structure. Examples of the application of the observation functions on a process structure are given in Example 4.3.

Struct utilizes two RSs, characterizing sets of processes and data stores, which are the results of instantiations of an RS schema, PLset shown in Figure 4.6. PLset extends the RS schema, Set, with functions which return the set of all inputs, outputs, and names of the parameter where the parameter is restricted to being one of PLEntity, PLstore, and PLprocess.

PLset(Set(PLelem) where PLelem is [PLEntity, PLstore, PLprocess]) =

Signature

observation functions

getallinputs, getalloutputs: set(plelem) \rightarrow set(plflow)

Laws $\forall e:plelem; se:set(plelem)$

S1. getalloutputs(insert(e,se)) = getoutputs(e)+getalloutputs(se)

S2. getalloutputs(\emptyset) = \emptyset

S3. getallinputs(insert(e,se)) = getinputs(e)+getallinputs(se)

S4. getallinputs(\emptyset) = \emptyset

Struct = PLset(PLprocess) + PLset(PLstore) +

Signature

sort struct

constructors

initstruct: plprocess \rightarrow struct

mkstruct1: plprocess, struct \rightarrow struct

mkstruct2: plstore, struct \rightarrow struct

observation functions

getinflows, getoutflows, getininterface: struct \rightarrow set(plflow)

getprocs : struct \rightarrow set(plprocess)

getstores: struct \rightarrow set(plstore)

ok-predicate

okstruct : struct

Laws $\forall p,p1:plprocess; ds:plstore; st:struct$

Laws characterizing the ok-predicate

ST1. okstruct(initstruct(p))

ST2. isempty(getpoutputs(p)-int-getoutflows(st)) = true \Rightarrow
okstruct(mkstruct1(p,st))

ST3. isempty(getsoutputs(ds)-int-getoutflows(st)) = true,
issubset(getsinputs(ds),getoutflows(st)) = true,
issubset(getsoutputs(ds),getinflows(st)) = true \Rightarrow
okstruct(mkstruct1(ds,st))

Laws characterizing the observation functions:

getprocs, getstores

ST4. getprocs(initstruct(p)) = insert(p, \emptyset)

ST5. getstores(initstruct(p)) = \emptyset

ST6. getprocs(mkstruct1(p1,st)) = p1+getprocs(st)

ST7. getprocs(mkstruct2(ds,st)) = getprocs(st)

ST8. getstores(mkstruct1(p1,st)) = getstores(st)

ST9. getstores(mkstruct2(ds,st)) = ds+getstores(st)

getoutflows, getinflows

ST10. getoutflows(mkstruct1(p1,st)) = getpoutputs(p1)+getoutflows(st)

ST11. getoutflows(mkstruct2(ds,st)) = getsoutputs(ds)+getoutflows(st)

ST12. getoutflows(initstruct(p)) = getpoutputs(p)

ST13. getinflows(mkstruct1(p,st)) = getpinputs(p)+getinflows(st)

ST14. getinflows(mkstruct2(ds,st)) = getsinputs(ds)+getinflows(st)

ST15. getinflows(initstruct(p)) = getpinputs(p)

getininterface

ST16. getininterface(st) = getinflows(st)-diff-getoutflows(st)

Figure 4.6 The RS characterizing structurally correct process structures

Example 4.3

The PL representation of the process struct for the level 1 DFD of the library application

To enhance readability the constructors associated with pflows will be left implicit, for example, `new_book = mkflow("new_book ")`.

The names of the PL representations for the processes are as follows:

```
AddCopy = mkplprocess({new_book, copy#_list}, {new_book_rec})
```

```
DeleteCopy = mkplprocess({delete_book, deleted_book}, {delete_message})
```

```
ReturnBook = mkplprocess({return_info, return_time,, ret_borr_list, return_detail},
                          {ret_updated_book, ret_updated_borr, return_message})
```

```
CheckoutBook = mkplprocess({out_borr, checkout_info, checkout_time, out_book},
                            {checkout_message, out_updated_borr, out_updated_book})
```

```
UpdateBorrStatus = mkplprocess({borr_update_info, update_time, borr_detail},
                               {updated_borr_detail, update_status})
```

```
DeleteBorrower = mkplprocess({del_borr, deleted_borr}, {del_borr_mess})
```

```
AddBorrower = mkplprocess({new_borr, other_borr}, {new_borr_rec, add_message})
```

The names of the PL representations for the data stores are as follows:

```
book = mkplstore({new_book_rec, out_updated_book, ret_updated_book}, {copy#_list,
                             deleted_book, out_book, return_detail})
```

```
borrower = mkplstore({ret_updated_borr, new_borr_rec, updated_borr_detail,
                     out_updated_borr}, {ret_borr_list, other_borr, deleted_borr,
                                         borr_detail, out_borr})
```

Example 4.3 continued

Example 4.3 (continued)

The PL representation of the process struct for the level 1 DFD of the library application

The PL representation, PSlib, for the process structure of the level 1 library DFD is:

```
PSlib = mkstruct2(book, mkstruct2(borrower, mkstruct1(AddCopy,
mkstruct1(DeleteCopy,
    mkstruct1(ReturnBook, mkstruct1(CheckoutBook, mkstruct1(UpdateBorrStatus,
    mkstruct1(AddBorrower, initstruct(DeleteBorrower))))))))))
```

The effects of the observation functions on PSlib are as follows:

```
getinflows(PSlib) = {new_book, copy#_list, delete_book, deleted_book, return_info,
    return_time,, ret_borr_list, return_detail, out_borr, checkout_info,
    checkout_time, out_book, borr_update_info, update_time,
    borr_detail, del_borr, deleted_borr, new_borr, other_borr,
    new_book_rec, out_updated_book, ret_updated_book,
    ret_updated_borr, new_borr_rec, updated_borr_detail,
    out_updated_borr }
```

```
getinflows(PSlib) = { new_book_rec, delete_message,
    ret_updated_book, ret_updated_borr,
    return_message, checkout_message, out_updated_borr,
    out_updated_book, updated_borr_detail, update_status,
    del_borr_mess, new_borr_rec, add_message, copy#_list,
    deleted_book, out_book, return_detail, ret_borr_list, other_borr,
    deleted_borr, borr_detail, out_borr}
```

```
getininterface(PSlib) = {new_book, delete_book, return_info,
    return_time, checkout_time,
    update_time, del_borr, new_borr, checkout_info, borr_update_info}
```

```
getprocs(PSlib) = {AddCopy, DeleteCopy, ReturnBook,
    CheckoutBook, UpdateBorrStatus,
    AddBorrower, DeleteBorrower}
```

```
getstores(PSlib) = {book, borrower}
```


4.1.5 The RS characterizing structurally correct flat DFDs

The RS, SimpleApplic, characterizing structurally correct flat DFDs is based on the RSs characterizing process structures, Struct, and sets of external entities, PLset(PLentity), as is shown in Figure 4.7. The objects of type plapplic, characterized by SimpleApplic, encapsulate the syntactic aspects of flat DFDs, and are built, via the constructor mkapplic, from a struct (process structure) and a set of plentities (the EE).

The rule F10 characterizing structurally correct flat DFDs given in Chapter 2 is repeated below, and is formally expressed by the laws of SimpleApplic. Example 4.4 gives the PL representation of the DFD in Figure 4.1. It is easily verified that it is structurally correct.

A flat DFD consists of a structurally correct process structure and a structurally correct EE (possibly empty) satisfying the following rule:

F10. The set of all outputs in the EE is equal to the set of the net inputs of the process structure, while the set of all inputs in the EE is a subset of the set of all outputs in the process structure. For a DFD with a non-empty EE, the result is that each data flow in the DFD is associated with a unique generator, and a non-empty set of receivers.

SimpleApplic \equiv Struct + PLset(PLentity) +

Signature

sort plapplic

constructor

mkapplic: struct, set(plentity) \rightarrow plapplic

ok-predicate

okapplic : plapplic \rightarrow boolean

Laws: \forall se:set(plentity); st:struct

- A1. $\text{isempty}(se) = \text{false}, \text{isempty}(\text{getallinputs}(se) - \text{int} - \text{getalloutputs}(se)) = \text{true},$
 $\text{getalloutputs}(se) = \text{getinterface}(st), \text{issubset}(\text{getallinputs}(se),$
 $\text{getoutflows}(st)) \Rightarrow \text{okapplic}(\text{mkapplic}(st, se))$
- A2. $\text{isempty}(se) = \text{true} \Rightarrow \text{okapplic}(\text{mkapplic}(st, se))$

Figure 4.7 The RS SimpleApplic characterizing non-hierarchical DFDs

Example 4.4

The PL representation of the DFD at level 1 of the library DFD

The PL representations of the external entities in the EE are:

```

staff = mkplentity({delete_message, return_message,
                   checkout_message, update_status,
                   del_borr_mess, add_message},
                  {return_info, delete_book, checkout_info,
                   borr_update_info, del_borr,
                   new_book, new_borr})
clock = mkplentity(∅, {update_time, checkout_time, return_time})

```

The PL representation for the DFD is:

```

Lib = mkapplic(PSlib, {staff, clock})

```

4.2 Characterizing the syntactic aspects of hierarchical DFDs (H_DFDs)

In this section, the RS SimpleApplic is modified to a RS characterizing syntactically correct H_DFDs, called H_PLapplic. The modifications concern the characterization of the hierarchical structures of data flows and processes ignored in SimpleApplic.

4.2.1 Characterizing structurally correct hierarchical data flows

Decomposition of data flows results in the revelation of their component data flows, as described in Chapter 2. Tree structures of data flows result from the successive decompositions of data flows. *Hierarchical data flows* encapsulate the syntactic aspects of such structures, and are formally characterized by the RS Flowstruct shown in Figure 4.8. The objects of type flowstruct characterized by Flowstruct are hierarchical data flows. Such objects are built up using three constructors: Nilfstruct, `_*_`, and `_|_`. Nilfstruct and `_|_` create objects of type fstruct, which, intuitively, are lists of hierarchical data flow, where Nilfstruct corresponds to an empty list. `_*_` creates objects of type flowstruct from a flowname and a fstruct representing the child decomposition set of the hierarchical data flow. An example of the PL representation for a hierarchical data flow is given in Example 4.5.

In Chapter 2 structurally correct hierarchical data flows were characterized by a single rule:

- Each sub data flow of a structurally correct hierarchical data flow is unique.

```

Flowstruct ≡ Set(Flowname) +
sorts fstruct, flowstruct
constructors
  Nilfstruct: → fstruct
  ·: flowname, fstruct → flowstruct
  |: flowstruct, fstruct → fstruct
observation functions
  getfstruct: flowstruct → fstruct
  getflow: flowstruct → flowname
  flat: fstruct → set(flowname)
ok-predicates
  okfstruct: fstruct
  okflowstruct: flowstruct
Laws ds:fstruct; d:flowstruct; f:flowname
Laws characterizing the observation functions
D1. getfstruct(f·ds) = ds
D2. getflow(f·ds) = f
D3. flat(Nilfstruct) = ∅
D4. flat(d|ds) = insert(getflow(d), flat(getfstruct(d))) + flat(ds)

Laws characterizing okfstruct
D5. okfstruct(Nilfstruct)
D6. isempty(insert(getflow(d), flat(getfstruct(d))) - int-flat(ds)) = true ⇒ okfstruct(d|ds)

Laws characterizing okflowstruct
D7. isin(f, flat(ds)) = false ⇒ okflowstruct(f·ds)

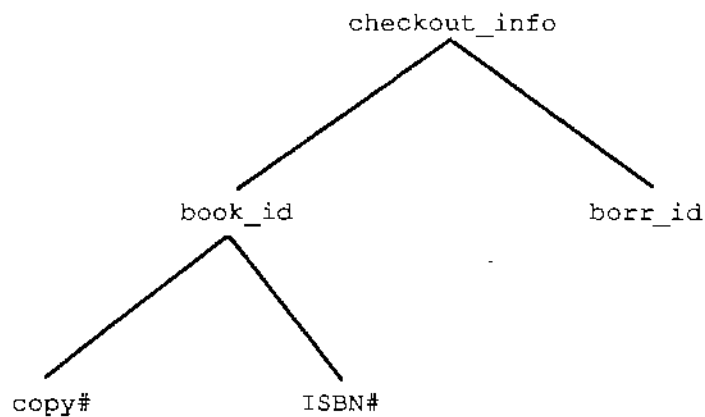
```

Figure 4.8 The RS Flowstruct characterizing structurally correct hierarchical data flows

The laws defining the ok-predicates for fstruct and flowstruct, D5 to D7, are formal expressions of the above rule for structural correctness. The following observation functions are used to express these laws:

- **getfstruct:** Returns the child decomposition set of a hierarchical data flow.
- **getflow:** Returns the flowname of a hierarchical data flow.
- **flat:** Returns the set of all sub data flows in a hierarchical data flow.

Law D5 states that an empty fstruct object is structurally correct, while D6 states that a non-empty fstruct is structurally correct if the flow names are all unique. D7 states that a flowstruct (hierarchical data flow) is structurally correct if its fstruct is structurally correct and the flowname of the flowstruct is not repeated in the fstruct.

Example 4.5PL representation of the hierarchical data flow `checkout_info`

The PL representation for hierarchical depicted above is:

```

checkoutinfo = checkout_info*(
    (book_id*(
        (copy#*Nilfstruct)|
        (ISBN*Nilfstruct)|
        Nilfstruct))
    (borr_id*Nilfstruct)|
    Nilfstruct)
  
```

The effects of the observation functions on `checkoutinfo` are given below:

```

getflow(checkoutinfo) = checkout_info
getfstruct(checkoutinfo) =
    (book_id*(copy#*Nilfstruct)|(ISBN*Nilfstruct)|Nilfstruct)|(borr_id*Nilfstruct)|
    Nilfstruct)
flat(getfstruct(checkoutinfo)) = {book_id, copy#, ISBN, borr_id}
  
```

In order to characterize the syntactic aspects of external entities, data stores, and (hierarchical) processes associated with hierarchical data flows, additional functions on flowstructs and sorts based on flowstructs are needed. The additional functions and sorts are provided by a RS named `ExtFlowstruct` shown in Figure 4.9.

The sorts `ininterface`, and `outinterface`, introduced by `ExtFlowstruct` are types whose objects are of the form $\langle df, \{f_1, \dots, f_n\} \rangle$, where df is a flowstruct,

called the *parent flow*, and f_1, \dots, f_n are flowstructs called *descendant flows*. These objects are used later, in the characterization of hierarchical processes, to relate the inputs and outputs of hierarchical processes to the decomposed flows in their bodies. An ininterface represents the partial decomposition relationship between an input (represented by the parent flow) of a hierarchical process and its decomposition set (whose elements are represented by the descendant flows). An outinterface represents the full decomposition relationship between an output (represented by the parent flow) of a hierarchical process and its decomposition set (whose elements are represented by the descendant flows). The relation \Downarrow , in ExtFlowstruct, represents the "is a full decomposition set of" relationship between a hierarchical data flow and a set of hierarchical data flows, and is defined by law E7. The definitions of full and partial decomposition sets given in Chapter 2 are repeated below:

A *full decomposition set*, F , of a hierarchical data flow D , is a set of sub hierarchical data flows of D satisfying the following conditions:

1. no two hierarchical data flows in F have common sub data flows; and
2. the set of all the primitive data flows in the hierarchical data flows in F is equal to the set of primitive data flows in D .

A *partial decomposition set* of a hierarchical data flow is simply a subset of its sub hierarchical data flows.

The following auxiliary and observation functions are also defined in ExtFlowstruct:

- **getleaves:** Returns the set of primitive data flows of a set of hierarchical data flows.
- **getflowstructs:** Returns the child decomposition set of a hierarchical data flow.
- **getsubstructs:** Returns the set of all sub hierarchical data flows in a hierarchical data flow.
- **disting:** Returns the value `true` if a set of hierarchical data flows is a distinguished set, else it returns the value `false`.
- **getinrhs:** Returns the set of descendant flows of an ininterface.
- **getoutrhs:** Returns the set of descendant flows of an outinterface.
- **getinlhs:** Returns the parent flow of an ininterface.
- **getoutlhs:** Returns the parent flow of an outinterface.
- **getallflows:** Returns the set of all sub data flows in a set of hierarchical data flows.

ExtFlowstruct \equiv Set(Flowstruct) +

Signature

sorts ininterface, outinterface

constructors

mk_{in}: flowstruct, set(flowstruct) \rightarrow ininterface

mk_{out}: flowstruct, set(flowstruct) \rightarrow outinterface

relations

\Downarrow : flowstruct, set(flowstruct)

observation functions

get_{inr}: ininterface \rightarrow set(flowstruct)

get_{outr}: outinterface \rightarrow set(flowstruct)

get_{inl}: ininterface \rightarrow flowstruct

get_{outl}: outinterface \rightarrow flowstruct

auxiliary functions

getleaves, getallflows : set(flowstruct) \rightarrow set(flowname)

getflowstructs : fstruct \rightarrow set(flowstruct)

getsubstructs: set(flowstruct) \rightarrow set(flowstruct)

disting: set(flowstruct) \rightarrow boolean

ok-predicates

okin: ininterface

okout: outinterface

Laws d,d1:flowstruct; ds,ds':fstruct; sd:set(flowstruct); f:pflow

Laws characterizing getleaves

E1. getleaves(\emptyset) = \emptyset

E2. getleaves(insert((f•Nilfstruct),sd)) =
insert(f,getleaves(getflowstructs(ds))+getleaves(sd))

E3. ds \neq Nilfstruct \Rightarrow
getleaves(insert(f•ds,sd)) = getleaves(getflowstructs(ds))+getleaves(sd)

Laws characterizing getflowstructs

E4. getflowstructs(djds) = insert(d, getflowstructs(ds))

E5. getflowstructs(Nilfstruct) = \emptyset

Laws characterizing getsubstructs

E6. getsubstructs(insert(f•ds,sd)) =
insert(f•ds,getsubstructs(getflowstructs(ds))+getsubstructs(sd))

Laws characterizing \Downarrow

E7. disting(sd), getleaves(getflowstructs(ds)) = getleaves(sd) \Rightarrow f•ds \Downarrow sd

Laws characterizing ok-predicates okout and okin

E8. d \Downarrow sd \Rightarrow okout(mkout(d,sd))

E9. issubset(sd,getsubstructs(insert(d, \emptyset))) = true \Rightarrow okin(mkin(d,sd))

Laws characterizing get_{inr}, get_{outr}, get_{inl}, and get_{outl}

E10. get_{inr}(mkin(d,sd)) = sd

E11. get_{outr}(mkout(d,ds)) = getflowstructs(ds)

E12. get_{inl}(mkin(d,sd)) = d

E13. get_{outl}(mkout(d,ds)) = d

Laws characterizing getallflows

E14. getallflows(insert(f•ds, sd)) = insert(f, flat(ds))+getallflows(sd)

E15. getallflows(\emptyset) = \emptyset

Laws characterizing disting

E16. isempty(insert(f, flat(ds))-int-getallflows(sd)) = true \Rightarrow
disting(insert(f•ds,sd)) = disting(sd)

E17. isempty(insert(f, flat(ds))-int-getallflows(sd)) = false \Rightarrow
disting(insert(f•ds,sd)) = false

E18. disting(\emptyset) = true

Figure 4.9 The RS ExtFlowstruct characterizing additional sorts and functions associated with the use of hierarchical data flows

The modified RSs characterizing the syntactic aspects of external entities and data stores with hierarchical inputs and outputs are given in Figure 4.10. The laws

of these RSs are formal expressions of the rules for hierarchical external entities and data stores given in Chapter 2 and repeated below.

- P1. A structurally correct (hierarchical) data store has a non-empty set of hierarchical inputs or a non-empty set of hierarchical outputs. The union of inputs and outputs of a data store is a distinguished set.
- H1. A structurally correct external entity has a non-empty set of inputs or a non-empty set of outputs. The union of inputs and outputs of an external entity is a distinguished set.

$H_PLentity \equiv Set(Flowstruct) +$

Signature

sorts hplentity

constructor

mkhplentity: set(flowstruct), set(flowstruct) \rightarrow hplentity

observation functions

geteinputs, geteoutputs: hplentity \rightarrow set(flowstruct)

ok-predicate

okhplentity: hplentity

Laws $\forall in, out: set(flowstruct)$

1. $disting(in+out) = true, isempty(in) = false \Rightarrow okhplentity(mkhplentity(in, out))$
2. $disting(in+out) = true, isempty(out) = false \Rightarrow okhplentity(mkhplentity(in, out))$
3. $geteinputs(mkhplentity(in, out)) = in$
4. $geteoutputs(mkhplentity(in, out)) = out$

$Set_HPLentity \equiv Set(H_PLentity) +$

Signature

observation functions

getalleinputs, getalleoutputs: set(hplentity) \rightarrow set(flowstruct)

Laws $\forall he: hplentity; se: set(hplentity)$

1. $getalleinputs(he, se) = geteinputs(he) + getalleinputs(se)$
2. $getalleoutputs(he, se) = geteoutputs(he) + getalleoutputs(se)$

$H_PLstore \equiv Set(Flowstruct) +$

Signature

sorts hplstore

constructor

mkhplstore: set(flowstruct), set(flowstruct) \rightarrow hplstore

observation functions

geteinputs, geteoutputs: hplstore \rightarrow set(flowstruct)

ok-predicate

okhplstore: hplstore

Laws $\forall in, out: set(flowstruct)$

1. $disting(in+out) = true, isempty(in) = false \Rightarrow okhplstore(mkhplstore(in, out))$
2. $disting(in+out) = true, isempty(out) = false \Rightarrow okhplstore(mkhplstore(in, out))$
3. $geteinputs(mkhplstore(in, out)) = in$
4. $geteoutputs(mkhplstore(in, out)) = out$

Figure 4.10 The RSs characterizing external entities and data stores with hierarchical inputs and outputs

4.2.2 Characterizing structurally correct hierarchical processes

Hierarchical processes encapsulate the syntactic aspects of process hierarchies resulting from successive process decompositions. From Chapter 2, a hierarchical process consists of a body, a set of inputs and a set of outputs. The body of a hierarchical process is a structure of (sub) hierarchical processes and data stores. The RS characterizing hierarchical processes `Procstruct` is shown in Figure 4.11. Hierarchical processes are objects of type `procstruct`, and are associated with four constructors: `Nilstruct`, `mkstruct`, `mkpstruct1`, and `mkpstruct2`. `Nilstruct`, `mkpstruct1`, and `mkpstruct2` create objects of type `pstruct` which are bodies of hierarchical processes. `Nilstruct` creates an empty body, `mkpstruct1` creates a new body from a given body by adding a hierarchical process to the body, and `mkpstruct2` creates a new body from a given body by adding a data store to it. The constructor `mkstruct` builds a hierarchical process given a `pstruct` (a body), a set of ininterfaces, and a set of outinterfaces. The ininterfaces and the outinterfaces explicitly state the relationships between the inputs and outputs of a hierarchical process and its internal data flows.

The rules characterizing structurally correct hierarchical processes given in Chapter 2 are repeated below:

- P2. A structurally correct body is either empty or contains at least one structurally correct (sub) hierarchical process. All data stores in a body are structurally correct.
- P3. No two hierarchical processes in a structurally correct body must have common sub processes.
- P4. The set of all data store inputs in a structurally correct body is a subset of the internal output set of the body, and the set of all data store outputs is a subset of the internal input set of the body. Furthermore, the receiver of a hierarchical data flow whose generator is a data store is never a data store.
- P5. Each hierarchical data flow in the internal output set has a unique generator in the body. The internal output set of a structurally correct body is a distinguished set.
- P6. There is at least one net input in a non-empty structurally correct body.
- P7. The set of inputs and the set of outputs of a structurally correct hierarchical process are both non-empty. Furthermore, the union of the inputs and the outputs of a hierarchical process is a distinguished set.
- P8. The body of a structurally correct hierarchical process is structurally correct. In a structurally correct hierarchical process with a non-empty body, an input corresponds to a subset of the net inputs in the body, called its *decomposition set*, which is a partial decomposition set of the input. The decomposition sets

of any two hierarchical data flows in the input interface are disjoint, and the union of the decomposition sets associated with the inputs of the hierarchical process is exactly the set of the net inputs of the body.

P9. For a structurally correct hierarchical process with a non-empty body, an output corresponds to a subset of the internal output set, called its *decomposition set*, which is a full decomposition set of the output. The decomposition sets of any two outputs is disjoint. If a hierarchical data flow in the internal output set of the body of a structurally correct hierarchical process is not in any decomposition set then it is directed towards hierarchical processes in the body.

Laws PS1 to PS3 express the rules P2 to P6 given above, where law PS1 states that an empty body is structurally correct, and law PS2 states that a structurally correct hierarchical process can only be added to a structurally correct body if:

- the outputs of all the sub processes and sub data stores of the hierarchical process are not also outputs in the body or outputs of sub processes and data stores of the hierarchical processes in the body; and
- the data flows generated by the sub processes of the hierarchical process which are not also outputs of the hierarchical process, are not sub data flows of the net inputs of the body.

Law PS3 states that a structurally correct data store can only be added to a structurally correct body if the data store is not already in the body, nor in the hierarchical processes of the body, and if the inputs of the data store are also outputs of hierarchical processes in the body, and outputs of the data store are also inputs of hierarchical processes in the body.

Laws PS4 and PS5 express the rules characterizing structurally correct hierarchical processes, where law PS4 states that a hierarchical process with an empty body (a primitive process) is structurally correct, and law PS5 expresses the rule characterizing structurally correct hierarchical processes with non-empty bodies.

The observation functions of `Procstruct` are informally described below:

- `getinflows` : Returns the set of all inputs in a body.
- `getoutflows` : Returns the set of all outputs in a body.
- `getnetinputs` : Returns the net inputs of a body.
- `getstores` : Returns the set of data stores in a process structure.
- `getinslhs` : Returns the set of parent flows in a set of ininterfaces.
- `getinsrhs` : Returns the set of descendant flows in a set of ininterfaces.
- `getoutslhs` : Returns the set of parent flows in a set of outinterfaces.

- `getoutsrhs` : Returns the set of descendant flows in a set of outinterfaces.
- `getinputs` : Returns the set of inputs of a hierarchical process (the parent flows of the ininterfaces of the hierarchical process).
- `getoutputs` : Returns the set of outputs of a hierarchical process (the parent flows of the outinterfaces of the hierarchical process).
- `getalloutflows` : Returns the set of flownames of all the outputs in the hierarchical processes of a body. In concrete terms, the function returns all the outputs of processes in the process tree representations of the hierarchical processes in a body.

Example 4.6 is an example of the PL representation of a structurally correct hierarchical process, and the effects of the observation functions on it.

`Procstruct` \equiv `ExtFlowstruct` + `H_PLstore` +

Signature

sorts `procstruct`, `pstruct`

constructors

`Nilstruct`: \rightarrow `pstruct`

`mkstruct`: `set(ininterface)`, `set(outinterface)`, `pstruct` \rightarrow `procstruct`

`mkpstruct1`: `procstruct`, `pstruct` \rightarrow `pstruct`

`mkpstruct2`: `hplstore`, `pstruct` \rightarrow `pstruct`

observation functions

`getinflows`, `getoutflows`, `getnetinputs`: `pstruct` \rightarrow `set(flowstruct)`

`getstores`: `pstruct` \rightarrow `set(plstore)`

`getoutputs`, `getinputs`: `procstruct` \rightarrow `set(flowstruct)`

`getinslhs`, `getinsrhs`: `set(ininterface)` \rightarrow `set(flowstruct)`

`getoutslhs`, `getoutsrhs`: `set(outinterface)` \rightarrow `set(flowstruct)`

`getalloutflows` : `pstruct` \rightarrow `set(flowname)`

ok-predicate

`okpstruct` : `pstruct` \rightarrow `boolean`

`okprocstruct`: `procstruct` \rightarrow `boolean`

Laws \forall `p:procstruct`; `st,sp:pstruct`; `ds:plstore`; `in, i1:set(ininterface)`; `out,o1:set(outinterface)`; `n:procname`

Laws characterizing okpstruct

PS1. `okpstruct(Nilstruct)`

PS2. `isempty(getalloutflows(st)-int-(getallflows(getoutslhs(out))+getalloutflows(sp))) = true`,
`isempty(getnetinputs(st)-int-(getalloutflows(sp)-diff-getallflows(getoutslhs(out)))) = true` \Rightarrow `okpstruct(mkpstruct1(mkstruct(in, out, sp), st))`

PS3. `isempty(getalloutflows(st)-int-getsoutputs(ds)) = true`,
`issubset(getsinputs(ds),getoutflows(st)) = true`,
`issubset(getsoutputs(ds),getinflows(st)) = true` \Rightarrow
`okpstruct(mkpstruct2(ds,st))`

Laws characterizing okprocstruct

PS4. `isempty(getinslhs(in)) = false`, `isempty(getoutslhs(out)) = false`,
`disting(getinslhs(in)+getoutslhs(out)) = true`, `getinsrhs(in) = \emptyset` ,
`getoutsrhs(out) = \emptyset` \Rightarrow `okprocstruct(mkstruct(in,out,Nilstruct))`

PS5. `sp \neq Nilstruct`, `isempty(getinslhs(in)) = false`, `isempty(getoutslhs(out)) = false`,
`disting(getinslhs(in)+getoutslhs(out)) = true`, `getinsrhs(in) = getnetinputs(sp)`,
`issubset(getoutsrhs(out),getoutflows(sp)) = true` \Rightarrow
`okprocstruct(mkstruct(in,out,sp))`

Laws characterizing getstores

PS6. `getstores(Nilstruct) = \emptyset`

PS7. `getstores(mkstruct1(mkstruct(n,in,out,sp),st)) = getstores(sp+getstores(st))`

PS8. `getstores(mkstruct2(ds,st)) = ds+getstores(st)`

Laws characterizing getinsrhs, getinslhs, getoutslhs, and getoutrhs

PS9. `getinslhs(insert(i1,in)) = insert(getinlhs(i1),getinslhs(in))`
 PS10. `getinslhs(\emptyset) = \emptyset`
 PS11. `getinsrhs(insert(i1,in)) = insert(getinrhs(i1),getinsrhs(in))`
 PS12. `getinsrhs(\emptyset) = \emptyset`
 PS13. `getoutslhs(insert(o1,out)) = insert(getoutlhs(o1),getoutslhs(out))`
 PS14. `getoutslhs(\emptyset) = \emptyset`
 PS15. `getoutsrhs(insert(o1,out)) = insert(getoutrhs(o1),getoutsrhs(out))`
 PS16. `getoutsrhs(\emptyset) = \emptyset`

Laws characterizing getoutputs and getinputs

P17. `getoutputs(mkstruct(in, out, sp)) = getoutslhs(out)`
 P18. `getinputs(mkstruct(in, out, sp)) = getinslhs(in)`

Laws characterizing getoutflows and getinflows

PS19. `getoutflows_(mkstruct1(p,st)) = getoutputs(p)+getoutflows_(st)`
 PS20. `getoutflows_(mkstruct2(ds,st)) = getsoutputs(ds)+getoutflows_(st)`
 PS21. `getoutflows_(Nilstruct) = \emptyset`
 PS22. `getinflows(mkstruct1(p,st)) = getinputs(p)+getinflows(st)`
 PS23. `getinflows(mkstruct2(ds,st)) = getsinputs(ds)+getinflows(st)`
 PS24. `getinflows(Nilstruct) = \emptyset`

Laws characterizing getnetinputs

PS25. `getnetinputs(Nilstruct) = \emptyset`
 PS26. `getnetinputs(st) = getinflows(st)-getoutflows(st)`

Laws characterizing getalloutflows

PS27. `getalloutflows(mkpstruct1(mkstruct(in, out, sp), st)) =
 (getallflows(getoutslhs(out))+getalloutflows(sp))+getalloutflows(st)`
 PS28. `getalloutflows(Nilstruct) = \emptyset`

Figure 4.11 The RS Procstruct characterizing hierarchical processes

Example 4.6

The PL representation for the hierarchical process `UpdateBorrStatus`

The process `UpdateBorrStatus` is decomposed into two primitive processes, `UpdateBorrRecord` and `GenerateFinesRecord` (see Chapter 2). The only non-primitive hierarchical data flow is `borr_update_info` which has the following PL representation:

```
borrinfo = borr_update_inf*((amount_paid*Nilstruct)|(update_id*Nilstruct)|Nilstruct)
```

The PL representations of the primitive processes are:

```
UpdateBorrRecord = mkstruct({<amount_paid,  $\emptyset$ >, <borr_fine_record,  $\emptyset$ >},  

  {<update_status,  $\emptyset$ >, <updated_borr_detail,  $\emptyset$ >}, Nilstruct)
```

```
GenFinesRecord = mkstruct({<update_id,  $\emptyset$ >, <update_time,  $\emptyset$ >, <borr_detail,  $\emptyset$ >},  

  {<borr_fine_record,  $\emptyset$ >}, Nilstruct)
```

The PL representation of the body consisting of the above two primitive processes is:

```
UpdateBody = mkpstruct1(GenFinesRecord, mkpstruct1(UpdateBorrRecord, Nilstruct))
```

Example 4.6 continued

Example 4.6 (continued)

The PL representation for the hierarchical process `UpdateBorrStatus`

The PL representation of the hierarchical processes is:

`UpdateBorrStatus =`

```
mkstruct({<borrinfo, {amount_paid•Nilfstruct, update_id•Nilfstruct}>,
  <update_time•Nilfstruct, {update_time•Nilfstruct}>,
  <borr_detail•Nilfstruct, {borr_detail•Nilfstruct}>}, {<update_status•Nilfstruct,
  {update_status•Nilfstruct}>, <updated_borr_detail•Nilfstruct,
  {updated_borr_detail•Nilfstruct}>, Nilstruct}
```

The effects of the observation functions on the above hierarchical process are given below:

`getinflows(UpdateBody) =`

```
{borr_fine_record•Nilfstruct, amount_paid•Nilfstruct, update_id•Nilfstruct,
  update_time•Nilfstruct, borr_detail•Nilfstruct}
```

`getoutflows(UpdateBody) =`

```
{update_status•Nilfstruct, updated_borr_detail•Nilfstruct, borr_fine_record•Nilfstruct}
```

`getnetinputs(UpdateBody) =`

```
{amount_paid•Nilfstruct, update_id•Nilfstruct, borr_detail•Nilfstruct,
  update_time•Nilfstruct}
```

`getstores(UpdateBody) = ∅`

`getoutputs(UpdateBorrStatus) = {update_status•Nilfstruct,`

```
  updated_borr_detail•Nilfstruct}
```

`getinputs(UpdateBorrStatus) = {borrinfo, update_time•Nilfstruct, borr_detail•Nilfstruct}`

`getailoutflows(UpdateBody) = {update_status, updated_borr_detail, borr_fine_record}`

4.2.3 The RS characterizing H_DFDs

`H_PLapplic`, the RS characterizing structurally correct `H_DFDs` shown in Figure 4.11, is obtained from `SimpleApplic`, by replacing the primitive RS `PLflow` by `Flowstruct`, replacing the RS `Struct` by `Procstruct`, and by replacing `PLentity` and `PLstore` by `H_PLentity` and `H_PLstore`, respectively.

Objects of the type `h_dfd` characterized by `H_PLapplic` are `H_DFDs`. The type is associated with a constructor `mkapplic`, which creates a `H_DFD` given a hierarchical process (`procstruct`) and an `EE` (`set(hplentity)`). The laws of the RS formally express the rule characterizing structurally correct `H_DFDs` given in Chapter 2 and repeated below:

H4. A structurally correct H_DFD consists of a structurally correct EE and a structurally correct hierarchical process. The set of all inputs (outputs) in the EE of a structurally correct H_DFD is equal to the set of inputs (outputs) of the hierarchical process of the H_DFD.

H_PLapplic \equiv Procstruct + Set(H_PLentity) +

Signature

sort h_dfd

constructor

mkapplic: procstruct, set(hplentity) \rightarrow h_dfd

ok-predicate

okapplic : h_dfd \rightarrow boolean

Laws: \forall se:set(hplentity); p \in procstruct

- A1. $\text{isempty}(se) = \text{false}, \text{isempty}(\text{getalleinputs}(se) - \text{int-getalleoutputs}(se)) = \text{true}, \text{getalleinputs}(se) = \text{getoutputs}(p), \text{getalleoutputs}(se) = \text{getinputs}(p) \Rightarrow \text{okapplic}(\text{mkapplic}(p, se))$
- A2. $\text{okapplic}(\text{mkapplic}(p, \emptyset))$

Figure 4.12 The RS H_PLapplic characterizing H_DFDs

4.3 Model and operational semantics for the PL

The models associated with the RS, as well as satisfying the explicit laws of the RS, also satisfy certain implicit laws arising from equality, inequality assumptions, and assumptions on relations, made when formulating the laws of the RS. Normalizing functions which transform terms to normal terms, where equal terms are transformed to the same normal term, and unequal terms are transformed to unequal normal terms, are used to express equality and inequality assumptions (see Chapter 3). A description of the normalizing functions associated with H_PLapplic follows.

Normalizing function for flowname

The RS characterizing flowname consists only of constructors for strings of alphanumeric characters. All ground constructor terms are assumed unique (i.e. two names are equal if and only if they are built in exactly the same way). Thus the identity function is the normalizing function for flownames. Flowames are also associated with an ordering based on an alphabetic and numeric ordering, in which alphabetic characters are less than numeric characters. Two strings are compared from left to right in the following manner: if the current character being checked in the first string is greater than the corresponding character in the second string then the first string is greater than the second string; if the two characters are the same then the next characters in the two strings are compared. For example, ap23d > apf5h since 2 > f. This ordering is used by normalizing functions for some of the other sorts of H_PLapplic.

Normalizing functions for flowstruct

Two hierarchical data flows are equal if they have the same name and their child decomposition sets are equal. The normalizing function for flowstruct orders the flownames at each level of a flowstruct using the ordering on flownames described earlier. For example the two flowstructs:

```
f1•(
  (f22•(
    (f221•Nilfstruct)|
    (f222•Nilfstruct)|
    Nilfstruct)|
  (f21•Nilfstruct)|
  (f23•Nilfstruct)|
  Nilfstruct), and f1•(
  (f23•Nilfstruct)|
  (f21•Nilfstruct)|
  (f22•(
    (f222•Nilfstruct)|
    (f221•Nilfstruct)|
    Nilfstruct)|
  Nilfstruct)
```

where $f21 < f22 < f23$, and $f221 < f222$, are both transformed to the normal term:

```
f1•(
  (f21•Nilfstruct)|
  (f22•(
    (f221•Nilfstruct)|
    (f222•Nilfstruct)|
    Nilfstruct)|
  (f23•Nilfstruct)|
  Nilfstruct)
```

Normalizing functions for hplentity and hplstore

Two hierarchical external entities (data stores) are equal if their sets of inputs and outputs are equal. The normalizing functions for hplentities and hplstores simply normalize their input and output sets of flowstructs.

Normalizing functions for ininterface and outinterface

Two in(out)interfaces are equal if and only if their parent flows are equal and their sets of descendant flows are equal. The normalizing functions for these sorts simply normalize the set of descendant flows.

Normalizing function for pstruct

Two bodies are equal if they contain the same hierarchical processes and data stores. The normalizing function for pstruct simply normalizes the hierarchical processes (see below) and data stores in the body.

Normalizing function for Procstruct

Two hierarchical processes are equal if they have equal ininterfaces, outinterfaces and bodies. The normalizing function for procstruct simply normalizes the in(out)interfaces and bodies of hierarchical processes.

The assumptions on negated relations are derived from the operational interpretation of H_PLapplic, described in the following section.

4.3.1 The PL R-CTRS

H_PLapplic can be converted to a R-CTRS by replacing the "=" symbol in the consequences of the laws to " \rightarrow ". This is possible since, in each law, the sets of variables of the literals in the antecedent are all a subset of the set of variables in the term on the left hand side of the equality symbol (or relation or ok-predicate) of the consequence, and the set of variables in the term on the right hand side of the equality symbol of the consequence is a subset of the set of variables in the term on the left hand side of the equality symbol.

Recall from Chapter 3 that a R-CTRS is sufficiently complete if and only if:

1. it is ground terminating and ground confluent; and
2. every defined non-constructor term rewrites (in one or more steps) to a ground constructor term (i.e. for a defined non-constructor term, $f \in T(F)$, $ok(f) \rightarrow^* TT \Rightarrow f \rightarrow^* c$, where c is a constructor term).

Since ok-predicates are characterized in terms of constructor terms only in the laws of H_PLapplic (see D5 to D7 of Flowstruct; E8, E9 of ExtFlowstruct; HE1, HE2 of H_PLentify; HS1, HS2 of H_PLstore; PS1 to PS5 of Procstruct; and A1, A2 of H_PLapplic), then if $ok(f) \rightarrow^* TT$, where f is a non constructor term, then f is reducible to a ground constructor term, c , such that $ok(c) \rightarrow TT$. Thus condition 2 above is satisfied Chapter 3 gives conditions for termination and confluence, which are repeated below:

An R-CTRS, R , is *terminating* if:

- (1) $<_h$ is a partial ordering on $T(F)$
- (2) for every rule with consequence $f(s) \rightarrow rhs$, every sub term of rhs and every sub term of the terms appearing in the premises, $g(t)$, is either:
 - $g <_h f$, or
 - $\text{NOT}(f <_h g)$, and $t \ll_h s$, where \ll_h is the multi-set ordering on terms based on $<_h$.

A R-CTRS is *confluent* if the following conditions hold:

- (1) The consequences of a rule must have left hand sides with only constructor terms as *proper* sub terms (i.e. a lhs must be of the form $f(c_1, \dots, c_n)$ where c_1, \dots, c_n are constructor terms); and
- (2) Let $A_1 \Rightarrow lhs_1 \rightarrow_R rhs_1$ and $A_2 \Rightarrow lhs_2 \rightarrow_R rhs_2$, be any two rules in a R-CTRS such that there is a defined substitution, σ , which unifies lhs_1 and lhs_2 (i.e. $\sigma lhs_1 =_s \sigma lhs_2$ where $=_s$ symbolizes syntactic equality). Then either:
 - $\sigma rhs_1 =_s \sigma rhs_2$; or
 - there exists $u' \neq v' \in A_1 + A_2$ such that $\sigma u'$ and $\sigma v'$ have a common reduct, or $\sigma u' \rightarrow^*_R c_1$, $ok_S(c_1)$, $\sigma v' \rightarrow^*_R c_2$, $ok_S(c_2)$, and $N_S(c_1) = N_S(c_2)$, where c_1 and c_2 are ground constructor terms of sort s' , and N_S is the normalizing function for the sort; or
 - there exists $u = v \in A_1 + A_2$ such that $\text{NOT}(\sigma u \downarrow \sigma v)$, and if $\sigma u \rightarrow^*_R c_1$, $ok_S(c_1)$, $\sigma v \rightarrow^*_R c_2$, $ok_S(c_2)$, then $N_S(c_1) \neq N_S(c_2)$, where c_1 and c_2 are ground constructor terms of sort s , and N_S is the normalizing function for the sort; or
 - there exists $r \in A_1 + A_2$ such that $\text{NOT}(\sigma r \rightarrow^*_R TT)$; or
 - there exists $\sim r' \in A_1 + A_2$ such that $\sigma r' \rightarrow^*_R TT$.

An inspection of the laws would show that the characterizing sets of non-constructors use only function and relation symbols that are already characterized in terms of other non-constructors at the same level or at lower levels (i.e. primitive non-constructors; see for example, the characterizing set of *getflowstructs* {E4, E5}, and the characterizing set for *getnetinputs* {PS25, PS26}). Also, primitive non-constructors are not characterized in terms of non-primitive function symbols. In the case of *ok*-predicates, recursive definitions, in terms of other *ok*-predicates at the same level are permitted (for example, see the characterizing sets for the constructors of data flow, {D5, D6, D7}, and process hierarchies, {PS1, PS2, PS3, PS4, PS5}). In such cases, inspection of the characterizing sets will reveal that the arguments of the constructors in the antecedents of the laws are simpler than the arguments on the left hand side of the equality symbol in the consequence, for example, in the law PS5, implicit in the antecedent is the literal *okpstruct(sp)*,

where $okpstruct$ and $okprocstruct$ are incomparable, but the arguments of $okpstruct$ are simpler than the arguments of $okprocstruct$, since $\{sp\} <_h \{mkstruct(in, out, sp)\}$. The relation $<_h$ on the function symbols of $H_PLapplic$ is thus a partial order and determines a simplification order on its ground terms. Thus termination of the R-CTRS corresponding to the $H_PLapplic$ is guaranteed.

The laws in the characterizing sets of the non-constructors and ok -predicates all have consequences whose left hand sides are have only constructor terms as arguments, thus satisfying condition (1) of the confluence conditions. Furthermore, no two rules may be applied to the same term, such that the left hand sides of the consequences of the rules match, but not their right hand sides. Thus $H_PLapplic$ is also ground confluent.

The R-CTRS generated from $H_PLapplic$ is thus sufficiently complete, providing an effective means by which the syntactic properties of DFDs can be investigated.

The following steps may be carried out in an investigation of the structural correctness of DFD structures:

1. Transform the construct to its PL representation, say c .
2. Find the set of laws characterizing the ok -predicate for the construct, ok , whose consequences can be matched with c . If the set is empty then the construct is not structurally correct. If the set is not empty then apply each rule to c , until either
 - a. a law is found such that $ok(c) \rightarrow TT$, in which case the construct is structurally correct; or
 - b. all laws in the set have been applied and none reduces $ok(c)$ to TT , in which case the construct is not structurally correct.

The operational interpretation of $H_PLapplic$ in terms of the R-CTRS, provides a formal basis for syntax analysis tools. Builders of such tools can use the PL to formally validate their tools.

4.4 Limitations of the PL

The PL provides a formal characterization of the syntactic aspects of DFD hierarchies and supports automated reasoning about such aspects via an operational interpretation. It has the potential to act as the formal basis for syntax analysis tools which take DFD structures and transform them into formal representations capable of being analysed. The PL in its present form though provides only a very basic foundation for an automated environment supporting the analysis of the syntactic aspects of DFDs. One notable limitation is that the PL provides very little support for the analysis of syntactic structure still undergoing development. For example, if

one decomposes the primitive processes of a hierarchy of DFDs further, it is viewed as the creation of a new hierarchical structure in the PL with no formal relationship to the hierarchy it was developed from. To enhance the use of the PL in this respect it would be useful to have a sub system of RSs in the PL which characterize modification functions (eg. adding, and deleting syntactic structures), and "is a refinement of" relationships between syntactic structures which have been decomposed further.

Another limitation related to the one above, is that currently the PL can only be used to reason about the structural correctness of complete structures. In the actual construction of DFDs one might also like to reason about incomplete syntactic structures, as well as reason about other properties of correct structures. By adding a special defined object, called a *void* object, in each sort, syntactic structures which are incomplete as a result of missing parts can be represented by placing the void objects of the appropriate sorts in the missing parts of the syntactic structures. The void objects indicate that the omissions were intentional, and 'completes' the syntactic structures so that the functions and relations which act only on complete syntactic structures, can also be applied to the incomplete constructs.

Relations, other than the ok-predicates characterizing structurally correct DFD structures can be added to express additional properties which may be of interest to tool developers and users. Such laws may be expressed using the observation, and auxiliary functions that already exist in the PL, or they may require additional functions to be added to the current PL.

The PL is useful only for the investigation of syntactic properties of DFDs. It does not provide semantic interpretations for the data objects in the DFD, nor does it provide behavioural interpretations for the processing components. Such interpretations are needed in order to fully specify the data and behavioural aspects of an application. Chapter 5 provides semantic interpretations for the structurally correct structures characterized in this chapter.

CHAPTER 5

The Specification Level: Deriving Behavioural Specifications from DFDs

5.0 Introduction

A number of researchers have proposed extensions to DFDs to support the specification of time-dependent behaviour. The tools and techniques based on such extensions lack the degree of formality required to support their use in the rigorous validation and verification of behavioural properties. The *Specification Level (SL)* of the formal framework for SA provides tools and techniques for pictorially describing and formally specifying the behaviour of applications, based on such formal foundations.

The derivation of the formal specification of behaviour from a hierarchy of DFDs goes through the following steps, as outlined in Chapter 2:

1. Generating a flat representation of the hierarchy of DFDs. Such a representation, called the *primitive DFD*, consists of the primitive processes, and all the data stores and external entities in the hierarchy of DFDs.
2. Introducing notation for describing state dependent behaviour into the primitive DFD, specifying the state dependent behaviour, and identifying actions, and state and asynchronous data flows to and from the external environment (EE). The result of this step is an ExtDFD.
3. Specifying the data types associated with the ExtDFD's data flows and data stores.
4. Specifying the behaviour of the ExtDFD's primitive processes and data stores.
5. Deriving the specifications of behaviours of the ExtDFD's actions from the specifications of behaviours of their constituent processes.
6. Deriving the specification of behaviour of the ExtDFD from the specifications of behaviour of its actions, data stores, and asynchronous data flows, and the specification of its state dependent behaviour.

This chapter describes the tools and techniques of the SL which are used in steps 3, 4, 5, and 6 above. Steps 1 and 2 were covered in Chapter 2. The use of the derived formal specification in the formal validation and verification of behavioural properties is also discussed in this chapter.

The SL consists of tools and techniques for formally specifying:

- (A) the static aspects of data flows and data stores in an ExtDFD, and
- (B) the dynamic aspects of data flows, data stores, processes and actions, in an ExtDFD.

The use of these techniques results in two types of specifications for ExtDFDs: the *Data Environment* (DE), and the *Behavioural Specification* (BS). The DE of an ExtDFD is the set of RSs resulting from the use of the techniques in (A). Such RSs characterize the object classes associated with the data flows and data stores in an ExtDFD, and their structures. In SA, such definitions were expressed quasi-formally in the data dictionary. The DE can be viewed as the formal counterpart of the data dictionary.

The BS of an ExtDFD is derived as a result of using the techniques in (B). It integrates specifications of the data aspects of an ExtDFD, provided by the DE, with specifications of the functional and control aspects of the ExtDFD. Behaviourally, actions and their constituent processes, data stores and asynchronous data flows are treated in the same manner, thus, allowing their specifications to be integrated in a "natural" way, that is, without resorting to techniques for bridging different specification tools. When only their behavioural aspects are of concern actions, ExtDFD processes, data stores and data flows are collectively called *processes*. To distinguish this use of the term process from its use in describing places of transformations in an ExtDFD, the latter use is qualified by the term ExtDFD, as is done in the previous sentence.

Furthermore, processes are treated as abstract data types (ADTs), thus permitting the integration of data specifications with specifications of behaviour. The BS of an ExtDFD is an algebraic characterization of an ADT representing the class of behaviours of the ExtDFD. A similar treatment of data and processes is used in the SMoLCS approach [AGR88]. Another approach to integrating data and process specifications based on ADTs can be found in Kaplan and Pnueli [KP87].

The SL techniques described in this chapter are demonstrated with the aid of the action `CheckoutBook` and its associated data stores, asynchronous data flows and state flows. The diagram is shown in Figure 5.1.

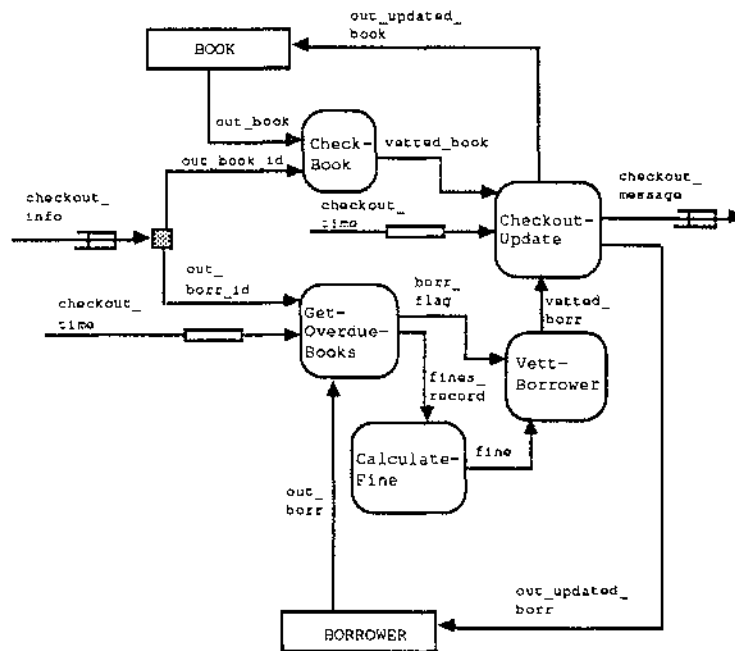


Figure 5.1 The ExtDFD for CheckoutBook

5.1 The Data Environment (DE)

When only the static aspects of data stores and data flows are of concern they are collectively called *data entities*. A data entity is associated with a class of objects, and data stores and asynchronous data flows are also associated with structures. The DE provides algebraic characterizations of the class of objects and structures of data entities, in the form of RSs. The RSs characterizing the object class of a data entity also include functions for 'splitting' objects into sub objects. Such functions are needed, for example, to establish the relationships between the input and outputs of splitters.

The algebraic characterizations of the structures associated with data entities include functions which observe and modify the structures.

5.1.1 Characterizing the object classes associated with data entities

The function symbols in a RS characterizing the object class of a data entity can be categorized as follows:

- *Constructors* : A constructor is a function which builds new objects of an object class.
- *Observation functions*: An observation function returns a sub object of an object.
- *Auxiliary functions*: Non-constructors which are not observation functions.

The RSs characterizing the object class of the data entities associated with CheckoutBook are given in Example 5.1. The characterizations are based on the type definitions given in Example 2.7 of Chapter 2, and repeated in Example 5.1.

Example 5.1

Characterizing the object classes of the data entities associated with the action

CheckoutBook

Type definitions for the data entities associated with the action CheckoutBook on which the specifications given in this example are based:

Non-base data types

```

bb_status          ::= <time_returned | "Not returned">
book               ::= <book_id, title, subject, author,
                      copy_type, borrower_indicator>
book_id           ::= <ISBN, copy#>
borr_detail       ::= [borrower_book_detail]
borr_fine_record  ::= <<number, borrower_id> | "Not in file"
borr_flag         ::= <"Not in file" | <borrower_id,
out_borr>>
borr_update_info  ::= <borrower_id, number>
borrower          ::= <borrower_id, borrower_name,
                      borrower_addr, borrower_type,
                      [borrower_book_detail],
                      payment_to_date>
borrower_book_detail ::= <book_id, due_time, bb_status>
borrower_id       ::= <[character]>
borrower_indicator ::= <"Available" | borrower_id>
checkout_info     ::= <book_id, borrower_id>
checkout_message  ::= <vetted_borr, vetted_book>
ISBN              ::= <[integer]>
out_book          ::= <borrower_indicator, copy_type>
out_book_id       ::= book_id
out_borr          ::= <[borrower_book_detail],
                      payment_to_date>
borrower_type,
out_updated_book ::= borrower_indicator
out_updated_borr ::= [borrower_book_detail]
vetted_book       ::= <<book_id, copy_type> | "book not in
                      file" | "book already checked out" |
                      "not borrowable">
vetted_borr       ::= <<"Fines over limit", number>|
                      "borrower not in file" |
                      <out_borr, borrower_id>>

```

Base data types

```

amount_paid       ::= number
author            ::= [character]
borrower_addr     ::= [character]
borrower_name     ::= [character]
borrower_type     ::= <"undergrad" | postgrad" | "staff">
checkout_time     ::= time
copy#             ::= integer
copy_type         ::= <"book" | "reference" | "periodical">
fine              ::= number
fines_record      ::= [number]
out_borr_id       ::= borrower_id
payment_to_date   ::= number
subject           ::= [character]
title             ::= [character]

```

Example 5.1 continued

Example 5.1 (continued)

Characterizing the object classes of the data entities associated with the action

CheckoutBook

The RSs characterizing the object classes associated with the data store BOOK

Copy_type ≡

Signature

sort copy_type

constructors

Book : → copy_type

--- corresponds to the "book" option in the type definition for copy_type ---

Ref : → copy_type

--- corresponds to the "reference" option in the type definition for copy_type ---

Per : → copy_type

--- corresponds to the "periodical" option in the type definition for copy_type ---

Book_id ≡ ISBN_code + Integer +

Signature

sort book_id

constructor

mkbkid : ISBN, integer → book_id

Borrower_indic ≡ Borrower_id +

Signature

sort borrower_indicator

constructors

mkbbind : borrower_id → borrower_indicator

Available : → borrower_indicator

--- corresponds to the "Available" option in the type definition for
borrower_indicator ---

Book ≡ Borrower_indic + Book_id + List(Character) + Time + Copy_type +

Signature

sorts book

constructor

mkbbook : book_id, list(char), list(char), list(char), copy_type,
borrower_indicator → book

Book characterizes the object class, **book**, of the data store BOOK. The three lists of characters in the domain of the constructor **mkbbook** represent the title, subject, and author of the book, respectively.

The RSs characterising the object classes associated with the data store
BORROWER

An RS **Borrower_id**, characterising the class of **borrower_id** objects, is assumed available in what follows.

Example 5.1 continued

Example 5.1 (continued)

Characterizing the object classes of the data entities associated with the action

CheckoutBookBorrower_book_detail \equiv Book_id + Time +**Signature****sorts** borrower_book_detail, bb_status**constructors**mkbbstatus : time \rightarrow bb_statusNotRet : \rightarrow bb_status

--- corresponds to the "Not returned " option in the type definition for

bb_status ---

Borrower_type \equiv **Signature****sort** borrower_type**constructors**Undergrad : \rightarrow borrower_typePostgrad : \rightarrow borrower_typeStaff : \rightarrow borrower_typeBorrower \equiv Borrower_id + List(Borrower_Book_Detail) + Number + List(Character) +
Borrower_type +**Signature****sort** borrower**constructor**

mkborrow : borrower_id, list(char), list(char), borrower_type,

list(borrower_book_detail), number \rightarrow borrower

Borrower characterizes the class of objects stored in the data store BORROWER. The two lists of characters in the constructor mkborrower represent the borrower's name and address, respectively, while the number represents the amount paid on fines by the borrower.

RSs characterizing the object classes associated with the data flows of CheckoutBookCheckout_info \equiv Book_id + Borrower_id +**Signature****sort** checkout_info**constructor**mkoutinfo : book_id, borrower_id \rightarrow checkout_info**observation functions**getbookid : checkout_info \rightarrow book_idgetborrid : checkout_info \rightarrow borrower_id**Laws** \forall bk:book_id; borr:borrower_id

1. getbookid(mkoutinfo(bk, borr)) = bk

2. getborrid(mkoutinfo(bk, borr)) = borr

Checkout_info defines the object class of the data flow checkout_info. The observation functions are associated with the splitting of the data flow into the sub data flows out book id (getbookid) and out borrower id (getborrid).

Example 5.1 continued

Example 5.1 (continued)

Characterizing the object classes of the data entities associated with the action

CheckoutBook

Out_book \equiv Copy_type + Borrower_indic +

Signature

sort out_book

constructor

mkoutbk : borrower_indic, copy_type \rightarrow out_book

Out_book characterizes the object class of the data flow out_book.

Vetted_book \equiv Book_id + Copy_type +

Signature

sort vetted_book

constructors

mkvbk : book_id, copy_type \rightarrow vetted_book

Bknotinfile : \rightarrow vetted_book

--- corresponds to the "book not in file" option in the type definition for vetted_book ---

CheckedOut : \rightarrow vetted_book

--- corresponds to the "book already checked out" option in the type definition for vetted_book ---

NotBorr : \rightarrow vetted_book

--- corresponds to the "not borrowable" option in the type definition for vetted_book ---

Vetted_book characterizes the object class for the data flow vetted_book.

Out_borr \equiv Borrower_type + List(Borrower_book_detail) + Number +

Signature

sort out_borr

constructor

mkoutbr : list(borrower_book_detail), borrower_type, number \rightarrow out_borr

Out_borr characterizes the object class of the data flow out_borr.

Borr_flag \equiv List(Borrower_book_detail) +

Signature

sorts borr_flag

constructors

Bflag : \rightarrow borr_flag

--- corresponds to the "Not in file" option in the type definition for borr_flag ---

mkbflag : borrower_id, out_borr \rightarrow borr_flag

Borrower_flag defines the object class of the data flow borr_flag.

Example 5.1 continued

Example 5.1 (continued)

Characterizing the object classes of the data entities associated with the action

CheckoutBook

Vetted_borr \equiv Borrower_flag + Number +**Signature****sorts** vetted_borr**constructors**mkerrvborr : number \rightarrow vetted_borr

--- corresponds to the "Fines over limit" option in the type definition for

vetted_borr ---

NoBorr : \rightarrow vetted_borr

--- corresponds to the "borrower not in file" option of the type definition for

vetted_borr ---

mkvetborr : out_borr, borrower_id \rightarrow vetted_borr

Vetted_borrower defines the object class of the data flow vetted_borrower.

Checkout_message \equiv Vetted_book + Vetted_borr +**Signature****sorts** checkout_message**constructor**mkoutmess : vetted_borr, vetted_book \rightarrow checkout_messageCheckout_message defines the object class of the data flow
checkout_message.**5.1.2 Characterizing the structure of data entities**

Asynchronous data flows and data stores may be associated with more than one object at any particular point in time. The structure of data stores and asynchronous data flows define the relationships between the objects associated with them. Structures are characterized in terms of *access functions* which modify and observe them.

Asynchronous Data Flows

Asynchronous data flows are associated with queue structures. The receivers of an asynchronous data flow receive objects from the top of the queue associated with the data flow, while the generator places an object at the bottom of the queue. The RS schema characterizing the generic structure of asynchronous data flows is given in Figure 5.2. The specification of the structure of a particular asynchronous flow is obtained by instantiating the schema with the RS characterizing the object class of the data flow.

```

Asynch(Element) ≡
  Signature
  sort asynch
  constructors
    emptyq : → asynch
    addq : elem, asynch → asynch
  observation function
    top : asynch → elem
  auxiliary function
    deleteq : asynch → asynch
  Laws  $\forall e:elem; q:asynch$ 
    1. deleteq(addq(e,emptyq)) = emptyq
    2.  $q \neq emptyq \Rightarrow deleteq(addq(e,q)) = add(e,deleteq(q))$ 
    3. top(addq(e,emptyq)) = e
    4.  $q \neq emptyq \Rightarrow top(addq(e,q)) = top(q)$ 

```

Figure 5.2 The RS schema characterizing the generic structure of asynchronous data flows

Data Stores

The access functions in a RS characterizing the structure of a data store can be classified as follows:

- *Read access functions* : functions which carry out observations on the structure. Such functions usually return an object in the structure, or an object indicating that a requested object is not in the structure.
- *Update functions* : functions which modify an existing object in the structure.
- *Add functions* : functions which add new objects to the structure.
- *Delete functions* : functions which delete existing objects from a structure.

Update and add functions are collectively referred to as *write access functions*. The access functions associated with the structure of a data store is determined by its inputs and outputs. Each input of a data store is associated with a write access function in the RS characterizing the structure of the data store, while each output is associated with a read and/or delete function in the RS. Example 5.2 gives the RS characterizing the structure of the data store BOOK, associated with the data flows shown in Figure 5.1.

Example 5.2

Characterizing the structure of the data store BOOK, shown in Figure 5.1

The data store BOOK has a single input, `out_updated_book`, and a single output, `out_book`, in Figure 5.1. The input `out_updated_book` represents an update on the data store and is associated with an update access function called `updatebk2` in the RS BookStore characterizing the structure of the data store BOOK. The output `out_book` represents a read access, and is associated with a read access function in BookStore called `readbook2`.

BOOK is a list of book objects. The type `readval` is the class of objects representing the status of a read access on the data store. Thus, if the read access is successful, that is, if the desired object is in the data store, then the `readval` object returned reflects the object retrieved otherwise the `readval` object returned, `Nullval`, reflects an error situation.

BookStore \equiv List(Book) + Out_book + Borrowe_indic +

Signature

sort `readval`

constructors

`Nullval` : \rightarrow `readval`

`mkreadval` : `out_book` \rightarrow `readval`

read-access function

`readbook2` : list(book), `book_id` \rightarrow `readval`

update function

`updatebk2` : list(book), `borrower_indicator`, `book_id` \rightarrow list(book)

Laws \forall `bid`, `bid'`:`book_id`; `t,s,a`:list(char); `ct`:`copy_type`; `lb`:list(book);
`indic`, `indic'`:`borrower_indicator`

Laws characterizing the read function associated with the output `out_book`

1. `bid = bid'` \Rightarrow `readbook2(mkbook(bid, t, s, a, ct, indic)||lb, bid')` =
`mkreadval(mkoutbk(indic, ct))`

2. `bid \neq bid'` \Rightarrow `readbook2(mkbook(bid, t, s, a, ct, indic)||lb, bid')` =
`readbook2(lb,bid')`

3. `readbook(emptylist, bid)` = `Nullval`

Laws characterizing the update function associated with the input `out_updated_book`

4. `bid = bid'` \Rightarrow
`updatebk2(mkbook(bid, t, s, a, ct, indic)||lb, indic', bid')`
= `mkbook(bid, t, s, a, ct, indic')||lb`

5. `bid \neq bid'` \Rightarrow
`updatebk2(mkbook(bid, t, s, a, ct, indic')||lb, indic', bid')`
= `mkbook(bid, t, s, a, ct, cs', indic')||updatebk2(lb, indic', bid')`

6. `updatebk2(emptylist, indic, bid)` = `emptylist`

5.2 The Behavioural Specification (BS)

An ExtDFD is interpreted as a system of asynchronously interacting actions, where actions are themselves systems of synchronously interacting ExtDFD processes. The BS characterizing the behaviour of an ExtDFD, is derived from the specifications of the behaviour of its actions, data stores and asynchronous data flows, and the specifications of its state dependent behaviour.

Process behaviour is characterized in terms of *labeled state transition systems*. The particular technique, algebraic specification of labeled transition systems, was chosen for the following reasons:

- the abstract nature of the derived specifications means that they are more likely not to specify detail which may unduly constrain subsequent development;
- it provides a framework for integrating specifications of the data, functional, and control aspects of an application; and
- its formal foundation can be used to support rigorous validation and verification activities.

Labeled transition systems have been used to specify complex interactions at various specification levels. At the program specification level the work of Milner and Hoare [Mil80, Hoa85] are outstanding examples, while the work of Lamport [Lam86, Lam88] is an example of the use of such systems at both the design and program specification levels. An algebraic characterization of labeled transition systems defining program behaviours was introduced by Broy and Wirsing [BW83], and further developed by Astesiano et al. in the SMoLCS approach [AR87, AGR88]. The characterizations are based on an operational interpretation of processes as labeled transition systems, and of systems of processes as the composition of their sub systems [AGR87]. The algebraic technique used here is based, in principle, on the work of Astesiano et al. [AGR88], but applied to a higher specification level.

5.2.1 Algebraic state transition systems (ASTSs)

A labeled state transition system is a triple $\langle S, L, T \rangle$ where S is a set of states, L is a set of event labels, and T is the labeled transition relation $[S, L, S]$, whose elements are called *transitions*. A transition $[s_1, l, s_2]$, where s_1 and s_2 are in S and l is in L , intuitively means that the effect of an event, represented by l , on the state s_1 is a change to the state s_2 .

In the SL, state transition systems are characterized algebraically by RSs of the form shown in Figure 5.3. The primitive RS STATE characterizes a set of states, LABEL characterizes a set of labels, and AUXS characterizes the additional functions and/or relations needed to characterize the transition relation, denoted by the symbol $_=_=>_$ and characterized by the set of equations TRANSEQS. Such algebraic specifications of state transition systems are called *algebraic state transition systems* (ASTSs).

TS = STATE + LABEL + AUXS +
 transition relation
 $_ = _ \Rightarrow _$: state, label, state
 laws TRANSEQS

Figure 5.3 Algebraic specification of a state transition system

The BS of an ExtDFD is an ASTS with the primitive RSs, STATE characterizing the states of the ExtDFD, and LABEL characterizing the event labels of the ExtDFD. The BS is compositional in the sense that it is built up from ASTSs characterizing the behaviour of asynchronous data flows, data stores, and actions. The ASTS for action are in turn built up from ASTSs characterizing their constituent ExtDFD processes.

5.2.2 Specifying the behaviour of ExtDFD processes

An ExtDFD process is characterized by its class of behaviours called *invocations*, where an invocation is a labeled sequence of process states. The class of invocations associated with a DFD process is characterized implicitly by a labeled state transition system, $\langle S, L, T \rangle$, where S is a set of ExtDFD process states, L is a set of ExtDFD process labels, and T is the transition relation defining the allowable state transition. For an ExtDFD process with n data inputs and m control and data outputs, a state in S is a $(n+m)$ -tuple, where each place of the tuple reflects the effect of event occurrences, represented by the labels in L , related to the receipt of data on the corresponding input or the generation of data or signals on the corresponding output.

The following types of event labels may be associated with an ExtDFD process:

- *Receive* : Labels representing the observable effects of events which take data off the inputs of the process which are not emanating from data stores.
- *Read/Delete* : Labels representing the observable effects of successful read or delete events on data stores.
- *Erread/Errdel* : Labels representing the observable effects of unsuccessful read or delete events on data stores.
- *Send*: Labels representing the observable effects of send events on data flows.

The behaviour of ExtDFD processes, as defined here, is determined by the events related to the receipt of data on their inputs and the generation of data and/or signals on their outputs. ASTSs specifying processes in the above manner can be said to characterize the *externally observable* behaviour of the processes. Details concerning the internal structure of processes, in the form of state changes occurring as the result of inputs being transformed into outputs, are not characterized by the ASTSs used here. The high level nature of these specifications

is appropriate at the requirements specification/initial design stages, for which the SL is intended, since they do not overly constrain subsequent development with internal details of a process's activities.

The ASTS for the process `CheckBook` is given in Example 5.3.

Example 5.3

Characterizing the state transitions of `CheckBook`

The DFD process `CheckBook` is associated with an abbreviated name `P5`, which is used in the names of the RSs characterizing its states, labels, and transition system.

`P5labels` = `Vetted_book` + `Out_book` + `Book_id` +

Signature

`sort p5label`

constructors

`Receivep5` : `book_id` → `p5label`

`Readp5` : `book_id`, `out_book` → `p5label`

`Erreadp5` : `book_id` → `p5label`

`Sendp5` : `vetted_book` → `p5label`

`P5labels` gives the labels associated with `CheckBook`.

`P5state` = `Vetted_book` + `Book_id` + `Out_book` +

Signature

`sorts` `receivep5`, `readp5`, `sendp5`, `p5state`

constructors

`Nullinp5` : → `receivep5`

`inp5` : `book_id` → `receivep5`

`Nullrdp5` : → `readp5`

`errp5` : → `readp5`

`rdp5` : `out_book` → `readp5`

`Nulloutp5` : → `sendp5`

`outp5` : `vetted_book` → `sendp5`

`<_ _ _>` : `receivep5`, `readp5`, `sendp5` → `p5state`

The state of `CheckBook` is a tuple of sorts `receivep5`, `readp5`, `sendp5` which represent the states of the accesses associated with the data flows `out_book_id`, `out_book`, and `vetted_book` respectively. The states `Nullinp5`, `Nullrdp5`, and `Nulloutp5` represent the situation where no accesses via the associated data flows have been attempted. `errp5` denotes an unsuccessful read attempt to the data store `BOOK`.

`CheckBook_TS` = `P5state` + `P5label` +

Signature

`transition relation`

`_ _ _>_ :` `p5state`, `p5label`, `p5state`

Laws \forall `outbk:out_book`; `ty:copy_type`; `bid:book_id`

1. `<Nullinp5, Nullrdp5, Nulloutp5>`

-`Receivep5(bid)`->

`<inp5(bid), Nullrdp5, Nulloutp5>`

2. `<inp5(bid), Nullrdp5, Nulloutp5>`

-`Readp5(bid, outbk)`->

`<inp5(bid), rdp5(outbk), Nulloutp5>`

Example 5.3 continued

Example 5.3 (continued)

Characterizing the state transitions of CheckBook

3. $ty \neq Ref \Rightarrow$
 $\langle inp5(bid), rdp5(mkoutbk(Available, ty)), Nulloutp5 \rangle$
 $-Sendp5(mkvbk(bid, ty)) \rightarrow$
 $\langle inp5(bid), rdp5(mkoutbk(Available, ty)), outp5(mkvbk(bid, ty)) \rangle$
4. $\langle inp5(bid), rdp5(mkoutbk(Available, Ref)), Nulloutp5 \rangle$
 $-Sendp5(Notborr) \rightarrow$
 $\langle inp5(bid), rdp5(mkoutbk(Available, ty)), outp5(Notborr) \rangle$
5. $inp5(bid), rdp5(mkoutbk(mkbind(borrid)), Nulloutp5 \rangle$
 $-Sendp5(Checkedout) \rightarrow$
 $\langle inp5(bid), rdp5(mkoutbk(mkbind(borrid)), outp5(Checkedout) \rangle$
6. $\langle inp5(bid), Nullrdp5, Nulloutp5 \rangle$
 $-Erreadp5(bid) \rightarrow$
 $\langle inp5(bid), errp5, Nulloutp5 \rangle$
7. $\langle inp5(bid), errp5, Nulloutp5 \rangle$
 $-Sendp5(Bknotinfile) \rightarrow$
 $\langle inp5(bid), errp5, outp5(Bknotinfile) \rangle$

Law 1 defines the transition caused by an access event on `out_book_id`. Law 2 defines the transition resulting from a successful read access to the data store `BOOK`, while law 6 defines the transition resulting from an unsuccessful read access to the same data store. Laws 3, 4, 5, and 7 define the transition resulting from the occurrence of the send event, under different conditions on the input data. Law 6 defines the transition resulting from the occurrence of the send error event which occurs after an unsuccessful read has been made.

5.2.3 Specifying ExtDFD actions

The actions of an ExtDFD are associated with states and event labels derived from the states and labels of their constituent ExtDFD processes. For an action consisting of n ExtDFD processes, P_1, \dots, P_n , a state of the action is of the form $\langle p_1, \dots, p_n \rangle$, where p_i is a state of P_i , $1 \leq i \leq n$.

The event labels of an action represent:

- synchronized send/receive events for for each data flow between ExtDFD processes, where such events are called *internal action events*,
- synchronized receive events associated with the inputs of its initiators,
- read and/or delete events associated with its ExtDFD processes,
- send events associated with its terminators,
- parallel events composed of internal action events,
- a termination event whose effect is to revert all the DFD processes to their idle state,

All data flows between DFD processes in an action are synchronous thus the send and receive events of DFD processes connected by data flows in an action are synchronized. Also the invocation events of the invokers of an action are all synchronized. Labels representing the effect of *synchronized* sets of events of an action are called *synchronous labels*. Such labels are generated by the function

SYNCH which takes a set of labels and returns a synchronous label representing the effect of the synchronized set of events. In an ASTS characterizing the behaviour of an action with n ExtDFD processes, the laws characterizing the effect of synchronized events on the state of an action, are of the following form:

$$p_1 \xrightarrow{l_1} p_1', \dots, p_j \xrightarrow{l_j} p_j', \text{cond}(l_1, \dots, l_j) \Rightarrow \\ \langle p_1, \dots, p_j, p_k, \dots, p_n \rangle \xrightarrow{\text{SYNCH}(\{l_1, \dots, l_j\})} \langle p_1', \dots, p_j', p_k, \dots, p_n \rangle$$

The above is interpreted as follows: if an ExtDFD process's state p_i is capable of being transformed into p_i' by an event labeled by l_i , $1 \leq i \leq j$, and the condition on the labels $\text{cond}(l_1, \dots, l_j)$, holds, then the state of the action $\langle p_1, \dots, p_j, p_k, \dots, p_n \rangle$ can be transformed by the synchronized events, represented by the synchronous event label $\text{SYNCH}(\{l_1, \dots, l_j\})$, to the state $\langle p_1', \dots, p_j', p_k, \dots, p_n \rangle$.

Certain DFD process events also become action events, called *single events* of the action, for example the read and send events of ExtDFD processes in an action. The effect of these events on the state of an action are expressed in their ASTS in the following manner:

$$p_k \xrightarrow{l_k} p_k', \text{cond}(l_k) \Rightarrow \langle p_1, \dots, p_k, \dots, p_n \rangle \xrightarrow{P_k(l_k)} \langle p_1, \dots, p_k', \dots, p_n \rangle$$

P_k is a coercion function, which converts an ExtDFD process label to an action label. In the ASTSs that follow, such coercion is left implicit so as to simplify the presentation of the ASTSs.

Internal action events which affect mutually exclusive parts of an action's state can occur in parallel. This is expressed by the laws of the form:

$$\langle p_1, \dots, p_i, p_j, \dots, p_n \rangle \xrightarrow{l_1} \langle p_1', \dots, p_i', p_j, \dots, p_n \rangle, \\ \langle p_1, \dots, p_i, p_j, \dots, p_n \rangle \xrightarrow{l_2} \langle p_1, \dots, p_i, p_j', \dots, p_n \rangle \Rightarrow \\ \langle p_1, \dots, p_i, p_j, \dots, p_n \rangle \xrightarrow{\text{PAR}(l_1, l_2)} \langle p_1', \dots, p_i', p_j', \dots, p_n \rangle$$

Internal action events which affect the same process state, but mutually exclusive parts of such states, can also occur in parallel, for example, some ExtDFD processes may be allowed to generate some of their outputs in parallel.

Actions are also associated with *termination* events which cause all its DFD processes to revert to the idle state. The resulting state is also called the *idle state* of the action. An action can only be invoked if it is in the idle state, thus parallel invocations of an action are not allowed.

Example 5.4 gives the ASTS characterizing the behaviour of the action CheckoutBook.

Example 5.4

Characterizing the behaviour of the action CheckoutBook

The DFD processes of the action CheckoutBook (A4) are associated with the following abbreviated names: CheckBook - P5, GetOverdueBooks - P6, CalculateFine - P7, VettBorrower - P8, CheckoutUpdate - P9.

The following auxiliary functions are needed in order to characterize the transition relation for CheckoutBook:

FinesRec \equiv Out_borr + Borrower_book_detail + List(Number) + Time +
Signature

auxiliary function

getfinesrec : list(borrower_book_detail), time \rightarrow list(number)

Laws \forall bid:book_id; lb:borrower_book_detail; bbs:bb_status;
t1,t2:time

1. $t2 > t1 \Rightarrow$ getfinesrec (mkbdet(bid, t1, bbs)|lb, t2) =
(Rate*(t2-t1))|getfinesrec (lb, t2)
2. $t2 < t1 \Rightarrow$ getfinesrec (mkbdet(bid, t1, bbs)|lb, t2) = getfinesrec (lb, t2)

FinesRec characterizes the functional relationship between the objects of the list(borrower_book_detail) sub class of out_borr and the data objects associated with the data flow fines_record. The class time is simply treated as an integer line, with successive integers representing successive days.

SumList \equiv List(Number) +
Signature

auxiliary function

sum : list(number) \rightarrow number

Laws \forall n:number; in:list(number)

1. sum(n|in) = n+sum(in)

SumList defines the functional relationship between the input fines_record and the output fine of the process CalculateFine.

A4_TS \equiv A4state + A4label +
Signature

transition relation

$_ == _ ==> _ : a4state, a4label, a4state$

Laws \forall bid:book_id; borrid:borrower_id; t:time; vbk:vetted_book;
p1,p1':statep1;...; p5,p5':statep5; A1,A2:a4label;
vbr:vetted_borrower; ln:list(number); f:number; bflag:borr_flag;
obk:out_book; obr:out_borrower; upbk:out_updated_book;
upbr:out_updated_borr; mess:checkout_message

Synchronized Events: process/process communication (via synchronized data flows)

1. p5--Receivep5(bid)-->p5', p6--Receive1p6(borrid)-->p6' \Rightarrow
<p5, p6, p7, p8, p9>
==SYNCH({Receivep5(bid), Receive1p2(borrid)})==>
<p5', p6', p7, p8, p9>

Example 5.4 continued

Example 5.4 (continued)

Characterizing the behaviour of the action CheckoutBook

2. $p5 \rightarrow \text{Sendp5}(\text{vbk}) \rightarrow p5', p9 \rightarrow \text{Receive1p9}(\text{vbk}) \rightarrow p9' \Rightarrow$
 $\langle p5, p6, p7, p8, p9 \rangle$
 $\Rightarrow \text{SYNCH}(\{\text{Sendp5}(\text{vbk}), \text{Receive1p9}(\text{vbk})\}) \Rightarrow$
 $\langle p5', p6, p7, p8, p9' \rangle$
 3. $p6 \rightarrow \text{Send1p6}(\text{ln}) \rightarrow p6', p7 \rightarrow \text{Receivep7}(\text{ln}) \rightarrow p7' \Rightarrow$
 $\langle p5, p6, p7, p8, p9 \rangle$
 $\Rightarrow \text{SYNCH}(\{\text{Send1p6}(\text{ln}), \text{Receivep7}(\text{ln})\}) \Rightarrow$
 $\langle p5, p6', p7', p8, p9 \rangle$
 4. $p6 \rightarrow \text{Send2p6}(\text{bflag}) \rightarrow p6', p8 \rightarrow \text{Receive2p8}(\text{bflag}) \rightarrow p8' \Rightarrow$
 $\langle p5, p6, p7, p8, p9 \rangle$
 $\Rightarrow \text{SYNCH}(\{\text{Send2p6}(\text{bflag}), \text{Receive2p8}(\text{bflag})\}) \Rightarrow$
 $\langle p5, p6', p7, p8', p9 \rangle$
 5. $p7 \rightarrow \text{Sendp7}(\text{f}) \rightarrow p7', p8 \rightarrow \text{Receive1p8}(\text{f}) \rightarrow p8' \Rightarrow$
 $\langle p5, p6, p7, p8, p9 \rangle$
 $\Rightarrow \text{SYNCH}(\{\text{Sendp7}(\text{f}), \text{Receive1p8}(\text{f})\}) \Rightarrow$
 $\langle p5, p6, p7', p8', p9 \rangle$
 6. $p8 \rightarrow \text{Sendp8}(\text{vbr}) \rightarrow p8', p9 \rightarrow \text{Receive2p9}(\text{vbr}) \rightarrow p9' \Rightarrow$
 $\langle p5, p6, p7, p8, p9 \rangle$
 $\Rightarrow \text{SYNCH}(\{\text{Sendp8}(\text{vb}), \text{Receive2p8}(\text{vbr})\}) \Rightarrow$
 $\langle p5, p6, p7, p8', p9' \rangle$
- Single Events: input and output (including read/write) events of the action
7. $p5 \rightarrow \text{Readp1}(\text{bid}, \text{obk}) \rightarrow p5' \Rightarrow$
 $\langle p5, p6, p7, p8, p9 \rangle \Rightarrow \text{Readp1}(\text{bid}, \text{obk}) \Rightarrow \langle p5', p6, p7, p8, p9 \rangle$
 8. $p5 \rightarrow \text{Erreadp1}(\text{bid}) \rightarrow p5' \Rightarrow$
 $\langle p5, p6, p7, p8, p9 \rangle \Rightarrow \text{Erreadp1}(\text{bid}) \Rightarrow \langle p5', p6, p7, p8, p9 \rangle$
 9. $p6 \rightarrow \text{Receive2p2}(\text{t}) \rightarrow p6' \Rightarrow$
 $\langle p5, p6, p7, p8, p9 \rangle \Rightarrow \text{Receive2p2}(\text{t}) \Rightarrow \langle p5, p6', p7, p8, p9 \rangle$
 10. $p6 \rightarrow \text{Readp2}(\text{borrid}, \text{obr}) \rightarrow p6' \Rightarrow$
 $\langle p5, p6, p7, p8, p9 \rangle \Rightarrow \text{Readp2}(\text{borrid}, \text{obr}) \Rightarrow \langle p5, p6', p7, p8, p9 \rangle$
 11. $p6 \rightarrow \text{Erreadp2}(\text{borrid}) \rightarrow p6' \Rightarrow$
 $\langle p5, p6, p7, p8, p9 \rangle \Rightarrow \text{Erreadp2}(\text{borrid}) \Rightarrow \langle p5, p6', p7, p8, p9 \rangle$
 12. $p9 \rightarrow \text{Send1}(\text{bid}, \text{upbk}) \rightarrow p9' \Rightarrow$
 $\langle p5, p6, p7, p8, p9 \rangle \Rightarrow \text{Update1}(\text{bid}, \text{upbk}) \Rightarrow \langle p5, p6, p7, p8, p9' \rangle$
 13. $p9 \rightarrow \text{Send2}(\text{borrid}, \text{upbr}) \rightarrow p9' \Rightarrow$
 $\langle p5, p6, p7, p8, p9 \rangle \Rightarrow \text{Update2}(\text{borrid}, \text{upbr}) \Rightarrow \langle p5, p6, p7, p8, p9' \rangle$
 14. $p9 \rightarrow \text{Send3}(\text{mess}) \rightarrow p9' \Rightarrow$
 $\langle p5, p6, p7, p8, p9 \rangle \Rightarrow \text{Send}(\text{mess}) \Rightarrow \langle p5, p6, p7, p8, p9' \rangle$
 15. $p9 \rightarrow \text{Receive3p5}(\text{t}) \rightarrow p9' \Rightarrow$
 $\langle p5, p6, p7, p8, p9 \rangle \Rightarrow \text{Receive3p5}(\text{t}) \Rightarrow \langle p5, p6, p7, p8, p9' \rangle$
- Parallel Events
- Events which affect separate parts of an action can be carried out in parallel ---
16. $\langle p5, p6, p7, p8, p9 \rangle \Rightarrow A1 \Rightarrow \langle p5', p6, p7, p8, p9 \rangle,$
 $\langle p5, p6, p7, p8, p9 \rangle \Rightarrow A2 \Rightarrow \langle p5, p6', p7', p8', p9' \rangle \Rightarrow$
 $\langle \langle p5, p6, p7, p8, p9 \rangle \Rightarrow \text{PAR}(A1, A2) \Rightarrow \langle p5', p6', p7', p8', p9' \rangle$
 17. $\langle p5, p6, p7, p8, p9 \rangle \Rightarrow A1 \Rightarrow \langle p5', p6', p7, p8, p9 \rangle,$
 $\langle p5, p6, p7, p8, p9 \rangle \Rightarrow A2 \Rightarrow \langle p5, p6, p7', p8', p9' \rangle \Rightarrow$
 $\langle p5, p6, p7, p8, p9 \rangle \Rightarrow \text{PAR}(A1, A2) \Rightarrow \langle p5', p6', p7', p8', p9' \rangle$
 18. $\langle p5, p6, p7, p8, p9 \rangle \Rightarrow A1 \Rightarrow \langle p5', p6', p7', p8, p9 \rangle,$
 $\langle p5, p6, p7, p8, p9 \rangle \Rightarrow A2 \Rightarrow \langle p5, p6, p7, p8', p9' \rangle \Rightarrow$
 $\langle p5, p6, p7, p8, p9 \rangle \Rightarrow \text{PAR}(A1, A2) \Rightarrow \langle p5', p6', p7', p8', p9' \rangle$
 19. $\langle p5, p6, p7, p8, p9 \rangle \Rightarrow A1 \Rightarrow \langle p5', p6', p7', p8', p9 \rangle,$
 $\langle p5, p6, p7, p8, p9 \rangle \Rightarrow A2 \Rightarrow \langle p5, p6, p7, p8, p9' \rangle \Rightarrow$
 $\langle p5, p6, p7, p8, p9 \rangle \Rightarrow \text{PAR}(A1, A2) \Rightarrow \langle p5', p6', p7', p8', p9' \rangle$

Example 5.4 continued

Example 5.4 (continued)

Characterizing the behaviour of the action CheckoutBook

Termination event--- Nullp_i, 5 ≤ i ≤ 9, is the abbreviated form for the idle state of P_i ---

20. <p5, p6, p7, p8, <in1p9(vbk), in2p9(vbr), timep9(t), out1p9(omess), out2p9(ubr), out3p9(ubk)>>
 ==Terminatea4==>
 <Nullp5, Nullp6, Nullp7, Nullp8, Nullp9>
21. <p5, p6, p7, p8, <in1p9(Notborr), in2, NullTimep9, out1p9(omess), Nullout2p9, Nullout3p9>>
 ==Terminatea4==>
 <Nullp5, Nullp6, Nullp7, Nullp8, Nullp9>
22. <p5, p6, p7, p8, <in1p9(CheckedOut), in2, NullTimep9, out1p9(omess), Nullout2p9, Nullout3p9>>
 ==Terminatea4==>
 <Nullp5, Nullp6, Nullp7, Nullp8, Nullp9>
23. <p5, p6, p7, p8, <in1p9(Bknotinfile), in2, NullTimep9, out1p9(omess), Nullout2p9, Nullout3p9>>
 ==Terminatea4==>
 <Nullp5, Nullp6, Nullp7, Nullp8, Nullp9>
24. <p5, p6, p7, p8, <in1, in2p9(NoBorr), NullTimep9, out1p9(omess), Nullout2p9, Nullout3p9>>
 ==Terminatea4==>
 <Nullp5, Nullp6, Nullp7, Nullp8, Nullp9>
25. <p5, p6, p7, p8, <in1, in2p9(mkerrvborr(f)), NullTimep9, out1p9(omess), Nullout2p9, Nullout3p9>>
 ==Terminatea4==>
 <Nullp5, Nullp6, Nullp7, Nullp8, Nullp9>
26. <p5, p6, p7, p8, <in1p9(mkvbk(bid, Per)), in2p9(mkvvetborr(mkoutb(lb, Undergrad, n), borrid)), NullTimep9, out1p9(omess), Nullout2p9, Nullout3p9>>
 ==Terminatea4==>
 <Nullp5, Nullp6, Nullp7, Nullp8, Nullp9>

5.2.4 Characterizing the behaviour of data flows and data stores

The dynamic aspects of synchronous data flows are implicit in the mechanisms used in specifying synchronous interactions amongst processes. Behaviorally, data stores and asynchronous data flows are treated as processes, and are associated with states, events, and transition relations defining state transitions. In what follows, data stores and asynchronous data flows are collectively called *store processes*.

The state of a store process is the state of its structure, for example, a state of an asynchronous data flow is a state of its queue structure. The event labels correspond to the access functions specified in the RSs characterizing the structures of the store processes, in the following manner:

For a data store with structure struct :-

- A read access function $read : struct, key \rightarrow readval$ is associated with the class of labels $READ(id, val)$, where $id:key, val:readval$.
- An update function $write : struct, key, data \rightarrow struct$ is associated with the class of labels $WRITE(id, d)$, where $id:key, d:data$.
- An add function $add : struct, data \rightarrow struct$ is associated with the class of labels $ADD(d)$, where $d:data$.
- A delete function $delete : struct, key \rightarrow struct$ is associated with the class of labels $DELETE(id)$, where $id:key$.

For an asynchronous data flow with structure asynch:-

- The function $addq : elem, asynch \rightarrow asynch$ is associated with the class of labels $ADD(e)$, where $e:elem$.
- The function $deleteq : asynch \rightarrow asynch$ is associated with the class of labels $DEL(top(q))$, where $q:asynch$ is the state of the data flow before the deletion is effected.

The ASTS for an asynchronous data flow is given in Figure 5.4. The primitive RS $AsynchLabels(Element)$ characterizes the labels, of sort $aslabel$, associated with the data flow.

$BehAsynch \equiv Asynch(Element) + AsynchLabel(Element) +$

Signature

transition relation

$_ == _ ==> _ : asynch, aslabel, asynch$

Laws $\forall e, e':elem; q:asynch$

1. $emptyq == ADD(e) ==> addq(e, emptyq)$
2. $addq(e, q) == ADD(e') ==> addq(e', addq(e, q))$
3. $q \neq emptyq \Rightarrow q == DEL(top(q)) ==> deleteq(q)$

Figure 5.4 The ASTS characterizing the behaviour of an asynchronous data flow

Example 5.5 gives the ASTS for the data store BOOK whose static aspects are defined in Example 5.3.

Example 5.5

Characterizing the behaviour of the data store BOOK

$Book_TS \equiv BookStore + BookLabels +$

Signature

transition relation

$_ == _ ==> _ : list(book), booklabel, list(book)$

Laws $\forall bid:book_id; lb:list(book); bind:borrower_indicator$

1. $lb == READBOOK2(bid, readbook2(lb, bid)) ==> lb$
2. $lb == UPDATEBK2(bid, bind) ==> updatebk2(lb, bind)$

5.2.5 Deriving the BS

ExtDFDs, like actions, are associated with state transition systems which characterize their behaviour. The BS is the ASTS characterizing the transition system of an ExtDFD, and is derived from the ASTSs characterizing the behaviour of the ExtDFD's actions, data stores, and asynchronous data flows.

The states of an ExtDFD with actions, A_1, \dots, A_n , data stores, DS_1, \dots, DS_p , and asynchronous data flows, AS_1, \dots, AS_q , are of the form $\langle sp, ds_1, \dots, dsp, as_1, \dots, as_q, mode \rangle$, where $sp = \{a_1, \dots, a_n\}$ is a set of action states, called the *action state set* of the ExtDFD, ds_i is a state of DS_i , for $1 \leq i \leq p$, and as_i is a state of AS_i , for $1 \leq i \leq q$, and $mode$ is a mode of operation of the ExtDFD.

Actions which are disabled cannot be affected by events during the period they are disabled, thus disabled actions need not be represented in the state of an ExtDFD during the periods they are disabled. This means that one need only represent *enabled* states (any state other than a disabled state) in the action state set of an ExtDFD.

The event labels of ExtDFDs represent the effects of the following classes of events:

- synchronized communication between actions and data stores, and between actions and the receive (ADD) and send (DEL) access mechanisms of asynchronous data flows,
- events depicted by signals (control flows generated by external entities and actions),
- parallel events composed of the above events.

Synchronized events are represented by synchronous event labels, and are characterized in the same manner as synchronous labels in actions. For example, the effect of a synchronous communication event between an action and a data store is characterized in the BS by a law of the form below (in the RSs that follow $\text{insert}(a, sp)$ is abbreviated to $\{a, sp\}$):

$$\begin{aligned}
 a == \text{Read}(bid, val) ==> a', ds_i == \text{READ}(bid, val) ==> ds_i', \text{cond}(l, s) \Rightarrow \\
 & \langle \{a, sp\}, ds_1, \dots, ds_i, \dots, dsp, as_1, \dots, as_q, mode \rangle \\
 & == \text{SYNCH}(\{\text{Read}(bid, val), \text{READ}(bid, val)\}) ==> \\
 & \langle \{a', sp\}, ds_1, \dots, ds_i', \dots, dsp, as_1, \dots, as_q, mode \rangle
 \end{aligned}$$

The following sections detail the interactions that can be specified in the BS.

Monitored access to data stores

The ExtDFD events which access data stores may need to be monitored, for example, an action may read an object from a data store, modify the object, and subsequently update the data store with the modified object. Such accesses are called read/update accesses. In such cases, another action which reads and updates the same object in parallel may cause the data store to move into an inconsistent state. To avoid such situations, read/update accesses to data stores need to be monitored. Static analysis of an ExtDFD, and examination of its data type definitions, can determine which pair of output and input data flows of a data store represent read/update accesses. Data stores in an ExtDFD associated with such pairs are said to be monitored.

A solution to the problem described above would be for the data store to prohibit access to objects which are being updated. The approach used here associates with monitored data stores a list of identifiers (or keys) which identify the objects in the data store on which monitored accesses are prohibited. An action wishing to access a monitored data store for a read/update, or deletion (the monitored accesses), can only do so if the object's identifier is not in the list. There is no need to check such lists if an action simply reads objects without subsequently updating the monitored data stores, nor if the action simply adds new objects to the data store. A synchronized update (of a read/update access) between a monitored data store and an action results in the updated object's identifier being removed from the list.

In the case where an action with a read/update pair, reads in data from the data store but does not subsequently update the data store, then the termination event of the action is synchronized with the event that removes the identifier of the object read in from the list. This is to avoid an object being permanently prohibited from being accessed.

The laws characterizing monitored interactions with data stores, under the simple scheme described above, are of the following form:

The read part

$$\begin{aligned}
 a == \text{Read}(id, val) ==> a', ds == \text{READ}(id, val) ==> ds', \text{contains}(id, blist) = \text{false} \Rightarrow \\
 & \langle \{a\}sp, ds1, \dots, \langle ds, blist \rangle, \dots, dsp, as1, \dots, asq, mode \rangle \\
 & == \text{SYNCH}(\{\text{Read}(id, val), \text{READ}(id, val)\}) ==> \\
 & \langle \{a\}sp, ds1, \dots, \langle ds, \text{add}(id, blist) \rangle, \dots, dsp, as1, \dots, asq, mode \rangle
 \end{aligned}$$

where `contains` is a function which takes an object, and a list of objects and returns `true` if the object is in the list, otherwise it returns the value `false`, and the function `add` inserts an object into a given list.

The update part

$$\begin{aligned}
 &a == \text{Send}(id, val) ==> a', ds == \text{UPDATE}(id, val) ==> ds' \Rightarrow \\
 &\quad \langle \{a\}sp, ds1, \dots, \langle ds, blist \rangle, \dots, dsp, as1, \dots, asq, mode \rangle \\
 &\quad == \text{SYNCH}(\{\text{Send}(id, val), \text{UPDATE}(id, val)\}) ==> \\
 &\quad \langle \{a\}jsp, ds1, \dots, \langle ds', \text{delete}(id, blist) \rangle, \dots, dsp, as1, \dots, asq, mode \rangle
 \end{aligned}$$

where `delete` is a function which deletes a given object from a list.

Laws are also needed to check whether an action associated with a read/update pair has terminated without updating a data store. These laws are of the form:

$$\begin{aligned}
 &\text{terminated}(a), \text{Noupdate}(a) \Rightarrow \cdot \\
 &\quad \langle \{a\}sp, ds1, \dots, \langle ds, blist \rangle, \dots, sdsp, as1, \dots, asq, mode \rangle \\
 &\quad == \text{Terminate}a ==> \\
 &\quad \langle \{a\}jsp, ds1, \dots, \langle ds, \text{delete}(id, blist) \rangle, \dots, dsp, as1, \dots, asq, mode \rangle
 \end{aligned}$$

where `terminated` is a function which return true if a process can revert to an idle state (i.e. it has finished transforming its inputs to outputs), and `Noupdate` checks whether, for a particular pair or read/updates, a read and subsequent update has been carried out, returning true if a read has been made on an object identified by `bid`, but no subsequent update has been carried out, and false otherwise.

Specifying state dependent behaviour in the BS

Signals in an ExtDFD depict event classes whose instances (event occurrences) can affect the current mode of operation of the ExtDFD. As described in Chapter 2, changes in the mode of operation of an ExtDFD can affect the behaviour of its actions in three ways: they can be initiated, enabled, or disabled. Thus the occurrence of events associated with signals, as well as changing the mode component of an ExtDFD state, also affects the action state set of the ExtDFD. A signal which causes an action to be disabled causes the action's state to be removed from the action state set, while a signal which causes an action to be enabled causes the action's idle state to be added to the action state set of the ExtDFD. Thus removing and adding action states to the action state set of an ExtDFD corresponds to the disabling and enabling of actions, respectively. Initiation signals are associated with enable and disable components, where the disable component takes effect when the corresponding actions have gone through a single invocation.

The laws in the BS characterizing the effects of events associated with signals are of the following form:

$$\langle sp, st1, \dots, stp, as1, \dots, asq, mode \rangle == \text{Sig} ==> \langle sp', st1, \dots, stp, as1, \dots, asq, mode' \rangle$$

The above is interpreted as follows: the occurrence of the event whose effect is denoted by the label Sig, when the ExtDFD is in the mode mode, causes the mode to change to mode', where such change causes changes in the state of some of the actions in sp, represented by sp'.

Parallel events and an example

The ExtDFD events which can affect disjoint parts of an ExtDFD can occur in parallel. Example 5.6 gives a BS for the diagram in Figure 5.1, viewed as a simple ExtDFD consisting of a single action.

Example 5.6

The BS for the ExtDFD shown in Figure 5.1

Beh \equiv Book_TS + Borrower_TS + Behstate + BehLabel +

Signature

transition relation

$_ \text{====} _ \text{====} _$: beh, lbeh, beh

Laws \forall bid:book_id; lb:list(book); out:out_updated_book;
 bklist:list(book_id); brlist:list(borrower_id); borrid:borrower_id;
 ckinfo:asynch(checkout_info); indic:borrower_indicator; a, a':p5state;
 obk:out_book; obr:out_borr; asynch(checkout_message);
 ck:checkout_info; outmess:checkout_message

Synchronized events: accesses to data stores

1. lb==READBOOK2(bid, mkreadval(obk))==>lb,
 readbook2(lb, bid) = mkreadval(obk),
 a==Readp5(bid, obk)==>a', contains(bid, bklist) = false \Rightarrow
 <a>, <lb,bklist>, <lbr, brlist>, ckinfo, ckmess>
 ===SYNCH({READBOOK2(bid, mkreadval(obk)),
 Readp5(bid, obk)})===>
 <a'>, <lb,add(bid, bklist)>, <lbr, brlist>, ckinfo, ckmess>
2. lbr==READBORR2(borrid, mkrdborr(obr))==>lbr,
 a==Readp6(borrid, obr)==>a',
 contains(borrid, brlist) = false, readborr2(lbr, borrid) = mkrdborr(obr) \Rightarrow
 <a>, <lb,bklist>, <lbr, brlist>, ckinfo, ckmess>
 ===SYNCH({READBORR2(borrid, mkrdborr(obr)),
 Readp6(borrid, obr)})===>
 <a'>, <lb, bklist>, <lbr, add(borrid, brlist)>, ckinfo, ckmess>
3. lb==READBOOK2(bid, Nullval)==>lb, readbook2(lb, bid) = Nullval,
 a==Erreadp5(bid)==>a' \Rightarrow
 <a>, <lb,bklist>, <lbr, brlist>, ckinfo>
 ===SYNCH({READBOOK2(bid, Nullval), Erreadp5(bid)})===>
 <a'>, <lb, bklist>, <lbr, brlist>, ckinfo, ckmess>
4. lbr==READBORR2(borrid, Nullbrval)==>lbr, readborr2(lbr, borrid) = Nullbrval,
 a==Erreadp6(borrid)==>a' \Rightarrow
 <a>, <lb,bklist>, <lbr, brlist>, ckinfo, ckmess>
 ===SYNCH({READBORR2(borrid, Nullbrval), Erreadp6(borrid)})===>
 <a'>, <lb, bklist>, <lbr, brlist>, ckinfo, ckmess>
5. lb==UPDATEBK(bid, indic)==>lb',
 a==Send1p9(bid, indic)==>a' \Rightarrow
 <a>, <lb,bklist>, <lbr, brlist>, ckinfo, ckmess>
 ===SYNCH({UPDATEBK(bid, indic), Send3p9(bid, indic)})===>
 <a'>, <lb', delete(bid, bklist)>, <lbr, brlist>, ckinfo, ckmess>

Example 5.6 continued

Example 5.6

The BS for the ExtDFD shown in Figure 5.1

```

6. lbr==UPDATEBR(borrid, blist)==>lbr',
   a==Send2p9(borrid, blist)==>a' =>
   <{a}, <lb,bklist>, <lbr, brlist>, ckinfo, ckmess>
   ==>SYNCH((UPDATEBR(borrid, blist), Send2p9(borrid, blist)))==>
   <{a'}, <lb, bklist>, <lbr', delete(borrid, brlist)>, ckinfo, ckmess>

```

Synchronized events: interactions with asynchronous data flows

```

7. ckinfo==DEL(mkoutinfo(bid, borrid))=>ckinfo',
   top(ckinfo) = mkoutinfo(bid, borrid),
   a==SYNCH((Receivep5(bid), Receive1p6(borrid)))=>a' =>
   <{a}, <lb,bklist>, <lbr, brlist>, ckinfo, ckmess>
   ==>SYNCH((DEL(mkoutinfo(bid, borrid)),
              SYNCH((Receivep5(bid), Receive1p6(borrid)))))==>
   <{a'}, <lb, bklist>, <lbr, brlist>, ckinfo', ckmess>
8. ckinfo==ADD(ck)==>ckinfo' =>
   <{a}, <lb,bklist>, <lbr, brlist>, ckinfo, ckmess>
   ==>SYNCH((ADD(ck)))==>
   <{a}, <lb,bklist>, <lbr, brlist>, ckinfo', ckmess>
9. ckmess==ADD(outmess)==>ckmess', a==Send3p9(outmess)==>a' =>
   <{a}, <lb,bklist>, <lbr, brlist>, ckinfo, ckmess>
   ==>SYNCH((ADD(outmess), Send3p9(outmess)))==>
   <{a'}, <lb,bklist>, <lbr, brlist>, ckinfo, ckmess'>
10. ckmess==DEL(outmess)==>ckmess' =>
   <{a}, <lb,bklist>, <lbr, brlist>, ckinfo, ckmess>
   ==>SYNCH((DEL(outmess)))==>
   <{a'}, <lb,bklist>, <lbr, brlist>, ckinfo, ckmess'>

```

Parallel events

Events which affect mutually exclusive parts of the state can be carried out in parallel

5.3 The BS as a formal basis for reasoning with ExtDFDs

The BS of an ExtDFD can be used to support the following activities:

- Investigating behavioural properties captured by the BS (validation).
- Proving that a specification implements the BS (verification).

In the following, the use and limitations of the BS for validation and verification are described.

5.3.1 Investigating behavioural properties of ExtDFDs with the BS

The state transition system characterized by the BS can be used to investigate properties, called *safety properties*, which concern what applications are allowed to do, or equivalently, not allowed to do. An example of a required safety property for the library application is that a book cannot be checked out and available at the same time, that is, there should be no transition to a state in which a book is both checked out and available. Safety properties can be investigated

directly with the BS, since its laws explicitly express what the system is allowed to do.

Properties related to what an application must do, called *liveness properties* [Lam86], are sometimes implicit in the BS of ExtDFDs. An example of a liveness property which can be implied from the BS is termination of an action. To facilitate the investigation of liveness properties the labeled sequences of states representing invocations need to be made explicit. To do this requires the addition of functions and relations to those already present in the BS. A useful relation for analysis purposes is the *reachability relation*, $state \xrightarrow{L} state'$: state, list(label), state, which is defined on process states and lists of labels as follows:

$$\begin{aligned} state \xrightarrow{A} state' &\Rightarrow state \xrightarrow{A|\emptyset} state' \\ state \xrightarrow{L} state', state \xrightarrow{A} state' &\Rightarrow state \xrightarrow{A|L} state' \end{aligned}$$

The reachability relation is an extension of the transition relation which represents state transitions resulting from a sequence of event occurrences. Used naively, the reachability relation is not of much use, since there are, in general, an infinite number of states reachable from a given state. State transition trees (STTs), described in Chapter 2, provide finite representations of reachability relations, where the nodes represent classes of states, while their edges represent classes of event labels. Such trees can be used to check whether certain states, or classes of states, are reachable from a given state, or a class of states. In particular such trees can be used to investigate whether inconsistent states are reachable from consistent states. Even so, the STT for a BS can be very large, making the automatic generation of such trees from ASTSs desirable.

An automated system which, given an ASTS, would generate STTs, and provide functions for analyzing such trees is needed in order for the framework to be of practical use in this respect. Such automated system can also be used to exercise (test) the behavioural specification under various control and/or data inputs. This would involve substituting appropriate values to obtain instances of the state and event classes in the STTs.

5.3.2 Proving implementations of the BS

The hiding of states of a process is sometimes desirable when attempting to establish its equivalence to another process under some criteria for equivalence. For example, two processes which behave identically as far as their inputs and outputs are concerned, but which differ in their internal workings, are equivalent under a criterion which makes two processes equivalent if and only if they generate the same results (output data) given the same inputs. Such equivalences are usually

called *observational* [ST87]. An observational equivalence can be used as the basis for determining whether a specification implements the BS.

The notion of a specification implementing the BS developed here depends on the ability to hide some of the states it specifies. *Observation specifications* are used for this purpose. An observation specification is a tuple $\langle B, S \rangle$, where B is an ASTS, and S is an ASTS specifying states, called *observable states*, and their allowable transitions, called *observable transitions*. The states specified in S are derived from the states of B by hiding some components of the states in B .

A particular observation specification, called an *In/Out (I/O) specification*, is used to carry out observations on actions. An I/O specification, $\langle A, S \rangle$ where A is an ASTS characterizing an action with states of the form $\langle \text{pinit}_1, \dots, \text{pinit}_q, \text{p}_1, \dots, \text{p}_n, \text{pter}_1, \dots, \text{pter}_m \rangle$, and S is an ASTS characterizing observations on the invokers, $\text{pinit}_1, \dots, \text{pinit}_q$, and terminators, $\text{pter}_1, \dots, \text{pter}_m$, of the action. The states specified in S , called *I/O states*, are of the form $\langle \text{pinit}_1, \dots, \text{pinit}_q, \text{pter}_1, \dots, \text{pter}_m \rangle$. State transitions in S are characterized by laws of the form:

$$\langle \text{pinit}_1, \dots, \text{pinit}_q, \text{p}_1, \dots, \text{p}_n, \text{pter}_1, \dots, \text{pter}_m \rangle \xrightarrow{A} \langle \text{pinit}'_1, \dots, \text{pinit}'_q, \text{p}'_1, \dots, \text{p}'_n, \text{pter}'_1, \dots, \text{pter}'_m \rangle \Rightarrow \langle \text{pinit}_1, \dots, \text{pinit}_q, \text{pter}_1, \dots, \text{pter}_m \rangle \xrightarrow{A} \langle \text{pinit}'_1, \dots, \text{pinit}'_q, \text{pter}'_1, \dots, \text{pter}'_m \rangle$$

The transition system characterized by S is called the *I/O transition system* of the action characterized by A .

An ASTS, Beh_1 , characterizing the behaviour of an action, is said to *implement an ASTS*, Beh_2 , if there exists a function from the I/O states of Beh_1 to the I/O states of Beh_2 , which preserves the I/O transition system of Beh_1 . Intuitively, the above definition captures the notion of action equivalence as determined by their external behaviours, encapsulated by their I/O specifications. The above notion of a specification implementing another is similar to that of Lamport's, which states that a "specification S_1 implements a specification S_2 if every externally visible behaviour allowed by S_1 is allowed by S_2 " [AL88]. Lamport's behaviours are sequences of states, which are implicit in the ASTSs defined here.

An action implementing another action A , may be a result of further decomposition of A 's processes, or may consist of a totally different process structure. In each case the action must have the same interface inputs and outputs (but which may be further decomposed). The decomposition approach can be used as the basis of a transformation development strategy, where a transformation occurs when an action's process is decomposed. In verifying whether an action,

A2, resulting from the decomposition of a process in another action, A1, implements A1 involves viewing the decomposed process and its decomposed system of processes as actions and proving that the decomposed system implements the process. This is the situation in the latter verification approach mentioned above.

The implementation of a BS by another specification is based on the above notion of an action's specification implementing another action's specification. Formally, a specification, S, implements the BS of an ExtDFD, B, if there exists a function from the actions specified in S to the actions specified in B, such that the action of S which is mapped to the action in B, implements the action in B.

5.4 Conclusion

In this chapter it is shown how a formal specification of behaviour for ExtDFDs, in the form of a BS, can be derived given specifications in the DE characterizing the structure and object classes of data flows and data stores, and specifications of behaviour for the DFD processes within actions. The BS can be viewed as an initial design specification, and alleviates some of the problems associated with generating such specifications from DFDs in the SA approach. Transition to design involves creating the ExtDFD from the DFD and then deriving the BS, which is a characterization of the formal interpretations associated with the ExtDFD. Such a transition is less likely to be as problematic as the transition from SA specifications to SD since it involves extending the DFDs themselves to incorporate control information.

The BS can be used to formally validate behaviour of ExtDFDs, either by associating with ExtDFDs a concrete operational model which is consistent with the state transition system it characterizes, thus making the ExtDFDs executable, or by analyzing the BS itself. Automated tools for analyzing the BS are desirable, given the volume of detail that may be involved in such analyses. The BS also provides a basis for verifying subsequent designs, via the notion of a specification implementing the BS introduced in this chapter.

CHAPTER 6

Two Examples of Deriving Behavioural Specifications from ExtDFDs

6.0 Introduction

The tools and techniques described in Chapter 5 are applied to two different types of applications in this chapter. The first example, adapted from an example given in Hatley and Piribhai [HP87], is concerned with the specification of requirements for a automobile cruise control application. This application is control-intensive in the sense that its behaviour is determined by the current mode of operation in which it is in. A change in the current mode is determined by the occurrence of external stimuli. The structures of data occurring in the application, and the relationships between them, are simple, thus the example serves to focus on the use of the tools and techniques for specifying the control aspects of applications.

The second example is the library application introduced in Chapter 2. This application is data-intensive in the sense that the structures of the data, and the relationships between them play an important role in the specification of its requirements. Furthermore, the application has one mode of operation, thus the control aspects of the application of the application are relatively simple. The example thus serves to focus on the use of the tools and techniques for specifying the processing and data aspects of the application.

In Section 6.1 the tools and techniques are applied the automobile cruise control application and in Section 6.2 they are applied to the library example.

6.1 The Automobile Cruise System

The function of the cruise control application is to maintain an automobile at a constant speed when commanded to do so by the driver. The driver must be able to enter the following commands:

- *cruise on* - activate the cruise control application.
- *cruise off* - deactivate the cruise control application.
- *start accelerating* - causes the automobile to accelerate at a comfortable rate.
- *stop accelerating* - stops the acceleration initiated by a start accelerating command.
- *resume* - causes the application to return the automobile to the speed selected prior to braking or gear shifting.

The cruise control application can only be activated while the engine is running and the automobile is in top gear. When activated the application selects the current speed as the desired speed only if it is at least 30 miles per hour. If the speed is less than 30 miles per hour then the application is automatically deactivated. Deactivation of the application by the driver returns control to the driver regardless of any other commands issued to the application. The start accelerating command causes the application to accelerate the car at a comfortable rate until the stop accelerating command is issued, at which time the application holds the car at the new speed. The driver is permitted to reduce speed by depressing the brake pedal while the application is active. Depressing the brake pedal or shifting out of top gear temporarily disables the application. Issuing the resume command after the brake is released and the automobile is in top gear causes the application to maintain the speed at the speed prior to braking or gear shifting, while issuing the start accelerating command after brake release and a return to top gear causes the application to accelerate the automobile. However, if a deactivate application command is issued in the intervening time then the resume and start accelerating commands do nothing.

The ExtDFD for the cruise control application is shown in Figure 6.1, and the supporting state transition diagram (STD) is shown in Figure 6.2. In the approach used here the driver commands are modeled as toggle signals, for example, the cruise on and off commands are represented as a single signal, called `cruise_on/off`, acting like a toggle switch, as is made clear in the STD for the application. It is also assumed that the shaft is interfacing with a system that can detect and pass on its pulse rate and rate of change to the application. Such a system is assumed to be part of the external entity `shaft`.

There are five actions in the ExtDFD, all consisting of single processes. The specification of behaviour is concerned mainly with the conditions under which these actions are enabled and disabled. In the relational specifications (RSs) characterizing states, labels and transition systems, the processes (actions) are identified by the following short forms: `CalcAcc` is P1, `CalcSpeed` is P2, `SelectDesiredSpeed` is P3, `MaintainSpeed` is P4, and `MaintainAcc` is P5. A RS called `Number`, specifying floating point numbers and arithmetic on such numbers, is assumed to be available. Throughout, the RSs are interspersed with informal textual annotations to enhance their readability.

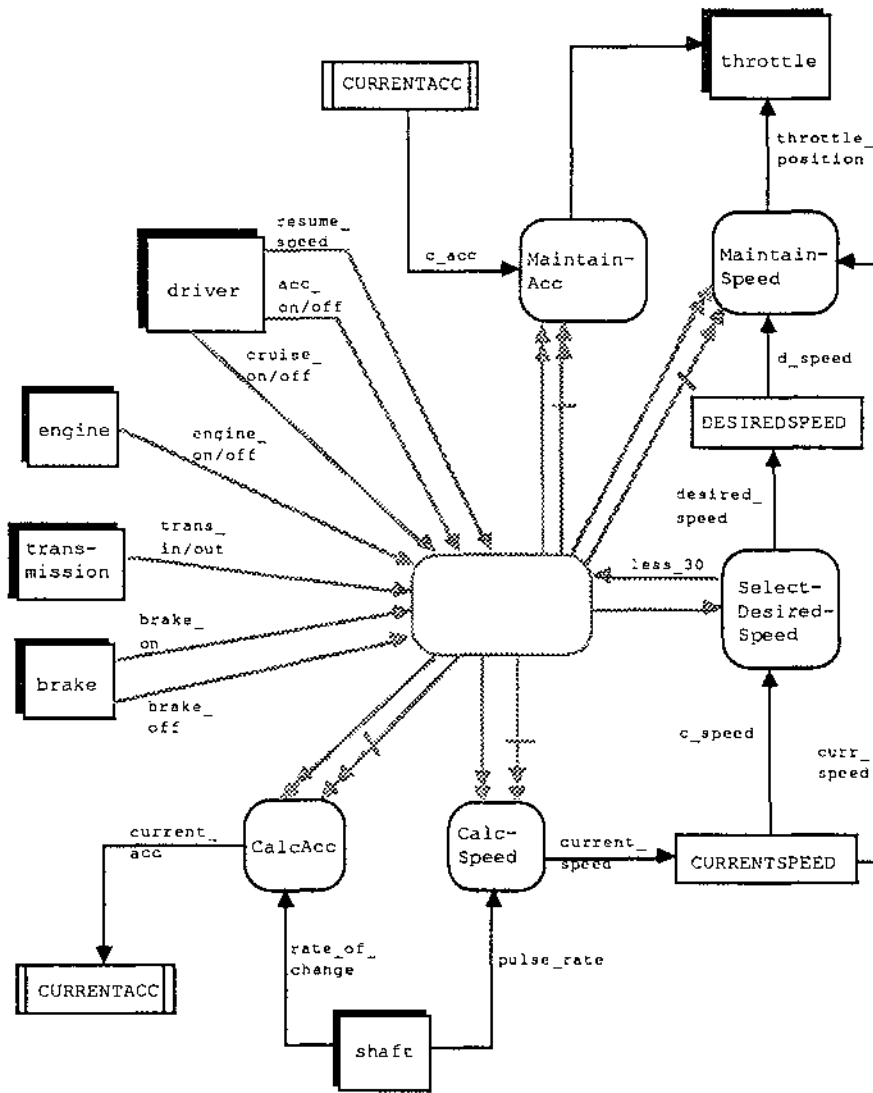
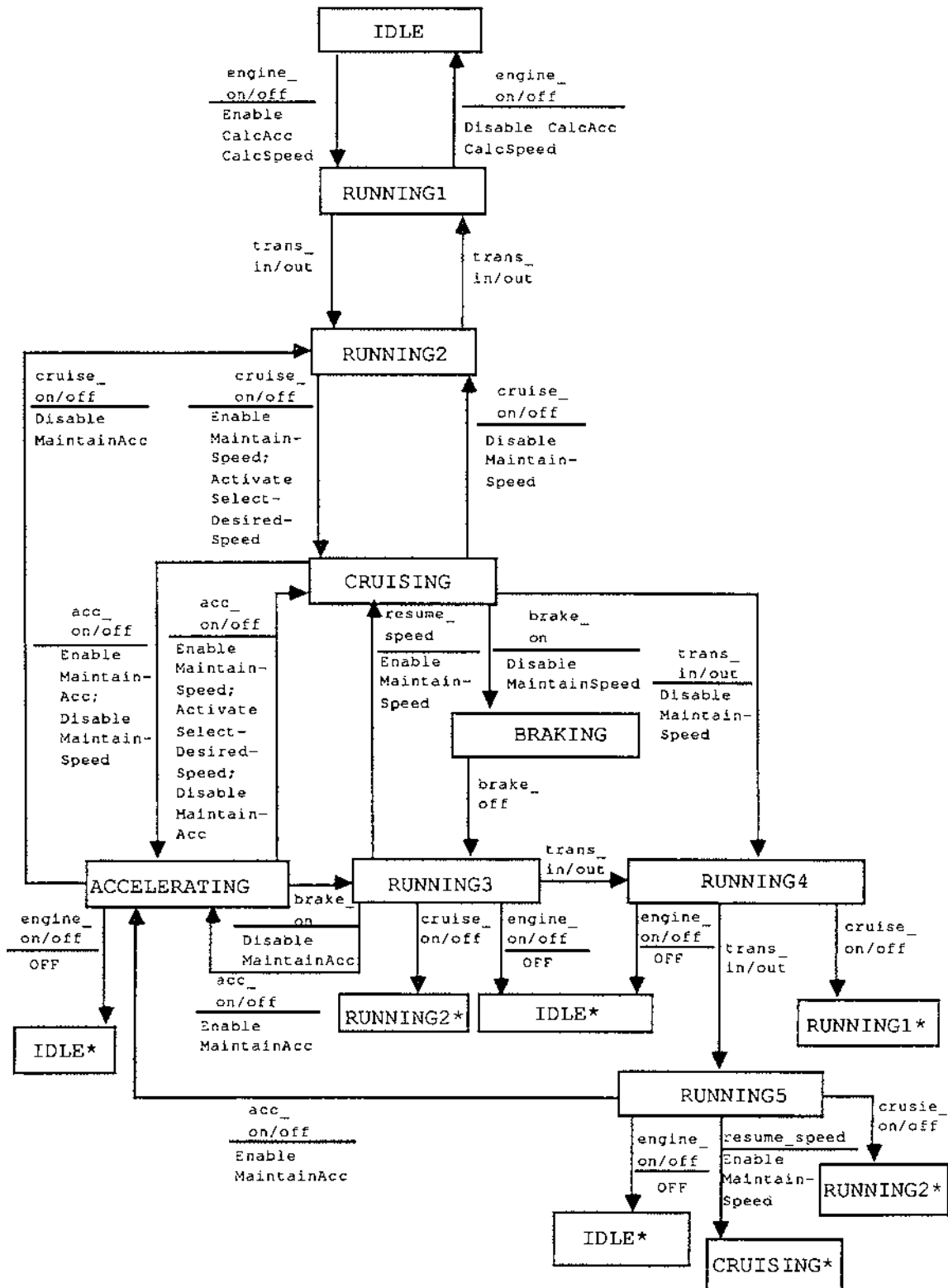


Figure 6.1 The ExtDFD for the Cruise Control Application



NOTE: OFF denotes the deactivation of all actions currently active

Figure 6.2 State Transition Diagram for the Cruise Control Application

All the data flows in the ExtDFD transmit objects of type number. Furthermore the constant MileCount, the number of shaft rotations in a single mile, is assumed to be defined in the RS Number. For clarity, numbers are written in their numeric form, for example the element representing the number two in Number is written as '2'.

The auxiliary functions needed to define the transition relations of the application's processes essentially define the functional relationship between their inputs and outputs. The RSs characterizing the auxiliary functions are given below.

Functional relationship between the input and output of CalcAcc

CalcAcc \equiv Number +

Signature

derivor

calcacc : number \rightarrow number

Laws \forall c:number

1. calcacc(c) = c/MileCount

Functional relationship between the input and output of CalcSpeed

CalcSpeed \equiv Number +

Signature

derivor

calcspeed : number \rightarrow number

Laws \forall c:number

1. calcspeed(c) = c/MileCount

Functional relationship between the input and output of MaintainSpeed

MaintainSpeed \equiv Number +

Signature

derivor

calcposn : number, number \rightarrow number

Laws \forall n1,n2:number

1. $n1-n2 > 2 \Rightarrow \text{calcposn}(n1, n2) = 0$
2. $n1-n2 \geq -2, n1-n2 \leq 2 \Rightarrow \text{calcposn}(n1, n2) = 2*(n1-n2+2)$
3. $n1-n2 < -2 \Rightarrow \text{calcposn}(n1, n2) = 8$

--- Varies throttle opening from closed to fully open as speed varies from 2 mph above desired speed, to 2 mph below it ---

Functional relationship between the input and output of MaintainAcc

MaintainAcc \equiv Number +

Signature**derivor**

calcthposn : number \rightarrow number

Laws \forall n:number

1. $n > 1.2 \Rightarrow \text{calcthposn}(n) = 0$
2. $n \geq 0.8, n \leq 1.2 \Rightarrow \text{calcthposn}(n) = 20 \cdot (1.2 - n)$
3. $n < 0.8 \Rightarrow \text{calcthposn}(n) = 0.8$

--- Varies throttle opening from closed to fully open as acceleration varies from 1.2 mph/sec to 0.8 mph/sec ---

Since the actions of the ExtDFD all consist of single processes, the states, labels and transition systems for the processes are specified as actions.

State, label, and transition system specification for CalcAcc

CalcAcc_State \equiv Number + P1substate +

Signature

sorts p1state, inp1, outp1

constructors

Nullinp1 : \rightarrow inp1
 Nulloutp1 : \rightarrow outp1
 inchangerate : number \rightarrow inp1
 outacc : number \rightarrow outp1
 <_,_> : inp1, outp1 \rightarrow p1state

CalcAcc_Labels \equiv Number +

Signature

sort p1label

constructors

Readchrte : number \rightarrow p1label
 Sendp1 : number \rightarrow p1label
 Terminatep1 : \rightarrow p1label

CalcAcc_TS \equiv CalcAcc_State + CalcAcc_Label + CalcAcc +

Signature

transition relation

$_ == _ ==> _ :$ p1state, p1label, p1state

Laws \forall a,c:number

1. $\langle \text{Nullinp2}, \text{Nulloutp2} \rangle == \text{Readchrate}(c) ==> \langle \text{inchangerate}(c), \text{Nulloutp2} \rangle$
2. $\text{calcacc}(c) = a \Rightarrow$
 $\langle \text{inchangerate}(c), \text{Nulloutp2} \rangle$
 $== \text{Sendp1}(a) ==>$
 $\langle \text{inchangerate}(c), \text{outacc}(a) \rangle$
3. $\langle \text{inchangerate}(c), \text{outacc}(a) \rangle == \text{Terminatep2} ==> \langle \text{Nullinp1}, \text{Nulloutp1} \rangle$

State, label, and transition system specification for CalcSpeed

CalcSpeed_State \equiv Number +

Signature

sorts p2state, inp2, outp2

constructors

Nullinp2 : \rightarrow inp2
 Nulloutp2 : \rightarrow outp2
 inrate: number \rightarrow inp2
 outspeed : number \rightarrow outp2
 $\langle _, _ \rangle$: inp2, outp2 \rightarrow p2state

CalcSpeed_Labels \equiv Number +

Signature

sort p2label

constructors

Readprate : number \rightarrow p2label
 Sendp2 : number \rightarrow p2label
 Terminatep2 : \rightarrow p2label

CalcSpeed_TS \equiv CalcSpeed_State + CalcSpeed_Label + CalcSpeed +

Signature

transition relation

$_ == _ ==> _ :$ p2state, p2label, p2state

Laws \forall n,cs:number

1. $\langle \text{Nullinp2}, \text{Nulloutp2} \rangle == \text{Readprate}(n) ==> \langle \text{inrate}(n), \text{Nulloutp2} \rangle$

2. $\text{calcspeed}(n) = \text{cs} \Rightarrow$
 $\langle \text{inrate}(n), \text{Nulloutp2} \rangle$
 $\quad \Rightarrow \text{Sendp2}(\text{cs}) \Rightarrow$
 $\langle \text{inrate}(n), \text{outspeed}(\text{cs}) \rangle$
3. $\langle \text{inrate}(n), \text{outspeed}(\text{cs}) \rangle \Rightarrow \text{Terminatp2} \Rightarrow \langle \text{Nullinp2}, \text{Nulloutp2} \rangle$

State, label, and transition system specification for *SelectDesiredSpeed*

$\text{SelectDesiredSpeed_State} \equiv \text{Number} +$

Signature

sorts p3state, inp3, outp3, outsig

constructors

Nullinp3: \rightarrow inp3

Nulloutp3: \rightarrow outp3

Nullsig: \rightarrow outsig

incspeed : number \rightarrow inp3

outdspeed: number \rightarrow outp3

less30: \rightarrow outsig

$\langle _, _, _ \rangle$: inp3, outp3, outsig \rightarrow p3state

$\text{SelectDesiredSpeed_Labels} \equiv \text{Number} +$

Signature

sort p3label

constructors

Readp3 : number \rightarrow p3label

Sendp3 : number \rightarrow p3label

Less30 : \rightarrow p3label

Terminatp3 : \rightarrow p3label

$\text{SelectDesiredSpeed_TS} \equiv \text{SelectDesiredSpeed_State} + \text{SelectDesiredSpeed_Label} +$

Signature

transition relation

$_ \Rightarrow _ \Rightarrow _$: p3state, p3label, p3state

Laws \forall **cs: number**

1. $\langle \text{Nullinp3}, \text{Nulloutp3}, \text{Nullsig} \rangle \Rightarrow \text{Readp3}(\text{cs}) \Rightarrow \langle \text{incspeed}(\text{cs}), \text{Nulloutp3}, \text{Nullsig} \rangle$
2. $\sim(\text{cs} < 30) \Rightarrow$
 $\langle \text{incspeed}(\text{cs}), \text{Nulloutp2}, \text{Nullsig} \rangle$
 $\quad \Rightarrow \text{Sendp3}(\text{cs}) \Rightarrow$
 $\langle \text{incspeed}(\text{cs}), \text{outdspeed}(\text{cs}), \text{Nullsig} \rangle$

3. $cs < 30 \Rightarrow$

```
<incspeed(cs), Nulloutp3, Nullsig>
  ==Less30==>
  <incspeed(cs), Nulloutp3, less30>
```

4. $\langle \text{incspeed}(cs), \text{outdspeed}(cs), \text{Nullsig} \rangle$

```
==Terminatep3==>
```

```
<Nullinp3, Nulloutp3, Nullsig>
```

5. $\langle \text{incspeed}(cs), \text{Nulloutp3}, \text{less30} \rangle$

```
==Terminatep3==>
```

```
<Nullinp3, Nulloutp3, Nullsig>
```

State, label, and transition system specification for MaintainSpeed

MaintainSpeed_State \equiv Number +

Signature

sorts p4state, in1p4, in2p4, outp4

constructors

Nullin1p4 : \rightarrow in1p4

Nullin2p4 : \rightarrow in2p4

Nulloutp4 : \rightarrow outp4

in1dspeed : number \rightarrow in1p4

in2cspeed : number \rightarrow in2p4

outposn : number \rightarrow outp4

$\langle _, _, _ \rangle$: in1p4, in2p4, outp4 \rightarrow p4state

MaintainSpeed_Labels \equiv Number +

Signature

sorts p4label

constructors

Read1p4, Read2p4 : number \rightarrow p4label

Sendp4 : number \rightarrow p4label

Terminatep4 : \rightarrow p4label

MaintainSpeed_TS \equiv MaintainSpeed_State + MaintainSpeed_Label + MaintainSpeed +

Signature

transition relation

$_ == _ == _ :$ p4state, p4label, p4state

Laws \forall in1:in1p4; in2:in2p4; s1,s2,pos:Number

1. $\langle \text{Nullin1p4}, \text{in2}, \text{Nulloutp4} \rangle \text{-Read1p4}(s1) \text{-} \langle \text{in1dspeed}(s), \text{in2}, \text{Nulloutp4} \rangle$

2. $\langle \text{in1}, \text{Nullin2p4}, \text{Nulloutp4} \rangle \text{-Read2p4}(s1) \text{-} \langle \text{in1}, \text{in2cspeed}(s), \text{Nulloutp4} \rangle$
3. $\text{calcposn}(s1, s2) = \text{pos} \Rightarrow$
 $\langle \text{in1dspeed}(s1), \text{in2dspeed}(s2), \text{Nulloutp4} \rangle$
 $\text{==Sendp4}(\text{pos})\text{==} \langle \text{in1dspeed}(s1), \text{in2dspeed}(s2), \text{outposn}(\text{pos}) \rangle$
4. $\langle \text{in1dspeed}(s1), \text{in2dspeed}(s2), \text{outposn}(\text{pos}) \rangle$
 $\text{==Terminatep4}\text{==} \langle \text{Nullin1p4}, \text{Nullin2p4}, \text{Nulloutp4} \rangle$

State, label, and transition system specification for MaintainAcc

MaintainAcc_State \equiv Number +

Signature

sorts p5state, inp5, outp5

constructors

Nullinp5 : \rightarrow inp5

Nulloutp5 : \rightarrow outp5

inacc : number \rightarrow inp5

throtposn : number \rightarrow outp5

$\langle _ _ \rangle$: inp5, outp5 \rightarrow p5state

MaintainAcc_Labels \equiv Number +

Signature

sorts p5label

constructors

Readp5 : number \rightarrow p5label

Sendp5 : number \rightarrow p5label

Terminatep5 : \rightarrow p5label

MaintainAcc_TS \equiv MaintainAcc_State + MaintainAcc_Label + MaintainAcc +

Signature

transition relation

$_ _ \rightarrow _$: p5state, p5label, p5state

Laws \forall a, pos: number

1. $\langle \text{Nullinp5}, \text{Nulloutp5} \rangle \text{==Readp5}(a) \text{==} \langle \text{inacc}(a), \text{Nulloutp5} \rangle$
2. $\text{calc}(\text{throtposn}(a)) = \text{pos} \Rightarrow$
 $\langle \text{inacc}(a), \text{Nulloutp5} \rangle \text{==Sendp5}(\text{pos}) \text{==} \langle \text{inacc}(a), \text{throtposn}(\text{pos}) \rangle$
3. $\langle \text{inacc}(a), \text{throtposn}(\text{pos}) \rangle \text{==Terminatep5} \text{==} \langle \text{Nullinp5}, \text{Nulloutp5} \rangle$

The data stores `CURRENTSPEED`, `DESIRESPEED`, and `CURRENTACC`, behave like variables in the sense that they contain a single value which is overwritten when the data store is written into. The transition systems characterizing the effects of reads and writes on the data store states are given below.

State, label, and transition system specification for `CURRENTSPEED`

`CurrentSpeed_TS` = Number +

Signature

sorts `currspeed`, `cslabel`

constructors

`Nullspeed` : \rightarrow `currspeed`

`valcs` : `number` \rightarrow `currspeed`

`Putcs` : `number` \rightarrow `cslabel`

`Getcs` : `number` \rightarrow `cslabel`

transition relation

$_{cs} == _{cs} ==> _{cs}$: `currspeed`, `cslabel`, `currspeed`

Laws \forall `s:number`; `cs:currspeed`

1. `cs==Putcs(s)==>valcs(s)`

2. `valcs(s)==Getcs(s)==>valcs(s)`

State, label, and transition system specification for `DESIRESPEED`

`DesiredSpeed_TS` = Number +

Signature

sorts `despeed`, `dslabel`

constructors

`Nullspeed` : \rightarrow `despeed`

`valds` : `number` \rightarrow `despeed`

`Putds` : `number` \rightarrow `dslabel`

`Getds` : `number` \rightarrow `dslabel`

transition relation

$_{ds} == _{ds} ==> _{ds}$: `despeed`, `dslabel`, `despeed`

Laws \forall `s:number`; `ds:despeed`

1. `ds==Putds(s)==>valds(s)`

2. `valds(s)==Getds(s)==>valds(s)`

State, label, and transition system specification for CURRENTACC

Acc_TS \equiv Number +

Signature

sorts acc, alabel

constructors

Nullacc : \rightarrow acc

valacc : number \rightarrow acc

Putacc : number \rightarrow calabel

Geta : number \rightarrow alabel

transition relation

$_ == _ ==> _ :$ acc, alabel, acc

Laws \forall a:number; da:acc

1. da==Putacc(a)==>valacc(a)
2. valacc(a)==Geta(a)==>valacc(a)

Specification of application states

CruiseSys_State \equiv CalcAcc_State + CalcSpeed_State + SelectDesiredSpeed_State +
MaintainSpeed_State + MaintainAcc_State + CurrentSpeed_TS + DesiredSpeed_TS +
Acc_TS +

Signature

sorts state, systate, sysflag

constructor

IDLE, RUN1, RUN2, RUN3, RUN4, RUN5, CRUISE, ACCEL, BRAKING : \rightarrow sysflag

--- modes of operation ---

P1 : p1state \rightarrow pstate

.

.

P5 : p5state \rightarrow pstate

--- state coercion functions, that is, functions which make states of actions into

ExtDFD process states ---

$\langle _ _ _ _ \rangle :$ set(pstate), acc, currspeed, despeed, asynch1, asynch2,
sysflag \rightarrow systate

--- the state of an ExtDFD ---

ok-predicate

okstate: systate

Laws \forall ac:acc; cs:currspeed; ds:despeed; p1:p1state; p2:p2state;
p3:p3state; p4:p4state; p5:p5state; as1:asynch1, as2:asynch2

1. okstate($\langle \emptyset, \text{Nullacc}, \text{Nullspeed}, \text{Nullspeed}, \text{as1}, \text{as2}, \text{IDLE} \rangle$)

2. $\text{okstate}(\langle\{p1, p2\}, \text{ac}, \text{cs}, \text{Nullspeed}, \text{as1}, \text{as2}, \text{RUN1}\rangle)$
3. $\text{okstate}(\langle\{p1, p2\}, \text{ac}, \text{cs}, \text{Nullspeed}, \text{as1}, \text{as2}, \text{RUN2}\rangle)$
4. $\text{okstate}(\langle\{p1, p2, p3, p4\}, \text{ac}, \text{cs}, \text{ds}, \text{as1}, \text{as2}, \text{CRUISE}\rangle)$
5. $\text{okstate}(\langle\{p1, p2, p5\}, \text{ac}, \text{cs}, \text{ds}, \text{as1}, \text{as2}, \text{ACCEL}\rangle)$
6. $\text{okstate}(\langle\{p1, p2\}, \text{ac}, \text{cs}, \text{ds}, \text{as1}, \text{as2}, \text{BRAKING}\rangle)$
7. $\text{okstate}(\langle\{p1, p2\}, \text{ac}, \text{cs}, \text{ds}, \text{as1}, \text{as2}, \text{RUN3}\rangle)$
8. $\text{okstate}(\langle\{p1, p2\}, \text{ac}, \text{cs}, \text{ds}, \text{as1}, \text{as2}, \text{RUN4}\rangle)$
9. $\text{okstate}(\langle\{p1, p2\}, \text{ac}, \text{cs}, \text{ds}, \text{as1}, \text{as2}, \text{RUN5}\rangle)$

Specification of action labels

$\text{Cruise_Label} \equiv \text{CalcAcc_Label} + \text{CalcSpeed_Label} + \text{SelectDesiredSpeed_Label} +$
 $\text{MaintainSpeed_Label} + \text{MaintainAcc_Label} + \text{CurrentSpeed_TS} + \text{DesiredSpeed_TS} +$
 $\text{Acc_TS} +$

Signature

sort label!

L1 : p1label \rightarrow label

.

.

L5 : p4label \rightarrow label

Lcs : cslabel \rightarrow label

Lds : dslabel \rightarrow label

La : alabel \rightarrow label

Specification of application labels

$\text{CruiseSys_Label} \equiv \text{Set}(\text{Cruise_Label}) +$

Signature

sort syslabel!

[] : label \rightarrow syslabel

SYNCH : set(label) \rightarrow syslabel

|| : syslabel, syslabel \rightarrow syslabel

The following shorthand notation will be used in the laws that follow.

- $\{p1, p2, \dots, pn\}$ denotes the finite set consisting of the elements $p1$ to pn .
- $\text{insert}(p, sp)$ will be written as $\{p, sp\}$
- Coercion functions for both states and labels will be left implicit, where doing so causes no confusion. For example, the systate $\langle\{P1(p1), \dots, P5(p5)\}, \dots\rangle$ will simply be written as $\langle\{p1, \dots, p5\}, \dots\rangle$.

Specification of the application's transition system

CruiseSys_TS \equiv CalcAcc_TS + CalcSpeed_TS + SelectDesiredSpeed_TS +
 MaintainSpeed_TS + MaintainAcc_TS + CurrentSpeed_TS + DesiredSpeed_TS +
 Acc_TS +

Signature

transition relation

$_====_====>_ :$ systate, syslabel, systate

Laws $\forall a,s,c:\text{number}; ac,ac':\text{acc}; cs,cs':\text{currspeed}; ds,ds':\text{despeed};$
f:sysflag;

sp,sp1,sp1',sp2,sp2':set(pstate); p1,p1':p1state; p2,p2':p2state;
p3,p3':p3state; p4,p4':p4state; p5,p5':p5state

Synchronized events: data store/action interactions

1. $p1==\text{Sendp1}(a)==>p1', ac==\text{Put}(a)==>ac' \Rightarrow$
 $\langle\{p1, sp\}, ac, cs, ds, as1, as2, f\rangle$
 $====\text{SYNCH}(\{\text{Sendp1}(a), \text{Put}(a)\})====>$
 $\langle\{p1', sp\}, ac', cs, ds, as1, as2, f\rangle$
 -- synchronized write to data store CURRENTACC by CalcACC --
2. $p5==\text{Readp5}(a)==>p5', ac==\text{Geta}(a)==>ac \Rightarrow$
 $\langle\{p5, sp\}, ac, cs, ds, as1, as2, f\rangle$
 $====\text{SYNCH}(\{\text{Readp5}(a), \text{Geta}(a)\})====>$
 $\langle\{p5', sp\}, ac, cs, ds, as1, as2, f\rangle$
 -- synchronized read from data store CURRENTACC by MaintainAcc --
3. $p2==\text{Sendp2}(s)==>p2', cs==\text{Putcs}(s)==>cs' \Rightarrow$
 $\langle\{p2, sp\}, ac, cs, ds, as1, as2, f\rangle$
 $====\text{SYNCH}(\{\text{Sendp2}(s), \text{Putcs}(s)\})====>$
 $\langle\{p2', sp\}, ac, cs', ds, as1, as2, f\rangle$
 -- synchronized write to CURRENTSPEED by CalcSpeed --
4. $p3==\text{Readp3}(s)==>p3', cs==\text{Getcs}(s)==>cs \Rightarrow$
 $\langle\{p3, sp\}, ac, cs, ds, as1, as2, f\rangle$
 $====\text{SYNCH}(\{\text{Readp3}(s), \text{Getcs}(s)\})====>$
 $\langle\{p3', sp\}, ac, cs, ds, as1, as2, f\rangle$
 -- synchronized read on CURRENTSPEED by SelectDesiredSpeed --
5. $p3==\text{Sendp3}(s)==>p3', ds==\text{Putds}(s)==>ds' \Rightarrow$
 $\langle\{p3, sp\}, ac, cs, ds, as1, as2, f\rangle$
 $====\text{SYNCH}(\{\text{Sendp3}(s), \text{Putds}(s)\})====>$
 $\langle\{p3', sp\}, ac, cs, ds', as1, as2, f\rangle$
 -- synchronized write to DESIREDSPD by SelectDesiredSpeed --

6. `p4==Read1p4(s)==>p4'`, `ds==Getds(s)==>ds =>`
`<{p4, sp}, ac, cs, ds, as1, as2, f>`
`===SYNCH({Read1p4(s), Getds(s)})===>`
`<{p4', sp}, ac, cs, ds, as1, as2, f>`
 -- synchronized read on `DESIRESPEED` by `MaintainSpeed` --
7. `p4==Read2p4(cs)==>p4'`, `cs==Getcs(s)==>cs =>`
`<{p4, sp}, ac, cs, ds, as1, as2, f>`
`===SYNCH({Read2p4(s), Getcs(s)})===>`
`<{p4', sp}, ac, cs, ds, as1, as2, f>`
 -- synchronized read on `CURRENTSPEED` by `MaintainSpeed` --
8. `p4==Sendp4(pos)==>p4'`, `as1==ADD1(pos)==>as1'`
`<{p4, sp}, ac, cs, ds, as1, as2, f>`
`===SYNCH({Sendp4(pos), ADD1(pos)})===>`
`<{p4', sp}, ac, cs, ds, as1', as2, f>`
 --- synchronized add access to data flow throttle_position1 by `MaintainSpeed` ---
9. `p4==Sendp5(pos)==>p5'`, `as2==ADD2(pos)==>as2'`
`<{p5, sp}, ac, cs, ds, as1, as2, f>`
`===SYNCH({Sendp5(pos), ADD2(pos)})===>`
`<{p5', sp}, ac, cs, ds, as1, as2', f>`
 --- synchronized add access to data flow throttle_position2 by `MaintainAcc` ---

Control events

-- The following transitions are derived directly from the STD for the application (see Figure 6.2) --

10. `<∅, Nullacc, Nullspeed, Nulldspeed, as1, as2, IDLE>`
`===engine_on/off===>`
`<{<Nullinp1, Nulloutp1>, <Nullinp2, Nulloutp2>},`
`Nullacc, Nullspeed, Nulldspeed, as1, as2, RUN1>`
11. `<{p1, p2}, ac, cs, Nulldspeed, as1, as2, RUN1>`
`===trans_in/out===>`
`<{p1, p2}, ac, cs, Nulldspeed, as1, as2, RUN2>`
12. `f ≠ IDLE =>`
`<sp, ac, cs, ds, as1, as2, f>`
`===engine_on/off===>`
`<∅, Nullacc, Nullspeed, Nulldspeed, as1, as2, IDLE>`
13. `f ≠ RUN1, f ≠ RUN2, f ≠ RUN4, f ≠ IDLE =>`
`<sp, ac, cs, ds, as1, as2, f>`
`===cruise_on/off===>`
`<{p1, p2}, ac, cs, Nulldspeed, as1, as2, RUN2>`

14. <{p1, p2}, ac, cs, Nullspeed, as1, as2, RUN2>
 ===cruise_on/off===>
 <{p1, p2, <Nullinp3, Nulloutp3>, <Nullin1p4, Nullin2p4, Nulloutp4>},
 ac, cs, Nullspeed, as1, as2, CRUISE>
15. <{p1, p2}, ac, cs, Nullspeed, as1, as2, RUN2>
 ===trans_in/out===>
 <{p1, p2}, ac, cs, Nullspeed, as1, as2, RUN1>
16. <{p1, p2, p3, p4}, ac, cs, valds(s), as1, as2, CRUISE>
 ===acc_on/off===>
 <{p1, p2, <Nullinp5, Nulloutp5>}, ac, cs, valds(s), as1, as2, ACCEL>
17. <{p1, p2, p3, p4}, ac, cs, valds(s), as1, as2, CRUISE>
 ===brake_on===>
 <{p1, p2}, ac, cs, valds(s), as1, as2, BRAKING>
18. <{p1, p2, p3, p4}, ac, cs, valds(s), as1, as2, CRUISE>
 ===trans_in/out===>
 <{p1, p2}, ac, cs, valds(s), as1, as2, RUN4>
19. <{p1, p2, p3, p4}, ac, cs, Nullspeed, as1, as2, CRUISE>
 ===(Less30)===>
 <{p1, p2}, ac, cs, Nullspeed, as1, as2, RUN2>
20. <{p1, p2, p5}, ac, cs, ds, as1, as2, ACCEL>
 ===brake_on===>
 <{p1, p2}, ac, cs, ds, as1, as2, BRAKING>
21. <{p1, p2, p5}, ac, cs, ds, as1, as2, ACCEL>
 =====acc_on/off=====>
 <{p1, p2, <Nullinp3, Nulloutp3>, <Nullin1p4, Nullin2p4, Nulloutp4>},
 ac, cs, Nullspeed, as1, as2, CRUISE>
22. <{p1, p2}, ac, cs, ds, as1, as2, BRAKING>
 ===brake_off===>
 <{p1, p2}, ac, cs, ds, as1, as2, RUN3>
23. <{p1, p2}, ac, cs, ds, as1, as2, RUN3>
 ===acc_on/off=====>
 <{p1, p2, <Nullinp5, Nulloutp5>}, ac, cs, ds, as1, as2, ACCEL>
24. <{p1, p2}, ac, cs, ds, as1, as2, RUN3>
 =====trans_in/out=====>
 <{p1, p2}, ac, cs, ds, as1, as2, RUN4>
25. <{p1, p2}, ac, cs, ds, as1, as2, RUN3>
 =====resume_speed=====>
 <{p1, p2, <Nullin1p4, Nullin2p4, Nulloutp4>}, ac, cs, ds, as1, as2, CRUISE>

26. <<{p1, p2}, ac, cs, ds>, as1, as2, RUN4>
 ===cruise_on/off===>
 <{p1, p2}, ac, cs, Nullspeed, as1, as2, RUN1>
27. <{p1, p2}, ac, cs, ds, as1, as2, RUN4>
 ===trans_in/out===>
 <{p1, p2}, ac, cs, ds, as1, as2, RUN5>
28. <{p1, p2}, ac, cs, ds, as1, as2, RUN5>
 ===acc_on/off===>
 <{p1, p2, <Nullinp5, Nulloutp5>}, ac, cs, ds, as1, as2, ACCEL>
29. <{p1, p2}, ac, cs, ds, as1, as2, RUN5>
 ===resume_speed===>
 <{p1, p2, <Nullin1p4, Nullin2p4, Nulloutp4>}, ac, cs, ds, as1, as2, CRUISE>
30. p3==Terminatedp3==<Nullinp3, Nulloutp3, Nullsig>
 <{p1, p2, p3, p4}, ac, cs, ds, as1, as2, CRUISE>
 ===Killp3===>
 <{p1, p2, p4}, ac, cs, ds, as1, as2, CRUISE>

Synchronized events: asynchronous data flow/action interactions

31. p4==Sendp4(pos)==>p4', as1==ADD1(pos)==>as1' =>
 <{p4, sp}, ac, cs, ds, as1, as2, f>
 ===SYNCH({Sendp4(pos), ADD1(pos)})===>
 <{p4', sp}, ac, cs, ds, as1', as2, f>
32. p5==Sendp5(pos)==>p5', as2==ADD2(pos)==>as2' =>
 <{p5, sp}, ac, cs, ds, as1, as2, f>
 ==SYNCH{Sendp5(pos), ADD2(pos)}==>
 <{p5', sp}, ac, cs, ds, as1, as2', f>

Action/state data flow interactions

33. p1==Readchrate==>p1' =>
 <{p1, sp}, ac, cs, ds, as1, as2, f>
 ===Readchrate===>
 <{p1', sp}, ac, cs, ds, as1, as2, f>
34. p2==Readprate==>p2' =>
 <{p2, sp}, ac, cs, ds, as1, as2, f>
 ===Readprate===>
 <{p2', sp}, ac, cs, ds, as1, as2, f>

Parallel events

35. <{p1, p5, sp}, ac, cs, ds, as1, as2, f>
 ===SYNCH({Sendp1(a), Puta(a)})===>
 <{p1', p5, sp}, ac', cs, ds, as1, as2, f>,
 <{p1, p5, sp}, ac', cs, ds, as1, as2, f>
 ===SYNCH({Readp5(a), Get(a)})===>
 <{p1, p5', sp}, ac', cs, ds, as1, as2, f> =>
 <{p1, p5, sp}, ac, cs, ds, as1, as2, f>
 ===SYNCH({Sendp1(a), Puta(a)})||SYNCH({Readp5(a), Get(a)})===>
 <{p1', p5', sp}, ac', cs, ds, as1, as2, f>
 -- parallel access to the data store CURRENTACC gives priority to the
 write access --
36. <{p2, p3, sp}, ac, cs, ds, as1, as2, f>
 ===SYNCH({Sendp2(s), Putcs(s)})===>
 <{p2', p3, sp}, ac, cs', ds, as1, as2, f>,
 <{p2, p3, sp}, ds, cs', ds, as1, as2, f>
 ===SYNCH({Readp3(s), Getcs(s)})===>
 <{p2, p3', sp}, ds, cs', ds, as1, as2, f> =>
 <{p2, p3, sp}, ds, cs, ds, as1, as2, f>
 ===SYNCH({Sendp2(s), Putcs(s)})||SYNCH({Readp3(s), Getcs(s)})===>
 <{p2', <inspeed(s1), Nulloutp3>, sp}, ac, cs', ds, as1, as2, f>
 -- parallel access to the data store CURRENTSPEED gives priority to the
 write access --
37. <{p4, sp}, ac, cs, ds, as1, as2, f>
 ===SYNCH({Read1p4(s), Getds(s)})===>
 <{<inspeed(s1), Nullin2p4, Nulloutp4>, sp}, ac, cs, ds, as1, as2, f>,
 <{p4, sp}, ac, cs, ds, as1, as2, f>
 ===SYNCH({Read2p4(s2), Getcs(s2)})===>
 <{<Nullin1p4, incspeed(s2), Nulloutp4>, sp}, ac, cs, ds, as1, as2, f> =>
 <{p4, sp}, ac, cs, ds, as1, as2, f>
 ===SYNCH({Read1p4(s1), Getds(s1)})||
 SYNCH({Read2p4(s2), Getcs(s2)})===>
 <{<inspeed(s1), incspeed(s2), Nulloutp4>, sp}, ac, cs, ds, as1, as2, f>
 -- parallel access to the data store DESIREDSPPEED gives priority to the
 write access --

38. <sp1+sp2, ac, cs, ds, as1, as2, f>
 ===A1===>
 <sp1'+sp2, ac', cs, ds, as1, as2, f>,
 <sp1+sp2, ac, cs, ds, as1, as2, f>
 ===A2===>
 <sp1+sp2', ac, cs', ds', as1', as2', f> ⇒
 <sp1+sp2, ac, cs, ds, as1, as2, f>
 ===A1||A2===>
 <sp1'+sp2', ac', cs', ds', as1', as2', f>
39. <sp1+sp2, ac, cs, ds, as1, as2, f>
 ===A1===>
 <sp1'+sp2, ac', cs', ds, as1, as2, f>,
 <sp1+sp2, ac, cs, ds, as1, as2, f>
 ===A2===>
 <sp1+sp2', ac, cs, ds', as1', as2', f> ⇒
 <sp1+sp2, ac, cs, ds, as1, as2, f>
 ===A1||A2===>
 <sp1'+sp2', ac', cs', ds', as1', as2', f>
40. <sp1+sp2, ac, cs, ds, as1, as2, f>
 ===A1===>
 <sp1'+sp2, ac', cs', ds', as1, as2, f>,
 <sp1+sp2, ac, cs, ds, as1, as2, f>
 ===A2===>
 <sp1+sp2', ac, cs, ds, as1', as2', f> ⇒
 <sp1+sp2, ac, cs, ds, as1, as2, f>
 ===A1||A2===>
 <sp1'+sp2', ac', cs', ds', as1', as2', f>
41. <sp1+sp2, ac, cs, ds, as1, as2, f>
 ===A1===>
 <sp1'+sp2, ac', cs', ds', as1', as2, f>,
 <sp1+sp2, ac, cs, ds, as1, as2, f>
 ===A2===>
 <sp1+sp2', ac, cs, ds, as1, as2', f> ⇒
 <sp1+sp2, ac, cs, ds, as1, as2, f>
 ===A1||A2===>
 <sp1'+sp2', ac', cs', ds', as1', as2', f>

-- Events that affect mutually exclusive parts of the application can occur in parallel --

6.2 Computer-based University Library Application

The ExtDFD for the library application is shown in Figure 6.3. The application is partitioned into seven actions namely `DeleteCopy (A1)`, `AddCopy (A2)`, `ReturnBook (A3)`, `CheckoutBook (A4)`, `UpdateBorrStatus (A5)`, `AddBorrower (A6)`, and `DeleteBorrower (A7)`. The actions communicate with the external entity `staff` via asynchronous data flows, while communication with the `clock` external entity is via state flows. Actions are activated solely by the occurrence of data events as is evident by the lack of control flows in the ExtDFD. Figure 6.4 gives the type definitions associated with the data flows in the ExtDFD.

Figure 6.4 defines, in a semi-formal manner, the type definitions of the data flows in the ExtDFD for the library application shown in Figure 6.3. Base types are classes of indivisible objects, or list or set structures of indivisible objects, while non-base types are classes of composite objects. In the definitions for the non-base types, the base components are written in bold. The base types used for the library application are:

number - the class of floating point numbers,
time - the class of time points,
character - the class of characters,
and list and set structures of the above.

Aliases for the base types are also defined in Figure 6.4, where the base types are differentiated from their aliases by writing them in italics. The names of the constructors in the RSs formally defining the types are enclosed within () in the definitions given in Figure 6.4. Constructors starting with a capital letter are constants.

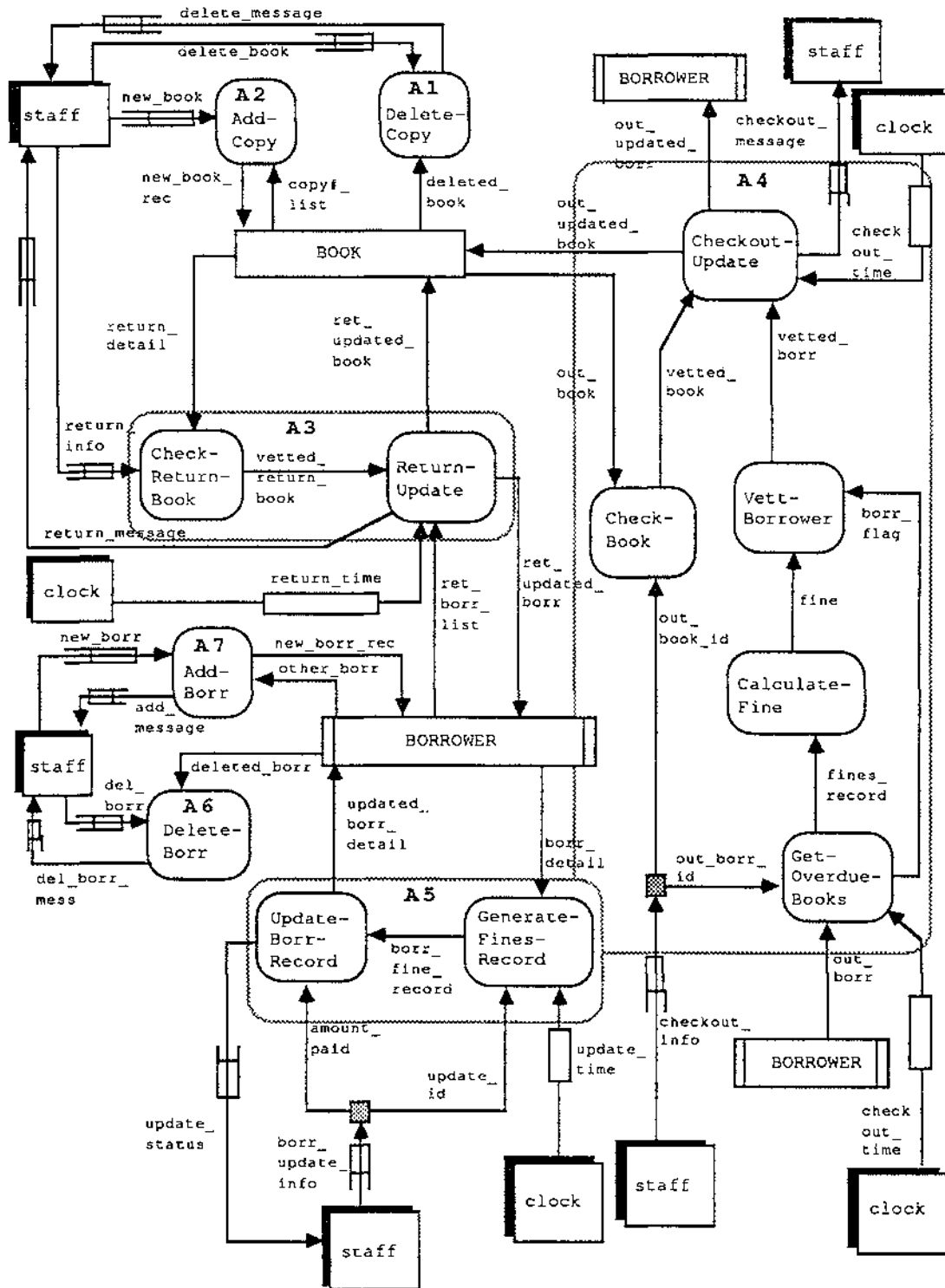


Figure 6.3 The ExtDFD for the library application

Non-base data types

```

bb_status      ::= <time (mkbbstatus)| "Not returned"
                (NotRet)>
book           ::= <book_id, title, subject, author,
                copy_type, borrower_indicator>
                (mkbook)
book_id        ::= <ISBN, copy#> (mkbkid)
borr_detail    ::= <[borrower_book_detail], number>
                mkbordet
borr_fine_record ::= <<number, borrower_id> (mkbfrec) | "Not
                in file" (NoRec)>
borr_flag      ::= <"Not in file" (Bflag)| <borrower_id,
                out_borr> (mkbfilag)>
borr_update_info ::= <borrower_id, number> (mkupinfo)
borrower       ::= <borrower_id, borrower_name,
                borrower_addr, borrower_type,
                [borrower_book_detail],
                payment_to_date> (mkborr)
borrower_book_detail ::= <book_id, due_time, bb_status>
                (mkbdet)
borrower_id    ::= <[character]> (mkborrid)
borrower_indicator ::= <"Available" (Available)| <borrower_id>
                (mkbind)>
checkout_info  ::= <book_id, borrower_id> (mkoutinfo)
checkout_message ::= <vetted_borr, vetted_book> (mkoutmess)
del_borr       ::= borrower_id
delete_book    ::= book_id
deleted_borr   ::= [borrower_book_detail]
deleted_book   ::= borrower_indicator
ISBN           ::= <[integer]>
new_book       ::= <ISBN, title, subject, author,
                copy_type> (mknewbk)
new_book_rec   ::= book
new_borr       ::= <borrower_id, borrower_name,
                borrower_addr, borrower_type>
                (mknewborr)
new_borr_rec   ::= borrower
other_borr     ::= borrower_id
out_book       ::= <borrower_indicator, copy_type>
                (mkoutbk)
out_book_id    ::= book_id
out_borr       ::= <[borrower_book_detail],
                borrower_type, payment_to_date>
                (mkoutbr)
out_borr_id    ::= borrower_id
out_updated_book ::= borrower_indicator
out_updated_borr ::= [borrower_book_detail]
ret_borr_list  ::= [borrower_book_detail]
ret_updated_book ::= borrower_indicator
ret_updated_borr ::= [borrower_book_detail]
return_detail  ::= borrower_indicator
return_info    ::= book_id
update_id      ::= borrower_id
update_status  ::= <"Outstanding" number (mkupstatus1)|
                "Excess" number (mkupstatus2)| "Not in
                file" (Norec)| "No fines" (Nofines)
                | "Fines cleared" (Cleared)>
vetted_book    ::= <<book_id, copy_type> (mkvbk)| "book
                not in file" (Bknotinfile)| "book already
                checked out" (CheckedOut)| "not
                borrowable" (NotBorr)>

```

```

vetted_borr      ::= <"Fines over limit" number (mkerrvborr) |
                  "borrower not in file" (NoBorr) |
                  <out_borr, borrower_id> (mkvetborr)>
vetted_return_book ::= <"Not in file" (Retnotinfile) | "Already
                  returned" (Retin) | <book_id,
                  borrower_id> (mkvetret)>

Base data types
add_message      ::= "OK" (OKadd) | "Borrower already in
                  file" (Alreadyinfile)
amount_paid      ::= number
author           ::= [character]
borrower_addr    ::= [character]
borrower_name    ::= [character]
borrower_type    ::= "undergrad" (Undergrad) | postgrad"
                  (Postgrad) | "staff" (Staff)
checkout_time    ::= time
copy#            ::= integer
copy#_list       ::= [integer]
copy_type        ::= "book" (Book) | "reference" (Ref) |
                  "periodical" (Per)
del_borr_mess    ::= "OK" (OKdel) | "Not in file" (Delnotinfile) |
                  "Has books out" (Bookscout)
delete_message   ::= <"delete-OK" (OKdel) | "Not in file"
                  (Nobk) | "Not available" (Notavailable)>
fine             ::= number
fines_record     ::= [number]
ISBN             ::= [integer]
new_copy#        ::= integer
paidup_amount    ::= number
payment_to_date  ::= number
return_message   ::= "Already in" (Alreadyin) | "Not in file"
                  (Retnotin)
return_time      ::= time
subject          ::= [character]
title            ::= [character]
update_time      ::= time
updated_borr_detail ::= number

```

Figure 6.4 Type definitions for the library application

The ASTS for the action DeleteCopy (A1)

The states of A1 are of the form $\langle \text{inp1}, \text{delp1}, \text{outp1} \rangle$, where inp1 is the state associated with the action's interaction with `delete_book`, delp1 is the state associated with the action's interaction with the data store `BOOK`, and outp1 is the state associated with the action's interaction with `delete_message`. The type of the state, p1state , is characterized by the RS P1state .

The labels of A1, of sort p1label characterized by P1label , are:

- $\text{Receivep1}(d1)$ - receive $d1$ from `delete_book`.
- $\text{Readp1}(id, d2)$ - read in $d2$ from `BOOK`.
- $\text{Erreadp1}(id)$ - unsuccessful read on `BOOK`.
- $\text{Sendp1}(d3)$ - generate $d3$ for output on `delete_message`.
- $\text{Deletep1}(id)$ - delete object with key id from `BOOK`.

- Terminatep1 - terminate action.

The ASTS for A1 is given below:

DeleteCopy_TS \equiv P1state + P1label +

Signature

transition relation

$_ == _ ==> _ : p1state, p1label, p1state$

Laws \forall borrid:borrower_id; bid:book_id; dmess:delete_message;

bind:borrower_indicator

1. $\langle \text{Nullinp1}, \text{Nulldelp1}, \text{Nulloutp1} \rangle$
 $\quad == \text{Receivep1}(\text{bid}) ==>$
 $\quad \langle \text{inp1}(\text{bid}), \text{Nulldelp1}, \text{Nulloutp1} \rangle$
2. $\langle \text{inp1}(\text{bid}), \text{Nulldelp1}, \text{Nulloutp1} \rangle$
 $\quad == \text{Readp1}(\text{bid}, \text{bind}) ==>$
 $\quad \langle \text{inp1}(\text{bid}), \text{delp1}(\text{bind}), \text{Nulloutp1} \rangle$
3. $\langle \text{inp1}(\text{bid}), \text{Nulldelp1}, \text{Nulloutp1} \rangle$
 $\quad == \text{Erreadp1}(\text{bid}) ==>$
 $\quad \langle \text{inp1}(\text{bid}), \text{errdelp1}, \text{Nulloutp1} \rangle$
4. $\langle \text{inp1}(\text{bid}), \text{delp1}(\text{mkbind}(\text{borrid})), \text{Nulloutp1} \rangle$
 $\quad == \text{Sendp1}(\text{NotAvailable}) ==>$
 $\quad \langle \text{inp1}(\text{bid}), \text{delp1}(\text{mkbind}(\text{borrid})), \text{outp1}(\text{NotAvailable}) \rangle$
5. $\langle \text{inp1}(\text{bid}), \text{delp1}(\text{Available}), \text{Nulloutp1} \rangle$
 $\quad == \text{Deletep1}(\text{bid}) ==>$
 $\quad \langle \text{inp1}(\text{bid}), \text{deleted}, \text{Nulloutp1} \rangle$
6. $\langle \text{inp1}(\text{bid}), \text{deleted}, \text{Nulloutp1} \rangle$
 $\quad == \text{Sendp1}(\text{OKdel}) ==>$
 $\quad \langle \text{inp1}(\text{bid}), \text{deleted}, \text{outp1}(\text{OKdel}) \rangle$
7. $\langle \text{inp1}(\text{bid}), \text{errdelp1}, \text{Nulloutp1} \rangle$
 $\quad == \text{Sendp1}(\text{De!notin}) ==>$
 $\quad \langle \text{inp1}(\text{bid}), \text{errdelp1}, \text{outp1}(\text{De!notin}) \rangle$
8. $\langle \text{inp1}(\text{bid}), \text{errdelp1}, \text{outp1}(\text{dmess}) \rangle$
 $\quad == \text{Terminatep1} ==>$
 $\quad \langle \text{Nullinp1}, \text{Nulldelp1}, \text{Nulloutp1} \rangle$
9. $\langle \text{inp1}(\text{bid}), \text{deleted}, \text{outp1}(\text{OKdel}) \rangle$
 $\quad == \text{Terminatep1} ==>$
 $\quad \langle \text{Nullinp1}, \text{Nulldelp1}, \text{Nulloutp1} \rangle$

The ASTS for the action AddCopy (A2)

The states of A2 are of the form $\langle \text{inp2}, \text{rdp2}, \text{outp2} \rangle$, where inp2 is the state associated with the action's interaction with `return_info`, rdp2 is the state associated with the action's interaction with the data store `BOOK`, and outp2 is the state associated with the action's interaction with `vetted_return_book`. A2 states are of sort `p2state` characterized by the RS `P2state`.

The labels of A2, of sort `p2label` characterized by `P2label`, are:

- `Receivep2(d1)` - receive $d1$ from `return_info`.
- `Readp2(id, d2)` - read in $d2$ from `BOOK`.
- `Erreadp2(id)` - unsuccessful read on `BOOK`.
- `Sendp2(d3)` - generate $d3$ for output on `vetted_return_book`.
- `Terminatep2` - terminate action.

The ASTS for A2 is given below:

`AddCopy_TS` \equiv `P2state` + `P2label` +

Signature

transition relation

$_ == _ == _ : \text{p2state}, \text{p2label}, \text{p2state}$

Laws $\forall \text{nbk}:\text{new_book}; \text{i}:\text{ISBN}; \text{t,s,a}:\text{list}(\text{character}); \text{ln}:\text{list}(\text{integer});$

ty:copy_type; bkrec:new_book_rec; n1:integer

1. $\langle \text{Nullinp2}, \text{Nullrdp2}, \text{Nulloutp2} \rangle$
 $\quad == \text{Receivep2}(\text{nbk}) == \rangle$
 $\langle \text{inp2}(\text{nbk}), \text{Nullrdp2}, \text{Nulloutp2} \rangle$
2. $\langle \text{inp2}(\text{mknewbk}(\text{i,t,s,a,ty})), \text{Nullrdp2}, \text{Nulloutp2} \rangle$
 $\quad == \text{Readp2}(\text{i}, \text{ln}) == \rangle$
 $\langle \text{inp2}(\text{mknewbk}(\text{i,t,s,a,ty})), \text{rdp2}(\text{ln}), \text{Nulloutp2} \rangle$
3. $\text{succ}(\text{max}(\text{ln})) = \text{n1} \Rightarrow$
 $\langle \text{inp2}(\text{mknewbk}(\text{i,t,s,a,ty})), \text{rdp2}(\text{ln}), \text{Nulloutp2} \rangle$
 $\quad == \text{Sendp2}(\text{mkbook}(\text{mkbkid}(\text{i}, \text{n1}), \text{t}, \text{s}, \text{a}, \text{ty}, \text{Available})) == \rangle$
 $\langle \text{inp2}(\text{mknewbk}(\text{i}, \text{t}, \text{s}, \text{a}, \text{ty})), \text{rdp2}(\text{ln}),$
 $\quad \text{outp2}(\text{mkbook}(\text{mkbkid}(\text{i}, \text{n1}), \text{t}, \text{s}, \text{a}, \text{ty}, \text{Available})) \rangle$
4. $\langle \text{inp2}(\text{nbk}), \text{rdp2}(\text{ln}), \text{outp2}(\text{bkrec}) \rangle$
 $\quad == \text{Terminatep2} == \rangle$
 $\langle \text{Nullinp2}, \text{Nullrdp2}, \text{Nulloutp2} \rangle$

The ASTS for the action A3

The action A3 consists of the processes `CheckReturnBook` (P3), and `ReturnUpdate` (P4). The states and labels of these processes are specified in the same manner as the states and labels of processes in previous examples, and thus are

not explicitly characterized in what follows. Such characterization should be obvious from their use in the specification of the processes' ASTSs.

The auxiliary functions needed to express the characterization of the processes' transition systems are characterized by the RS `BorrBookFns`. The functions characterized by this RS are:

- `getbk: book_id, list(borrower_book_detail) → borrower_book_detail` - returns an object of sort `borrower_book_detail` in the list argument with key matching the `book_id` argument.
- `gettime: borrower_book_detail → time` - returns the time attribute of a `borrower_book_detail` object.
- `deletebk: book_id, list(borrower_book_detail) → list(borrower_book_detail)` - deletes the `borrower_book_detail` object with the key given by the `book_id` argument from the list argument.

To simplify the presentation, `BorrBookFns` is not given here. The ASTSs for the processes follow:

`CheckReturnBook_TS ≡ P3state + P3label +`

Signature

transition relation

`-->_: p3state, p3label, p3state`

Laws \forall `borrid:borrower_id; bid:book_id; bind:borrower_indicator`

1. `<Nullinp3, Nullrdp3, Nulloutp3>`
`-Receivep3(bid)->`
`<inp3(bid), Nullrdp3, Nulloutp3>`
2. `<inp3(bid), Nullrdp3, Nulloutp3>`
`-Readp3(bid, bind)->`
`<inp3(bid), rdp3(bind), Nulloutp3>`
3. `<inp3(bid), rdp3(Available), Nulloutp3>`
`-Sendp3(Retin)->`
`<inp3(bid), rdp3(bind), outp3(Retin)>`
4. `<inp3(bid), rdp3(mkbind(borrid)), Nulloutp3>`
`-Sendp3(mkvetret(bid, borrid))->`
`<inp3(bid), rdp3(mkbind(borrid)), outp3(mkvetret(bid, borrid))>`
5. `<inp3(bid), Nullrdp3, Nulloutp3>`
`-Erreadp3(bid)->`
`<inp3(bid), errdp3, Nulloutp3>`
6. `<inp3(bid), errdp3, Nulloutp3>`
`-Sendp3(Retnotinfile)->`
`<inp3(bid), errdp3, outp3(Retnotinfile)>`

ReturnUpdate = P4state + P4label + Retup + BorrBookFns +

Signature

transition relation

$_ _ \rightarrow _ _$: p4state, p4label, p4state

Laws \forall borrid:borrower_id; t,t':time; bid:book_id; o3:out3p4;

o2:out2p4; vetbk:vetted_return_book;

lb,lb':list(borrower_book_detail);

1. $\langle \text{inp4}, \text{Nulltimep4}, \text{Nullrdp4}, \text{Nullout1p4}, \text{Nullout2p4}, \text{Nullout3p4} \rangle$
 $\text{-Receivep4(vetbk)-}$
 $\langle \text{inp4(vetbk)}, \text{Nulltimep4}, \text{Nullrdp4}, \text{Nullout1p4}, \text{Nullout2p4}, \text{Nullout3p4} \rangle$
2. $\langle \text{inp4(Retnotinfile)}, \text{Nulltimep4}, \text{Nullrdp4}, \text{Nullout1p4}, \text{Nullout2p4}, \text{Nullout3p4} \rangle$
 $\text{-Send1p4(Retnotin)-}$
 $\langle \text{inp4(Retnotinfile)}, \text{Nulltimep4}, \text{Nullrdp4}, \text{out1p4(Retnotin)},$
 $\text{Nullout2p4}, \text{Nullout3p4} \rangle$
3. $\langle \text{inp4(Retin)}, \text{Nulltimep4}, \text{Nullrdp4}, \text{Nullout1p4}, \text{Nullout2p4}, \text{Nullout3p4} \rangle$
 $\text{-Send1p4(Alreadyin)-}$
 $\langle \text{inp4(Retin)}, \text{Nulltimep4}, \text{Nullrdp4}, \text{out1p4(Alreadyin)}, \text{Nullout2p4}, \text{Nullout3p4} \rangle$
4. $\langle \text{inp4(mkvetret(bid, borrid))}, \text{Nulltimep4}, \text{Nullrdp4},$
 $\text{Nullout1p4}, \text{Nullout2p4}, \text{Nullout3p4} \rangle$
 $\text{-Readp4(borrid,lb)-}$
 $\langle \text{inp4(mkvetret(bid, borrid))}, \text{Nulltimep4}, \text{rdp4(lb)}, \text{Nullout1p4},$
 $\text{Nullout2p4}, \text{Nullout3p4} \rangle$
5. $\langle \text{inp4(mkvetret(bid, borrid))}, \text{Nulltimep4}, \text{rdp4(lb)}, \text{Nullout1p4},$
 $\text{Nullout2p4}, \text{Nullout3p4} \rangle$
 -Timep4(t)-
 $\langle \text{inp4(mkvetret(bid, borrid))}, \text{timep4(t)}, \text{rdp4(lb)}, \text{Nullout1p4},$
 $\text{Nullout2p4}, \text{Nullout3p4} \rangle$
6. $\sim(\text{gettime}(\text{getbk}(\text{bid}, \text{lb})) > t), \text{deletebk}(\text{bid}, \text{lb}) = \text{lb}' \Rightarrow$
 $\langle \text{inp4(mkvetret(bid, borrid))}, \text{timep4(t)}, \text{rdp4(lb)}, \text{Nullout1p4}, \text{Nullout2p4}, \text{o3} \rangle$
 $\text{-Send2p4(bid, lb')-}$
 $\langle \text{inp4(mkvetret(bid, borrid))}, \text{timep4(t)}, \text{rdp4(lb)}, \text{Nullout1p4}, \text{out2p4(lb')}, \text{o3} \rangle$
7. $\text{gettime}(\text{getbk}(\text{bid}, \text{lb})) = t', t' > t \Rightarrow$
 $\langle \text{inp4(mkvetret(bid, borrid))}, \text{timep4(t)}, \text{rdp4(lb)}, \text{Nullout1p4}, \text{Nullout2p4}, \text{o3} \rangle$
 $\text{-Send2p4(bid, mkbdet(bid, t', t)|deletebk(bid, lb))-}$
 $\langle \text{inp4(mkvetret(bid, borrid))}, \text{timep4(t)}, \text{rdp4(lb)}, \text{Nullout1p4},$
 $\text{mkbdet(bid, t', t)|deletebk(bid, lb)}, \text{o3} \rangle$

8. $\langle \text{inp4}(\text{mkvetret}(\text{bid}, \text{borrid})), \text{int}, \text{rdp4}(\text{lb}), \text{Nullout1p4}, \text{o2}, \text{Nullout3p4} \rangle$
 $\text{-Send3p4}(\text{bid}, \text{Available}) \text{-}$
 $\langle \text{inp4}(\text{mkvetret}(\text{bid}, \text{borrid})), \text{int}, \text{rdp4}(\text{lb}), \text{Nullout1p4}, \text{o2}, \text{out3p4}(\text{Available}) \rangle$

The ASTS for A3 can now be given. The characterization of the action's states and labels should be obvious from their use in the ASTS below:

$A3_TS \equiv A3\text{state} + A3\text{label} +$

Signature

transition relation

$_ == _ == _ : a3\text{state}, a3\text{label}, a3\text{state}$

Laws $\forall p3, p3': p3\text{state}; p4, p4': p4\text{state}; \text{vbk}: \text{vetted_return_book};$

$\text{rmess}: \text{return_message}; \text{rubr}: \text{ret_updated_borr};$

$\text{rubk}: \text{ret_updated_book};$

$\text{bind}: \text{book_indicator}; \text{bid}: \text{book_id}; \text{t}: \text{time}; \text{borrid}: \text{borrower_id};$

$\text{lb}: \text{list}(\text{borrower_book_detail})$

1. $p3\text{-Sendp3}(\text{vbk}) \text{-} p3', p4\text{-Receivep4}(\text{vetbk}) \text{-} p4' \Rightarrow$
 $\langle p3, p4 \rangle == \text{SYNCH}(\text{Sendp3}(\text{vbk}), \text{Receivep4}(\text{vetbk})) == \langle p3', p4' \rangle$
 --- synchronized communication between P3 and P4 via the data flow
 vetted_return_book ---
2. $p3\text{-Receivep3}(\text{bid}) \text{-} p3' \Rightarrow \langle p3, p4 \rangle == \text{Receivep3}(\text{bid}) == \langle p3', p4 \rangle$
 --- an input event of the action ---
3. $p3\text{-Readp3}(\text{bid}, \text{bind}) \text{-} p3' \Rightarrow \langle p3, p4 \rangle == \text{Readp3}(\text{bid}, \text{bind}) == \langle p3', p4 \rangle$
 --- a successful read by the action on BOOK ---
4. $p3\text{-Erreadp3}(\text{bid}) \text{-} p3' \Rightarrow \langle p3, p4 \rangle == \text{Erreadp3}(\text{bid}) == \langle p3', p4 \rangle$
 --- an unsuccessful read by the action on BOOK ---
5. $p4\text{-Send1p4}(\text{rmess}) \text{-} p4' \Rightarrow \langle p3, p4 \rangle == \text{Send1p4}(\text{rmess}) == \langle p3, p4' \rangle$
 --- an output event of the action ---
6. $p4\text{-Send2p4}(\text{rubr}) \text{-} p4' \Rightarrow \langle p3, p4 \rangle == \text{Send2p4}(\text{rubr}) == \langle p3, p4' \rangle$
 --- an output event of the action ---
7. $p4\text{-Send3p4}(\text{rubk}) \text{-} p4' \Rightarrow \langle p3, p4 \rangle == \text{Send3p4}(\text{rubk}) == \langle p3, p4' \rangle$
 --- an output event of the action ---
8. $p4\text{-Readp4}(\text{borrid}, \text{lb}) \text{-} p4' \Rightarrow \langle p3, p4 \rangle == \text{Readp4}(\text{borrid}, \text{lb}) == \langle p3, p4' \rangle$
 --- a successful read by the action on BORROWER ---
9. $p4\text{-Erreadp4}(\text{borrid}) \text{-} p4' \Rightarrow \langle p3, p4 \rangle == \text{Erreadp4}(\text{borrid}) == \langle p3, p4' \rangle$
 --- an unsuccessful read by the action on BORROWER ---
10. $p4\text{-Timep4}(\text{t}) \text{-} p4' \Rightarrow \langle p3, p4 \rangle == \text{Timep4}(\text{t}) == \langle p3, p4' \rangle$
 --- a state read on return_time by the action ---

The ASTS for the action A4

The action A4 consists of the processes `CheckBook` (P5), `GetOverdueBooks` (P6), `CalculateFine` (P7), `VettBorrower` (P8), and `CheckoutUpdate` (P9). The states and labels of these processes are specified in the same manner as the states and labels of processes in previous examples, and thus are not explicitly characterized in what follows. Such characterization should be obvious from their use in the specification of the processes' ASTSs.

The ASTSs for the processes follow:

`CheckBook` \equiv P5state + P5label +

Signature**transition relation**

$_{-} _ \rightarrow _ : p5state, p5label, p5state$

Laws \forall `outbk:out_book`; `ty:copy_type`; `bid:book_id`

1. `<Nullinp5, Nullrdp5, Nulloutp5>`
`-Receivep5(bid)->`
`<inp5(bid), Nullrdp5, Nulloutp5>`
2. `<inp5(bid), Nullrdp5, Nulloutp5>`
`-Readp5(bid, outbk)->`
`<inp5(bid), rp5(outbk), Nulloutp5>`
3. `ty \neq Ref` \Rightarrow
`<inp5(bid), rp5(mkoutbk(Available, ty)), Nulloutp5>`
`-Sendp5(mkvbk(bid, ty))->`
`<inp5(bid), rp5(mkoutbk(Available, ty)), outp5(mkvbk(bid, ty))>`
4. `<inp5(bid), rp5(mkoutbk(Available, Ref)), Nulloutp5>`
`-Sendp5(Notborr)->`
`<inp5(bid), rp5(mkoutbk(Available, Ref)), outp5(Notborr)>`
5. `inp5(bid), rp5(mkoutbk(mkbind(borrid), ty), Nulloutp5>`
`-Sendp5(CheckedOut)->`
`<inp5(bid), rp5(mkoutbk(mkbind(borrid), ty), outp5(CheckedOut)>`
6. `<inp5(bid), Nullrdp5, Nulloutp5>`
`-Erreadp5(bid)->`
`<inp5(bid), errp5, Nulloutp5>`
7. `<inp5(bid), errp5, Nulloutp5>`
`-Sendp5(Bknotinfile)->`
`<inp5(bid), errp5, outp5(Bknotinfile)>`

The ASTS characterizing the transition system for P6 utilizes the RS FinesRec which characterizes the function

getfinesrec: list(borrower_book_detail), time \rightarrow list(number)

which derives a list of fines given a list of borrower_book_detail objects and the current time.

GetOverdueBooks \equiv P6state + P6label + FinesRec +

Signature

transition relation

$_ _ \rightarrow _ _$: p6state, p6label, p6state

Laws \forall borrid:borrider_id; outb:out_borrower; t:time; o1:o1p6;

o2:o2p6; lb:list(borrower_book_detail); bt:borrider_type; n:number;

In:list(integer)

1. \langle Nullinp6, Nullrdp6, Nulltimep6, Nullout1p6, Nullout2p6 \rangle
 -Receivep6(borrid)- \rightarrow
 \langle inp6(borrid), Nullrdp6, Nulltimep6, Nullout1p6, Nullout2p6 \rangle
2. \langle inp6(borrid), Nullrdp6, Nulltimep6, Nullout1p6, Nullout2p6 \rangle
 -Readp6(borrid, outb)- \rightarrow
 \langle inp6(borrid), rp6(outb), Nulltimep6, Nullout1p6, Nullout2p6 \rangle
3. \langle inp6(borrid), rp6(outb), Nulltimep6, Nullout1p6, Nullout2p6 \rangle
 Timep6(t)- \rightarrow
 \langle inp6(borrid), rp6(outb), timep6(t), Nullout1p6, Nullout2p6 \rangle
4. getfinesrec(lb, t) = ln \Rightarrow
 \langle inp6(borrid), rp6(mkoutbr(lb, bt, n)), timep6(t), Nullout1p6, o2 \rangle
 -Send1p6(ln)- \rightarrow
 \langle inp6(borrid), rp6(mkoutb(lb, bt, n)), timep6(t), out1p6(ln), o2 \rangle
5. \langle inp6(borrid), rp6(outb), timep6(t), o1, Nullout2p6 \rangle
 -Send2p6(mkbflag(borrid,outb))- \rightarrow
 \langle inp6(borrid), rp6(outb), timep6(t), o1, out2p6(mkbflag(borrid,outb)) \rangle
6. \langle inp6(borrid), Nullrdp6, Nulltimep6, Nullout1p6, Nullout2p6 \rangle
 -Erreadp6(borrid)- \rightarrow
 \langle inp6(borrid), errp6, Nulltimep6, Nullout1p6, Nullout2p6 \rangle
7. \langle inp6(borrid), errp6, Nulltimep6, Nullout1p6, Nullout2p6 \rangle
 -Send2p6(Bflag)- \rightarrow
 \langle inp6(borrid), errp6, Nulltimep6, Nullout1p6, out2p6(Bflag) \rangle

P7 utilizes an RS, SumList, which characterizes the auxiliary function

sum: list(integer) \rightarrow integer

which returns the sum of a list of integers.

GetOverdueBooks \equiv P7state + P7label + SumList +

Signature

transition relation

$_ _ \rightarrow _$: p7state, p7label, p7state

Laws \forall In:list(integer); n:integer

1. \langle Nullinp7, Nulloutp7 \rangle
 -Receivep7(In)-
 \langle inp7(In), Nulloutp7 \rangle
2. $\text{sum(In) = n} \Rightarrow$
 \langle inp7(In), Nulloutp7 \rangle
 -Sendp7(n)-
 \langle inp7(In), outp7(n) \rangle

VettBorrower \equiv P8state + P8label +

Signature

transition relation

$_ _ \rightarrow _$: p8state, p8label, p8state

Laws \forall n,n',f:number; t:time; borrid:borrower_id; in1:i1p8; in2:i2p8;

lb:list(borrower_book_detail); bt:copy_type; bflag:borr_flag

1. \langle Nullin1p8, in2, Nulloutp8 \rangle
 -Receive1p8(n)-
 \langle in1p8(n), in2, Nulloutp8 \rangle
2. \langle in1, Nullin2p8, Nulloutp8 \rangle
 $\text{-Receive2p8(bflag)-}$
 \langle in1, in2p8(bflag), Nulloutp8 \rangle
3. \langle in1, in2p8(Bflag), Nulloutp8 \rangle
 -Sendp8(NoBorr)-
 \langle in1, in2p8(Bflag), outp8(NoBorr) \rangle
4. $n - n' \leq \text{Limit} \Rightarrow$
 \langle in1p8(n), in2p8(mkbflag(borrid, mkoutb(lb, bt, n'))), Nulloutp8 \rangle
 $\text{-Sendp8(mkvetborr(mkoutb(lb, bt, n'), borrid))-}$
 \langle in1p8(n), in2p8(mkbflag(borrid, mkoutb(lb, bt, n'))),
 $\text{outp8(mkvetborr(mkoutb(lb, bt, n'), borrid))}\rangle$
5. $n - n' = f, f > \text{Limit} \Rightarrow$
 \langle in1p8(n), in2p8(mkbflag(borrid, mkoutb(lb, bt, n'))), Nulloutp8 \rangle
 $\text{-Sendp8(mkerrvborr(f))-}$
 \langle in1p8(n), in2p8(mkbflag(borrid, mkoutb(lb, bt, n'))), outp8(mkerrvborr(f)) \rangle

The function `week` used in the ASTS for P9, given below, is assumed to be characterized by the RS Time characterizing time, where the basic unit of time is a day. The function takes a day, `t`, and a number, `n`, and returns the day `n` weeks (a week is seven days) in the future from `t`.

CheckoutUpdate = P9state + P9label +

Signature

transition relation

`_ ->_ : p9state, p9label, p9state`

Laws \forall `vbk:vetted_book`; `vbr:vetted_borr`; `in1:i1p9`; `in2:i2p9`;

`time:tp9`; `out1:o1p9`; `out2:o2p9`; `out3:o3p9`

1. `<Nullin1p9, in2, NullTimep9, Nullout1p9, Nullout2p9, Nullout3p9>`
`-Receive1p9(vbk)->`
`<in1p9(vbk), in2, NullTimep9, Nullout1p9, Nullout2p9, Nullout3p9>`
2. `<in1, Nullin2p9, NullTimep9, Nullout1p9, Nullout2p9, Nullout3p9>`
`-Receive2p9(vbr)->`
`<in1, in2p9(vbr), NullTimep9, Nullout1p9, Nullout2p9, Nullout3p9>`
3. `<in1p9(vbk), in2p9(vbr), time, Nullout1p9, out2, out3>`
`-Send1p9(mkoutmess(vbr, vbk))->`
`<in1p9(vbk), in2p9(vbr), time, out1p9(mkoutmess(vbr, vbk)), out2, out3>`
4. `<in1p9(mkvbk(bid, ty)), in2p9(mkvetborr(ob, borrid)),`
`NullTimep9, out1, Nullout2p9, Nullout3p9>`
`-Timep9(t)->`
`<in1p9(mkvbk(bid, ty)), in2p9(mkvetborr(ob, borrid)),`
`timep9(t), out1, Nullout2p9, Nullout3p9>`
5. `week(t, 2) = t' \Rightarrow`
`<in1p9(mkvbk(bid, Book)),`
`in2p9(mkvetborr(mkoutb(lb, Undergrad, n), borrid)),`
`timep9(t), out1, Nullout2p9, Nullout3p9>`
`-Send2p9(borrid, (mkbdet(bid, t', Notret))|lb)->`
`<in1p9(mkvbk(bid, Book)),`
`in2p9(mkvetborr(mkoutb(lb, Undergrad, n), borrid)),`
`timep9(t), out1, out2p9(mkbdet(bid, t', Notret))|lb, Nullout3p9>`
6. `week(t, 4) = t' \Rightarrow`
`<in1p9(mkvbk(bid, Book)),`
`in2p9(mkvetborr(mkoutb(lb, Postgrad, n), borrid)),`
`timep9(t), out1, Nullout2p9, Nullout3p9>`
`-Send2p9(borrid, (mkbdet(bid, t', Notret))|lb)->`
`<in1p9(mkvbk(bid, Book)),`

- ```

in2p9(mkvetborr(mkoutb(lb, Postgrad, n), borrid)),
timep9(t), out1, out2p9(mkbdet(bid, t', Notret))|lb), Nullout3p9>
7. week(t, 6) = t' ⇒
<in1p9(mkvbk(bid, Book)), in2p9(mkvetborr(mkoutb(lb, Staff, n), borrid)),
timep9(t), out1, Nullout2p9, Nullout3p9>
-Send2p9(borrid, (mkbdet(bid, week(6,t), Notret))|lb)->
<in1p9(mkvbk(bid, Book)), in2p9(mkvetborr(mkoutb(lb, Staff, n), borrid)),
timep9(t), out1, out2p9(mkbdet(bid, t', Notret))|lb), Nullout3p9>
8. week(t, 2) = t' ⇒
<in1p9(mkvbk(bid, Per)), in2p9(mkvetborr(mkoutb(lb, Postgrad, n), borrid)),
timep9(t), out1, Nullout2p9, Nullout3p9>
-Send2p9(borrid, (mkbdet(bid, t', Notret))|lb)->
<in1p9(mkvbk(bid, Per)), in2p9(mkvetborr(mkoutb(lb, Postgrad, n), borrid)),
timep9(t), out1, out2p9(mkbdet(bid, t', Notret))|lb), Nullout3p9>
9. week(t, 4) = t' ⇒
<in1p9(mkvbk(bid, Per)), in2p9(mkvetborr(mkoutb(lb, Staff, n), borrid)),
timep9(t), out1, Nullout2p9, Nullout3p9>
-Send2p9(borrid, (mkbdet(bid, t', Notret))|lb)->
<in1p9(mkvbk(bid, Per)),
in2p9(mkvetborr(mkoutb(lb, Staff, n), borrid)), timep9(t),
out1, out2p9(mkbdet(bid, t', Notret))|lb), Nullout3p9>
10. <in1p9(mkvbk(bid, ty)), in2p9(mkvetborr(ob, borrid)), t9, out1,
out2p9(borrid, lb), Nullout3p9>
-Send3p9(bid, mkbind(borrid))->
<in1p9(mkvbk(bid, ty)), in2p9(mkvetborr(ob, borrid)), t9, out1,
out2p9(borrid, lb), out3p9(mkbind(borrid))>

```

The ASTS for the action A4 follows:

A4\_TS ≡ A4state + A4label +

**Signature**

**transition relation**

**\_==\_==>\_** : a4state, a4label, a4state

Laws  $\forall$  bid:book\_id; borrid:borrower\_id; t:time; vbk:vetted\_book;  
 p1,p1':statep1;...; p5,p5':statep5; A1,A2:a4label;  
 vbr:vetted\_borrower; ln:list(number); f:number; bflag:borr\_flag;  
 obk:out\_book; obr:out\_borrower; upbk:out\_updated\_book;  
 upbr:out\_updated\_borr; mess:checkout\_message

Synchronized Events: process/process communication (via synchronized data flows)

1. p5--Receivep5(bid)-->p5', p6--Receive1p6(borrid)-->p6'  $\Rightarrow$   
 <p5, p6, p7, p8, p9>  
 ==SYNCH((Receivep5(bid), Receive1p2(borrid)))==>  
 <p5', p6', p7, p8, p9>
2. p5--Sendp5(vbk)-->p5', p9--Receive1p9(vbk)-->p9'  $\Rightarrow$   
 <p5, p6, p7, p8, p9>  
 ==SYNCH((Sendp5(vbk), Receive1p9(vbk)))==>  
 <p5', p6, p7, p8, p9'>
3. p6--Send1p6(ln)-->p6', p7--Receivep7(ln)-->p7'  $\Rightarrow$   
 <p5, p6, p7, p8, p9>  
 ==SYNCH((Send1p6(ln), Receivep7(ln)))==>  
 <p5, p6', p7', p8, p9>
4. p6--Send2p6(bflag)-->p6', p8--Receive2p8(bflag)-->p8'  $\Rightarrow$   
 <p5, p6, p7, p8, p9>  
 ==SYNCH((Send2p6(bflag), Receive2p8(bflag)))==>  
 <p5, p6', p7, p8', p9>
5. p7--Sendp7(f)-->p7', p8--Receive1p8(f)-->p8'  $\Rightarrow$   
 <p5, p6, p7, p8, p9>  
 ==SYNCH((Sendp7(f), Receive1p8(f)))==>  
 <p5, p6, p7', p8', p9>
6. p8--Sendp8(vbr)-->p8', p9--Receive2p9(vbr)-->p9'  $\Rightarrow$   
 <p5, p6, p7, p8, p9>  
 ==SYNCH((Sendp8(vb), Receive2p8(vbr)))==>  
 <p5, p6, p7, p8', p9'>

Single Events: input and output (including read/write) events of the action

7. p5--Readp1(bid, obk)-->p5'  $\Rightarrow$   
 <p5, p6, p7, p8, p9>==Readp1(bid, obk)==><p5', p6, p7, p8, p9>
8. p5--Erreadp1(bid)-->p5'  $\Rightarrow$   
 <p5, p6, p7, p8, p9>==Erreadp1(bid)==><p5', p6, p7, p8, p9>
9. p6--Receive2p2(t)-->p6'  $\Rightarrow$   
 <p5, p6, p7, p8, p9>==Receive2p2(t)==><p5, p6', p7, p8, p9>

10.  $p6 \rightarrow \text{Readp2}(\text{borrid}, \text{obr}) \rightarrow p6' \Rightarrow$   
 $\langle p5, p6, p7, p8, p9 \rangle \Rightarrow \text{Readp2}(\text{borrid}, \text{obr}) \Rightarrow \langle p5, p6', p7, p8, p9 \rangle$
11.  $p6 \rightarrow \text{Erreadp2}(\text{borrid}) \rightarrow p6' \Rightarrow$   
 $\langle p5, p6, p7, p8, p9 \rangle \Rightarrow \text{Erreadp2}(\text{borrid}) \Rightarrow \langle p5, p6', p7, p8, p9 \rangle$
12.  $p9 \rightarrow \text{Update1}(\text{bid}, \text{upbk}) \rightarrow p9' \Rightarrow$   
 $\langle p5, p6, p7, p8, p9 \rangle \Rightarrow \text{Update1}(\text{bid}, \text{upbk}) \Rightarrow \langle p5, p6, p7, p8, p9' \rangle$
13.  $p9 \rightarrow \text{Update2}(\text{borrid}, \text{upbr}) \rightarrow p9' \Rightarrow$   
 $\langle p5, p6, p7, p8, p9 \rangle \Rightarrow \text{Update2}(\text{borrid}, \text{upbr}) \Rightarrow \langle p5, p6, p7, p8, p9' \rangle$
14.  $p9 \rightarrow \text{Send}(\text{mess}) \rightarrow p9' \Rightarrow$   
 $\langle p5, p6, p7, p8, p9 \rangle \Rightarrow \text{Send}(\text{mess}) \Rightarrow \langle p5, p6, p7, p8, p9' \rangle$
15.  $p9 \rightarrow \text{Receive3p5}(t) \rightarrow p9' \Rightarrow$   
 $\langle p5, p6, p7, p8, p9 \rangle \Rightarrow \text{Receive3p5}(t) \Rightarrow \langle p5, p6, p7, p8, p9' \rangle$

### Parallel Events

--- Events which affect separate parts of an action can be carried out in parallel ---

16.  $\langle p5, p6, p7, p8, p9 \rangle \Rightarrow A1 \Rightarrow \langle p5', p6, p7, p8, p9 \rangle,$   
 $\langle p5, p6, p7, p8, p9 \rangle \Rightarrow A2 \Rightarrow \langle p5, p6', p7', p8', p9' \rangle \Rightarrow$   
 $\langle p5, p6, p7, p8, p9 \rangle \Rightarrow \text{PAR}(A1, A2) \Rightarrow \langle p5', p6', p7', p8', p9' \rangle$
17.  $\langle p5, p6, p7, p8, p9 \rangle \Rightarrow A1 \Rightarrow \langle p5', p6', p7, p8, p9 \rangle,$   
 $\langle p5, p6, p7, p8, p9 \rangle \Rightarrow A2 \Rightarrow \langle p5, p6, p7', p8', p9' \rangle \Rightarrow$   
 $\langle p5, p6, p7, p8, p9 \rangle \Rightarrow \text{PAR}(A1, A2) \Rightarrow \langle p5', p6', p7', p8', p9' \rangle$
18.  $\langle p5, p6, p7, p8, p9 \rangle \Rightarrow A1 \Rightarrow \langle p5', p6', p7', p8, p9 \rangle,$   
 $\langle p5, p6, p7, p8, p9 \rangle \Rightarrow A2 \Rightarrow \langle p5, p6, p7, p8', p9' \rangle \Rightarrow$   
 $\langle p5, p6, p7, p8, p9 \rangle \Rightarrow \text{PAR}(A1, A2) \Rightarrow \langle p5', p6', p7', p8', p9' \rangle$
19.  $\langle p5, p6, p7, p8, p9 \rangle \Rightarrow A1 \Rightarrow \langle p5', p6', p7', p8', p9 \rangle,$   
 $\langle p5, p6, p7, p8, p9 \rangle \Rightarrow A2 \Rightarrow \langle p5, p6, p7, p8, p9' \rangle \Rightarrow$   
 $\langle p5, p6, p7, p8, p9 \rangle \Rightarrow \text{PAR}(A1, A2) \Rightarrow \langle p5', p6', p7', p8', p9' \rangle$

### Termination event

---  $\text{Nullp}_i, 5 \leq i \leq 9$ , is the abbreviated form for the idle state of  $P_i$  ---

20.  $\langle p5, p6, p7, p8, \text{in1p9}(\text{vbk}), \text{in2p9}(\text{vbr}), \text{timep9}(t),$   
 $\text{out1p9}(\text{omess}), \text{out2p9}(\text{ubr}), \text{out3p9}(\text{ubk}) \rangle \Rightarrow$   
 $\Rightarrow \text{Terminatea4} \Rightarrow$   
 $\langle \text{Nullp}_5, \text{Nullp}_6, \text{Nullp}_7, \text{Nullp}_8, \text{Nullp}_9 \rangle$
21.  $\langle p5, p6, p7, p8, \text{in1p9}(\text{Notborr}), \text{in2}, \text{NullTimep9}, \text{out1p9}(\text{omess}),$   
 $\text{Nullout2p9}, \text{Nullout3p9} \rangle \Rightarrow$   
 $\Rightarrow \text{Terminatea4} \Rightarrow$   
 $\langle \text{Nullp}_5, \text{Nullp}_6, \text{Nullp}_7, \text{Nullp}_8, \text{Nullp}_9 \rangle$



22. <p5, p6, p7, p8, <in1p9(CheckedOut), in2, NullTimep9, out1p9(omess),  
Nullout2p9, Nullout3p9>>  
==Terminatea4==>  
<Nullp5, Nullp6, Nullp7, Nullp8, Nullp9>
23. <p5, p6, p7, p8, <in1p9(Bknotinfile), in2, NullTimep9, out1p9(omess),  
Nullout2p9, Nullout3p9>>  
==Terminatea4==>  
<Nullp5, Nullp6, Nullp7, Nullp8, Nullp9>
24. <p5, p6, p7, p8, <in1, in2p9(NoBorr), NullTimep9, out1p9(omess),  
Nullout2p9, Nullout3p9>>  
==Terminatea4==>  
<Nullp5, Nullp6, Nullp7, Nullp8, Nullp9>
25. <p5, p6, p7, p8, <in1, in2p9(mkerrvborr(f)), NullTimep9, out1p9(omess),  
Nullout2p9, Nullout3p9>>  
==Terminatea4==>  
<Nullp5, Nullp6, Nullp7, Nullp8, Nullp9>
26. <p5, p6, p7, p8, <in1p9(mkvbk(bid, Per)),  
in2p9(mkvetborr(mkoutb(lb, Undergrad, n), borrid)), NullTimep9,  
out1p9(omess), Nullout2p9, Nullout3p9>>  
==Terminatea4==>  
<Nullp5, Nullp6, Nullp7, Nullp8, Nullp9>

### The ASTS for the action A5

The action A5 consists of the processes `GenerateFinesRecord` (P10), and `UpdateBorrRecord` (P11). As in the specification of the ASTS for A4 given above, the states and labels of these processes are not explicitly characterized in what follows, as such characterization should be obvious from their use in the specification of the processes' ASTSs.

The ASTSs for the processes follow:

`GenerateFinesRecord` = P10state + P10label + FinesRec + SumList +

#### **Signature**

#### **transition relation**

`_->_`: p10state, p10label, p10state

**Laws**  $\forall$  borrid:borrower\_id; t:time; lb:list(borrower\_book\_detail);

**bd:borr\_detail; n,n':number**

1. <Nullinp10, Nullrdp10, NullTimep10, Nulloutp10>

-Receivep10(borrid)->

<inp10(borrid), Nullrdp10, NullTimep10, Nulloutp10>

2.  $\langle \text{inp10}(\text{borrid}), \text{Nullrdp10}, \text{NullTimep10}, \text{Nulloutp10} \rangle$   
 $\text{-Readp10}(\text{borrid}, \text{bd})\text{-}$   
 $\langle \text{inp10}(\text{borrid}), \text{rdp10}(\text{bd}), \text{NullTimep10}, \text{Nulloutp10} \rangle$
3.  $\langle \text{inp10}(\text{borrid}), \text{rdp10}(\text{bd}), \text{NullTimep10}, \text{Nulloutp10} \rangle$   
 $\text{-Timep10}(t)\text{-}$   
 $\langle \text{inp10}(\text{borrid}), \text{rdp10}(\text{bd}), \text{timep10}(t), \text{Nulloutp10} \rangle$
4.  $\text{sum}(\text{getfinesrec}(\text{lb}, t))\text{-}n = n' \Rightarrow$   
 $\langle \text{inp10}(\text{borrid}), \text{rdp10}(\text{mkborrdet}(\text{lb}, n)), \text{timep10}(t), \text{Nulloutp10} \rangle$   
 $\text{-Sendp10}(\text{mkbfrec}(n', \text{borrid}))\text{-}$   
 $\langle \text{inp10}(\text{borrid}), \text{rdp10}(\text{mkborrdet}(\text{lb}, n)), \text{timep10}(t), \text{outp10}(n', \text{borrid}) \rangle$
5.  $\langle \text{inp10}(\text{borrid}), \text{Nullrdp10}, \text{NullTimep10}, \text{Nulloutp10} \rangle$   
 $\text{-Erreadp10}(\text{borrid})\text{-}$   
 $\langle \text{inp10}(\text{borrid}), \text{errdp10}, \text{NullTimep10}, \text{Nulloutp10} \rangle$
6.  $\langle \text{inp10}(\text{borrid}), \text{errdp10}, \text{NullTimep10}, \text{Nulloutp10} \rangle$   
 $\text{-Sendp10}(\text{NoRec})\text{-}$   
 $\langle \text{inp10}(\text{borrid}), \text{errdp10}, \text{NullTimep10}, \text{outp10}(\text{NoRec}) \rangle$

UpdateBorrRecord  $\equiv$  P11state + P11label +

### Signature

transition relation

$\_ \_ \text{-} \_ \_ \text{-}$ : p11state, p11label, p11state

**Laws**  $\forall$  borrid:borrower\_id; t:time; n,n',f:number;

**frec:borr\_fine\_record; in1:i1p11; in2:i2p11; out1:o1p11; out2:o2p11**

1.  $\langle \text{Nullin1p11}, \text{in2}, \text{Nullout1p11}, \text{Nullout2p11} \rangle$   
 $\text{-Receive1p11}(n)\text{-}$   
 $\langle \text{in1p11}(n), \text{in2}, \text{Nullout1p11}, \text{Nullout2p11} \rangle$
2.  $\langle \text{in1}, \text{Nullin2p11}, \text{Nullout1p11}, \text{Nullout2p11} \rangle$   
 $\text{-Receive2p11}(\text{frec})\text{-}$   
 $\langle \text{in1}, \text{in2p11}(\text{frec}), \text{Nullout1p11}, \text{Nullout2p11} \rangle$
3.  $\langle \text{in1}, \text{in2p11}(\text{NoRec}), \text{Nullout1p11}, \text{Nullout2p11} \rangle$   
 $\text{-Send2p11}(\text{Norec})\text{-}$   
 $\langle \text{in1}, \text{in2p11}(\text{NoRec}), \text{Nullout1p11}, \text{out2p11}(\text{Norec}) \rangle$
4.  $\langle \text{in1p11}(n), \text{in2p11}(\text{mkbfrec}(0, \text{borrid})), \text{Nullout1p11}, \text{Nullout2p11} \rangle$   
 $\text{-Send2p11}(\text{Nofines})\text{-}$   
 $\langle \text{in1p11}(n), \text{in2p11}(\text{mkbfrec}(0, \text{borrid})), \text{Nullout1p11}, \text{out2p11}(\text{Nofines}) \rangle$

5.  $n' > n, n' - n = f \Rightarrow$

$\langle \text{in1p11}(n), \text{in2p11}(\text{mkbfreq}(n', \text{borrid})), \text{out1}, \text{Nullout2p11} \rangle$

$-\text{Send2p11}(f) \rightarrow$

$\langle \text{in1p11}(n), \text{in2p11}(\text{mkbfreq}(n', \text{borrid})), \text{out1}, \text{out2p11}(f) \rangle$

6.  $n' < n, n - n' = f \Rightarrow$

$\langle \text{in1p11}(n), \text{in2p11}(\text{mkbfreq}(n', \text{borrid})), \text{out1}, \text{Nullout2p11} \rangle$

$-\text{Send2p11}(f) \rightarrow$

$\langle \text{in1p11}(n), \text{in2p11}(\text{mkbfreq}(n, \text{borrid})), \text{out1}, \text{out2p11}(f) \rangle$

7.  $n' = n \Rightarrow$

$\langle \text{in1p11}(n), \text{in2p11}(\text{mkbfreq}(n', \text{borrid})), \text{out1}, \text{Nullout2p11} \rangle$

$-\text{Send2p11}(\text{Cleared}) \rightarrow$

$\langle \text{in1p11}(n), \text{in2p11}(\text{mkbfreq}(n', \text{borrid})), \text{out1}, \text{out2p11}(\text{Cleared}) \rangle$

8.  $n' \neq 0, n + n' = f \Rightarrow$

$\langle \text{in1p11}(n), \text{in2p11}(\text{mkbfreq}(n', \text{borrid})), \text{Nullout1p11}, \text{out2} \rangle$

$-\text{Send1p11}(\text{borrid}, f) \rightarrow$

$\langle \text{in1p11}(n), \text{in2p11}(\text{mkbfreq}(n', \text{borrid})), \text{out1p11}(f), \text{out2} \rangle$

The ASTS for the action A5 is given below:

$A5\_TS \equiv A5state + A5label +$

#### Signature

##### transition relation

$_{-} == _ == > _ : a5state, a5label, a5state$

**Laws**  $\forall \text{borrid:borrower\_id}; t:\text{time}; n:\text{number}; \dots$

#### Synchronized events

1.  $\text{p10-Receivep10}(\text{borrid}) \rightarrow \text{p10}, \text{p11-Receive1p11}(n) \rightarrow \text{p11}' \Rightarrow$

$\langle \text{p10}, \text{p11} \rangle$

$==\text{SYNCH}(\{\text{Receivep10}(\text{borrid}), \text{Receive1p11}(n)\})==>$

$\langle \text{p10}', \text{p11}' \rangle$

--- synchronized invocation of action ---

2.  $\text{p10-Sendp10}(n) \rightarrow \text{p10}', \text{p11-Receive2p11}(n) \rightarrow \text{p11}' \Rightarrow$

$\langle \text{p10}, \text{p11} \rangle$

$==\text{SYNCH}(\{\text{Sendp10}(n), \text{Receive2p11}(n)\})==>$

$\langle \text{p10}', \text{p11}' \rangle$

--- synchronized communication via the data flow `borr_fine_record` ---

#### Single events

{All events of P10 and P11 that are not synchronized in 1 and 2}

Parallel events

{All action events that affect mutual exclusive parts of the state of A4 are allowed to occur in parallel}

The ASTS for the action DeleteBorr (P12)

DeleteBorr\_TS  $\equiv$  P12state + P12label +

**Signature****transition relation**

$\_ == \_ ==> \_ :$  p12state, p12label, p12state

**Laws**  $\forall$  borrid:borrower\_id; lb:list(borrower\_book\_detail)

1.  $\langle \text{Nullinp12}, \text{Nullrdp12}, \text{Nulloutp12} \rangle$   
 $\quad ==\text{Receivep12}(\text{borrid})==\rangle$   
 $\quad \langle \text{inp12}(\text{borrid}), \text{Nullrdp12}, \text{Nulloutp12} \rangle$
2.  $\langle \text{inp12}(\text{borrid}), \text{Nullrdp12}, \text{Nulloutp12} \rangle$   
 $\quad ==\text{Readp12}(\text{borrid}, \text{lb})==\rangle$   
 $\quad \langle \text{inp12}(\text{borrid}), \text{rdp12}(\text{lb}), \text{Nulloutp12} \rangle$
3.  $\text{lb} = \text{emptylist} \Rightarrow$   
 $\quad \langle \text{inp12}(\text{borrid}), \text{rdp12}(\text{lb}), \text{Nulloutp12} \rangle$   
 $\quad ==\text{Deletebr}(\text{borrid})==\rangle$   
 $\quad \langle \text{inp12}(\text{borrid}), \text{delbr}, \text{Nulloutp12} \rangle$
4.  $\langle \text{inp12}(\text{borrid}), \text{delbr}, \text{Nulloutp12} \rangle$   
 $\quad ==\text{Sendp12}(\text{OKdelbr})==\rangle$   
 $\quad \langle \text{inp12}(\text{borrid}), \text{delbr}, \text{outp12}(\text{OKdelbr}) \rangle$
4.  $\text{lb} \neq \text{emptylist} \Rightarrow$   
 $\quad \langle \text{inp12}(\text{borrid}), \text{rdp12}(\text{lb}), \text{Nulloutp12} \rangle$   
 $\quad ==\text{Sendp12}(\text{Booksout})==\rangle$   
 $\quad \langle \text{inp12}(\text{borrid}), \text{rdp12}(\text{lb}), \text{outp12}(\text{Booksout}) \rangle$
5.  $\langle \text{inp12}(\text{borrid}), \text{Nullrdp12}, \text{Nulloutp12} \rangle$   
 $\quad ==\text{Errdelbr}(\text{borrid})==\rangle$   
 $\quad \langle \text{inp12}(\text{borrid}), \text{errdel}, \text{Nulloutp12} \rangle$
6.  $\langle \text{inp12}(\text{borrid}), \text{errdel}, \text{Nulloutp12} \rangle$   
 $\quad ==\text{Sendp12}(\text{Delnotinfile})==\rangle$   
 $\quad \langle \text{inp12}(\text{borrid}), \text{errdel}, \text{outp12}(\text{Delnotinfile}) \rangle$
7.  $\langle \text{inp12}(\text{borrid}), \text{rdp12}(\text{lb}), \text{outp12}(\text{dmess}) \rangle$   
 $\quad ==\text{Terminatep12}==\rangle$   
 $\quad \langle \text{Nullinp12}, \text{Nullrdp12}, \text{Nulloutp12} \rangle$

8. `<inp12(borrid), errdel, outp12(dmess)>`  
`==Terminatep12==>`  
`<Nullinp12, Nullrdp12, Nulloutp12>`
9. `<inp12(borrid), delbr, outp12(dmess)>`  
`==Terminatep12==>`  
`<Nullinp12, Nullrdp12, Nulloutp12>`

### The ASTS for the action `AddBorr` (P13)

`AddBorr_TS`  $\equiv$  P13state + P13label +

#### **Signature**

**transition relation**

`_==_==>_`: p13state, p13label, p13state

**Laws**  $\forall$  borrid:borrower\_id;nborr:new\_borr; n,a:list(character);

t:borrower\_type; out1:o1p13, out2:o2p13

1. `<Nullinp13, Nullrdp13, Nullout1p13, Nullout2p13>`  
`==Receivep1(nborr)==>`  
`<inp13(nborr), Nullrdp13, Nullout1p13, Nullout2p13>`
2. `<inp13(mknewborr(borrid, n, a, t)), Nullrdp13, Nullout1p13, Nullout2p13>`  
`==Readp13(borrid)==>`  
`<inp13(mknewborr(borrid, n, a, t)), rdp13(borrid), Nullout1p13, Nullout2p13>`
3. `<inp13(mknewborr(borrid, n, a, t)), Nullrdp13, Nullout1p13, Nullout2p13>`  
`==Erreadp13(borrid)==>`  
`<inp13(mknewborr(borrid, n, a, t)), erreadp13, Nullout1p13, Nullout2p13>`
4. `<inp13(mknewborr(borrid, n, a, t)), rdp13(borrid), Nullout1p13, Nullout2p13>`  
`==Send1p13(Alreadyinfile)==>`  
`<inp13(mknewborr(borrid, n, a, t)), rdp13(borrid),`  
`out1p13(Alreadyinfile), Nullout2p13>`
5. `<inp13(mknewborr(borrid, n, a, t)), dp13(borrid),`  
`out1p13(Alreadyinfile), Nullout2p13>`  
`==Terminatep13==>`  
`<Nullinp13, Nullrdp13, Nullout1p13, Nullout2p13>`
6. `<inp13(nborr), erreadp13, Nullout1p13, out2>`  
`==Send1p13(OKadd)==>`  
`<inp13(nborr), erreadp13, out1p13(OKadd), out2>`
7. `<inp13(mknewborr(borrid, n, a, t)), erreadp13, out1, Nullout2p13>`  
`==Send2p13(mkborr(borrid, n, a, t, emptylist, 0))==>`  
`<inp13(mknewborr(borrid, n, a, t)), erreadp13, out1,`  
`out2p13(mkborr(borrid, n, a, t, emptylist, 0))>`

```

8. <inp13(mknewborr(borrid, n, a, t)), erreadp13, out1p13(OKadd),
 out2p13(mkborr(borrid, n, a, t, emptylist, 0))>
 ==Terminatep13==>
 <Nullinp13, Nullrdrp13, Nullout1p13, Nullout2p13>

```

The asynchronous data flows of the library application are associated with queue structures each having an ADD access function, which puts objects on the queue, and a DEL access function, which removes the object at the top of the queue. The RSs specifying the asynchronous flows are not given here since they are merely instantiations of the queue RS given in Chapter 5. Below is a brief description of the RSs characterizing these data flows:

| Data flow        | RS name    | Access events |
|------------------|------------|---------------|
| new_book         | : Asynch1  | ADD1, DEL1    |
| delete_book      | : Asynch2  | ADD2, DEL2    |
| delete_message   | : Asynch3  | ADD3, DEL3    |
| return_info      | : Asynch4  | ADD4, DEL4    |
| return_message   | : Asynch5  | ADD5, DEL5    |
| checkout_info    | : Asynch6  | ADD6, DEL6    |
| checkout_message | : Asynch7  | ADD7, DEL7    |
| borr_update_info | : Asynch8  | ADD8, DEL8    |
| update_status    | : Asynch9  | ADD9, DEL9    |
| del_borr         | : Asynch10 | ADD10, DEL10  |
| del_borr_mess    | : Asynch11 | ADD11, DEL11  |
| new_borr         | : Asynch12 | ADD12, DEL12  |
| add_message      | : Asynch13 | ADD13, DEL13  |

The RSs characterising the behaviour of the data stores are given below:

Borr\_TS  $\equiv$  BorrStore + Borrlabel +

### Signature

transition relation

$\_==\_==>\_:$  list(borrower), borrlabel, list(borrower)

Laws  $\forall$  borrid:borr\_id; lbr:list(borrower); rub:ret\_updated\_borr;

oub:out\_updated\_borr; brec:borr

1. lbr==READBORR1(borrid, readborr1(lbr, borrid))==>lbr  
--- read associated with ret\_borr ---
2. lbr==READBORR2(borrid, readborr2(lbr, borrid))==>lbr  
--- read associated with out\_borr ---

3.  $lbr == READBORR3(borrid, readborr3(lbr, borrid)) ==> lbr$   
--- read associated with `borr_detail` ---
4.  $lbr == DELBORR(borrid, readborr4(lbr, borrid)) ==> delborr(lbr, borrid)$   
--- read associated with `deleted_borr` ---
5.  $lbr == READBORR5(borrid, readborr5(lbr, borrid)) ==> lbr$   
--- read associated with `other_borr` ---
6.  $lbr == UPDATEBR1(borrid, rub) ==> updatebr1(lbr, borrid, rub)$   
--- update associated with `ret_updated_borr` ---
7.  $lbr == UPDATEBR2(borrid, oub) ==> update2(lbr, borrid, oub)$   
--- update associated with `out_updated_borr` ---
8.  $lbr == UPDATEBR3(borrid, ubd) ==> updatebr3(lbr, borrid, ubd)$   
--- update associated with `updated_borr_detail` ---
9.  $lbr == PUTBR(brec) ==> brec|lbr$   
--- update associated with `new_borr_rec` ---

$Book\_TS \equiv BookStore + Booklabel +$

### Signature

**transition relation**

$_{==} _{==>} : list(book), booklabel, list(book)$

**Laws**  $\forall bid:book\_id; lbk:list(book); l:ISBN; rub:ret\_updated\_book;$

**oub:out\\_updated\\_book; bkrec:book**

1.  $lbk == READBOOK1(bid, readbk1(lbk, bid)) ==> lbk$   
--- read associated with `return_detail` ---
2.  $lbk == READBOOK2(bid, readbk2(lbk, bid)) ==> lbk$   
--- read associated with `out_book` ---
3.  $lbk == DELBOOK(bid, readbk3(lbk, bid)) ==> delbook(lbk, bid)$   
--- read associated with `deleted_book` ---
4.  $lbk == READBOOK3(i, readbk3(lbk, i)) ==> lbk$   
--- read associated with `copy#\_list` ---
5.  $lbk == PUTBK(bkrec) ==> bkrec|lbk$   
--- addition associated with `new_book_rec` ---
6.  $lbk == UPDATEBK1(bid, rub) ==> updatebk1(lbk, rub)$   
--- update associated with `ret_updated_book` ---
7.  $lbk == UPDATEBK2(bid, oub) ==> updatebk2(lbk, oub)$   
--- update associated with `out_updated_book` ---

The state of the ExtDFD representing the library application is of the form  $\langle \{a_1, \dots, a_7\}, as_1, \dots, as_{13}, \langle ds_1, l_1 \rangle, \langle ds_2, l_2 \rangle \rangle$ , where  $a_i$  is a state of action  $A_i$ ,  $as_i$

is an *asynchi* object, representing a state of its corresponding asynchronous data flow,  $\langle ds1, l1 \rangle$  is a state of the monitored data store BOOK, (*ds1* is the state of the data store while *l1* contains information about which objects in BOOK cannot be updated), and  $\langle ds2, l2 \rangle$  is the state of the monitored data store BORROWER, (*ds2* is the state of the data store while *l2* contains information about which objects in BORROWER cannot be updated).

The outline of the BS for the library ExtDFD is given below:

Lib\_BS  $\equiv$  Libstate + Liblabel +

### Signature

#### transition relation

$\_====\_====>\_$ : libstate, liblabel, libstate

### Laws

#### Synchronized Events

- A. Synchronized events between the receiving events of actions and the remove events (DEL) of asynchronous data flows. These laws are of the form below:

$$\begin{aligned} & ai == \text{Receivepi}(\text{data}) ==> ai', \quad asj == \text{DELj}(\text{data}) ==> asj' \Rightarrow \\ & \langle \{ai, sp\}, as1, \dots, asj, \dots, as13, \langle ds1, l1 \rangle, \langle ds2, l2 \rangle \rangle \\ & \quad == \text{SYNCH}(\{\{\text{Receivepi}(\text{bid}), \text{DELj}(\text{bid})\}\}) ==> \\ & \langle \{ai', sp\}, as1, \dots, asj', \dots, as13, \langle ds1, l1 \rangle, \langle ds2, l2 \rangle \rangle \end{aligned}$$

For example, the synchronized interaction between the action A5 and the asynchronous data flow *borr\_update\_info* is defined by the law:

{borrid:borrower\_id; n:integer}

$$\begin{aligned} & a5 == \text{SYNCH}(\{\{\text{Receivep10}(\text{borrid}), \text{Receive1p11}(n)\}\}) ==> a5', \\ & as8 == \text{DEL8}(\text{mkupinfo}(\text{borrid}, n)) ==> as8' \Rightarrow \\ & \langle \{a5, sp\}, as1, \dots, as8, \dots, as13, \langle ds1, l1 \rangle, \langle ds2, l2 \rangle \rangle \\ & \quad == \text{SYNCH}(\{\{\text{SYNCH}(\{\{\text{Receivep10}(\text{borrid}), \text{Receive1p11}(n)\}\}), \\ & \quad \text{DEL8}(\text{mkupinfo}(\text{borrid}, n))\}\}) ==> \\ & \langle \{a5', sp\}, as1, \dots, as8', \dots, as13, \langle ds1, l1 \rangle, \langle ds2, l2 \rangle \rangle \end{aligned}$$

- B. Synchronized events between actions and data stores. For example, the law characterizing the interaction between the action A3 and the data store BOOK via the data flow *ret\_updated\_book* is:

{rub:ret\_updated\_book; bid:book\_id}

$$\begin{aligned} & a3 == \text{Send2p4}(\text{bid}, \text{rub}) ==> a3', \quad ds1 == \text{UPDATEBK1}(\text{bid}, \text{rub}) ==> ds1' \Rightarrow \\ & \langle \{a3, sp\}, as1, \dots, as13, \langle ds1, l1 \rangle, \langle ds2, l2 \rangle \rangle \\ & \quad == \text{SYNCH}(\{\{\text{Send2p4}(\text{bid}, \text{rub}), \text{UPDATEBK1}(\text{bid}, \text{rub})\}\}) ==> \\ & \langle \{a3', sp\}, as1, \dots, as13, \langle ds1', \text{delete}(\text{bid}, l1) \rangle, \langle ds2, l2 \rangle \rangle \end{aligned}$$



Single Events

- C. All action events which are concerned with the reading of state flows from external entities. Such events are of the form:

$$\begin{aligned}
 & a_i \text{==|} \Rightarrow a_i' \Rightarrow \\
 & \langle \{a_i, sp\}, as_1, \dots, as_{13}, \langle ds_1, l_1 \rangle, \langle ds_2, l_2 \rangle \rangle \\
 & \text{==|} \Rightarrow \\
 & \langle \{a_i', sp\}, as_1, \dots, as_{13}, \langle ds_1, l_1 \rangle, \langle ds_2, l_2 \rangle \rangle
 \end{aligned}$$

where  $l$  is an event label encapsulating the observable effect of a read from a state flow event.

Parallel Events

- D. All ExtDFD events that affect mutually exclusive parts of the ExtDFD state can be carried out in parallel. These laws are of the form:

$$\begin{aligned}
 & \langle \{sp_1, sp_2\}, as_1, \dots, as_{i-1}, as_i, \dots, as_{13}, \langle ds_1, l_1 \rangle, \langle ds_2, l_2 \rangle \rangle \\
 & \text{==|} \Rightarrow \\
 & \langle \{sp_1', sp_2'\}, as_1, \dots, as_{i-1}, as_i', \dots, as_{13}', \langle ds_1', l_1' \rangle, \langle ds_2, l_2 \rangle \rangle, \\
 & \langle \{sp_1, sp_2\}, as_1, \dots, as_{i-1}, as_i, \dots, as_{13}, \langle ds_1, l_1 \rangle, \langle ds_2, l_2 \rangle \rangle \\
 & \text{==|} \Rightarrow \\
 & \langle \{sp_1, sp_2'\}, as_1', \dots, as_{i-1}', as_i, \dots, as_{13}, \langle ds_1, l_1 \rangle, \langle ds_2', l_2' \rangle \rangle \Rightarrow \\
 & \langle \{sp_1, sp_2\}, as_1, \dots, as_{i-1}, as_i, \dots, as_{13}, \langle ds_1, l_1 \rangle, \langle ds_2, l_2 \rangle \rangle \\
 & \text{==|} \parallel \Rightarrow \\
 & \langle \{sp_1', sp_2'\}, as_1', \dots, as_{i-1}', as_i', \dots, as_{13}', \langle ds_1', l_1' \rangle, \langle ds_2', l_2' \rangle \rangle
 \end{aligned}$$

Also all ExtDFD events which affect mutually exclusive substates of an action can be carried out in parallel. For example, the output events of the action A4 may occur in parallel, that is the synchronized update interactions between A4 and the data stores BOOK and BORROWER and the synchronized interaction between A4 and the asynchronous data flow checkout\_message can occur in parallel.

**6.3 Conclusion**

The examples presented in this chapter illustrate how formal specifications can be derived from DFDs extended with notation for depicting control relationships. The two different types of applications used show that the techniques are equally applicable to data, and control-intensive applications. The formal specifications derived are, admittedly, not easy to read or understand, nor are they easy to produce manually. In this respect, the examples highlight the need for powerful specification building and derivation tools in the practical application of the framework.

---

Once produced, the formal specifications can be used to rigorously validate and verify behavioural properties, and in this respect, they serve to establish confidence in the software product and the activities involved in building the product.

# CHAPTER 7

## Conclusions and Further Work

### 7.1 Thesis Summary and Achievements

In this thesis a formal framework for developing and interpreting DFDs was developed. The framework provides DFDs with a mathematical basis, and consists of two parts: the Picture Level (PL) and the Specification Level (SL). The PL is a mathematical theory characterizing the syntactic properties of DFDs. The theory can be used to investigate the absence or presence of syntactic properties in DFDs. The operational interpretation associated with the PL takes the form of a relational conditional term rewriting system (R-CTRS), and provides an effective means for carrying out the investigation of the syntactic properties. *Structural correctness* is a useful syntactic property that can be investigated in the PL. A DFD construct, or a structure of DFD constructs, is said to be structurally correct if it satisfies the formation rules associated with it. Such rules are directly stated as laws of the PL.

The SL provides support for specifying control in DFDs and for deriving initial design from DFDs. It consists of tools and techniques for describing state dependent behaviour and control relationships in DFDs, and for deriving formal specifications, called Behavioural Specifications (BSs), from control-extended DFDs, called ExtDFDs. An ExtDFD is derived from a hierarchy of DFDs, in the following manner:

- 1 Generate the primitive DFD of the hierarchy. The primitive DFD consists of the primitive processes of the hierarchy, and the external entities and data stores of the hierarchy. Decomposed and combined data flows are depicted in the primitive DFD via splitters and binders.
- 2 Add control flows and a state entity to the primitive DFD to pictorially describe the state dependent behaviour of the application. Specify the behaviour of the state entity in terms of a state transition diagram (STD).
- 3 Partition the primitive processes into actions, and identify the asynchronous and state flow interfaces between actions and the external entities. The internal data flows of actions are all synchronous, as well as the data flows between actions and data stores. The data flows between actions are all asynchronous.

An ExtDFD is viewed in the formal framework as a system of actions which interacts with its environment (depicted by external entities) in an uncooperative manner (depicted by asynchronous and/or state interfaces between external entities and the ExtDFD). The BS of an ExtDFD characterizes the behaviour of the BS in terms of its allowable state transitions, and is generated from information

concerning the relationships between ExtDFD components, explicitly depicted in the ExtDFD (for example, via the use of special symbols for synchronous, asynchronous, state, and control flows), formal specifications of behaviour of the processes, data stores, and asynchronous flows, in terms of labeled state transition systems, and specifications of the data structures associated with the data objects in the ExtDFD. The BS is derived in a modular manner:

- 1 the static and dynamic aspects of data stores and data flows are formally specified;
- 2 the behaviour of processes are formally specified;
- 3 the specifications of actions are generated from specifications of their constituent processes, and the synchronous relationships between them (which are explicitly depicted in the ExtDFD);
- 4 The BS is generated from the specifications generated in 1 and 3, and from the types of interactions between actions, data stores, and external entities depicted in the ExtDFD.

The BS can be used to formally validate behavioural properties of ExtDFDs, and can also be used as the basis for formal verification of subsequent implementations.

### 7.1.1 Achievements

The formal framework described in this thesis provides a firm mathematical foundation for DFDs which can be used as a basis for formally evaluating the structure of DFDs, and which facilitates the generation of formal specifications from them. Earlier work in this respect [TP86b, Tse85a, Tse85b] provide only formal foundations for the syntactic aspects of DFDs. The framework developed here provides a formal basis for both the syntactic and semantic aspects of DFDs, and thus can be viewed as extensions of these earlier works.

The formal framework also provides facilities for depicting and formally specifying control information in DFDs. The work on this aspect of the formal framework improves upon other popular approaches to introducing control information in DFDs [HP87, Woo88], by associating formal interpretations with DFD structures built up with the additional control constructs. As above, this facilitates the generation of formal specifications from the control-extended DFDs. The use of the formal specifications for formally investigating behavioural properties of applications, and as bases for formal verification activities, is discussed in Chapter 5.

Once the BS is generated from an ExtDFD, the ExtDFD can be viewed as the informal 'front' of the BS. This provides the BS with a more visually appealing

front, which abstracts away from its detail but still provides insight, via the graphical notation, consistent with such detail. This approach supports Naur's view of formalisms as extensions of informal expressions.

The formal framework thus facilitates the generation of specifications which are *understandable*, by providing a graphical 'front' in the form of ExtDFDs, *precise*, by facilitating the generation of formal specifications from ExtDFDs, and *testable*, by providing formal notions of specifications implementing the BS.

The specification technique used by the framework extends and combines current algebraic specification techniques in order to derive a more expressive specification system. The extensions made in this respect concern the derivation of model-theoretic and operational interpretations for specifications with partial functions, negated relations (predicates), and inequalities. This work builds upon the work of Wirsing and Broy on partial algebraic specifications [WB82], Astesiano et al on relational specifications [ARW86], and Mohan and Srivas on model-theoretic and operational interpretations of specifications with inequalities [MS87].

### 7.1.2 Comments

It is this author's opinion that the number of useful automated tools supporting the use of SA tools and techniques will gradually level out if no formal basis for the tools and techniques are developed. Too often have practical tools been built without first establishing a formal foundation for the techniques they support. Many such tools are of superficial use only, for example, most tools for DFDs currently available have relatively firm foundations for the syntactic aspects, but provide little or no foundation for the semantic aspects, thus limiting their use in formally specifying and investigating behavioural properties. Yet, it is the investigation of these behavioural properties that will have a bearing on subsequent development. This thesis attempts to change matters by providing a formal framework which can be used as the basis for the building of automated tools supporting the use of DFDs in software development. No attempt has been made in the thesis to suggest particular tools based on the framework, but the mathematical theories have been developed in a manner that does not preclude practical implementation. Thus conditions under which sound and complete rewriting systems can be generated from RSs (the theories) are provided, and can be used as guidelines in constructing the RSs.

## 7.2 Further Work

The PL of the formal framework can be extended in many directions as indicated in Chapter 4. Support for formally reasoning about the syntactic properties of incomplete structures, as well as support for modifying DFDs, and for reasoning about such modifications are some of the more useful extensions that can be made to the PL. Such extensions are simply exercises in building theories of well-defined syntactic manipulations. Further work is also needed in making practical use of the SL. In this respect, computer-aided tools for interrogating BSs, and for analyzing the behavioural properties they capture are essential. Work by this author and Docker on a practical environment for the formal framework is currently in progress [DF89, FD89]. The structure of the proposed environment is shown in Figure 7.1.

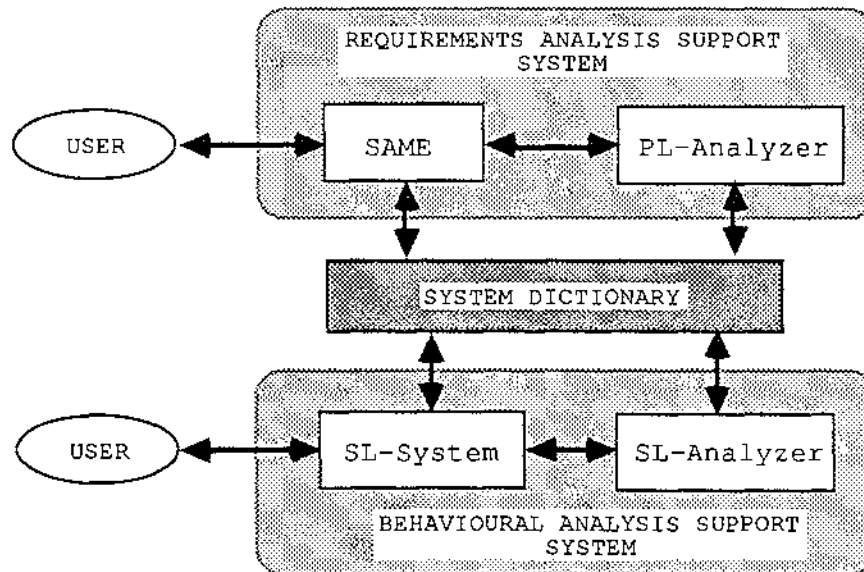


Figure 7.1 The structure of an environment incorporating the formal framework

The proposed environment consists of two sub systems: the *Requirements Analysis Support System* (RASS), and the *Behavioural Analysis Support System* (BASS). The two sub systems are supported by a system dictionary which stores the representations generated by them, and facilitates the sharing of such representations.

In the RASS DFDs are developed informally using SAME [Doc88], which is an executable DFD specification system. The PL-Analyzer, a DFD syntax checking tool based on the PL theory, is used to check the syntactic consistency of the DFDs constructed in SAME. SAME acts as the informal 'front' of the RASS, providing tools for drawing diagrams, for entering semi-formal descriptions of DFD

components, and for building executable data dictionaries for the DFDs. The executable nature of SAME specifications essentially makes the RASS a prototyping system.

The BASS provides support for the specification and analysis of behavioural properties of application with DFDs, and is based on the SL. The SL-system is the front end of the BASS and consists of automated tools for extending DFDs with control information, and for entering process specifications, and specifications of the static and dynamic aspects of data stores and data flows. From these, the SL-system generates the BS. The SL-Analyzer provides tools for analyzing the BS. Work on the BASS is still in the initial stages.

Further research is needed in incorporating the formal framework in a formal development method. An evolutionary method, somewhat similar to the transformation approach described in Chapter 0, where a program is derived from a sequence of specifications, with the BS as the start of the sequence, and where each specification in the sequence implements the specification prior to it in the sequence, is a possibility that warrants further investigation. The criteria for establishing implementation described in Chapter 5 can be used in such a method.

### 7.3 Conclusion

To conclude, further research and work is needed in order to make practical use of the formal framework via automated support environments for formally specifying applications with DFDs. The framework, though, has the potential to initiate research into a new generation of 'semantically-based' automated tools for DFDs, which could see their use as specification tools in formal development methods. Furthermore, the graphical nature of DFDs, coupled with the formal foundation developed here, makes for a formal specification method which does not sacrifice understandability for formality.

# Bibliography

\* LNCS is the abbreviated form for Lecture Notes in Computer Science

- [AL88] Abadi, M., & Lamport, L., 'The Existence of Refinement Mappings', Digital Systems Research Centre Report, Aug 1988.
- [AGR88] Astesiano, E., Giovini, A., & Reggio, G., 'Data in a Concurrent Environment', in *Proceedings of the International Conference on Concurrency*, LNCS 335, Springer-Verlag, 1988, 140-159.
- [AR87] Astesiano, E., & Reggio, G., 'SMoLCS-Driven Concurrent Calculi', in *TAPSOFT '87*, Eds H. Ehrig, R. Kowalski, G. Levi, & U. Montanari, Vol.1, LNCS 249, Springer-Verlag, 1987, 169-201.
- [ARW86] Astesiano, E., Reggio, G., & Wirsing, M., 'Relational Specifications and Observational Semantics', in *Mathematical Basis for Computer Science*, Eds G. Goos & J. Hartmanis, LNCS 233, 1986, 209-217.
- [BG87] Balzer, R., & Goldman, N., 'Principles of Good Software Specification and their Implications for Specification Languages', in *Software Specification Techniques*, Eds N. Gehani, & A. D. McGettrick, 25-39.
- [BK82] Bergstra, J.A., & Klop, J.W., 'Conditional Rewrite Rules: Confluency and Termination', Research Report IW 198/82, Mathematical Centre of Amsterdam, 1982.
- [Boe76] Boehm, B. W., 'Software Engineering', *IEEE Transactions on Computers*, Vol. C-25, No. 12, Dec. 1976, 1226-1241.
- [Boe81] Boehm, B. W., *Software Engineering Economics*, Prentice Hall, 1981.
- [BW83] Broy, M., & Wirsing, M., 'On The Algebraic Specification Of Finitary Infinite Communicating Sequential Processes', in *Formal Description of Programming Concepts-II*, Eds D. Bjorner, North-Holland, IFIP, 1983, 171-198.
- [CTL87] Chua, T.S., Tan, K.P., & Lee, P.T., 'EXT-DFD: A Visual Language for Extended DFD', Technical Report, Department of Information Systems and Computer Science, National University of Singapore, 1987.
- [CTRS87] *Proceedings of the 1st International Workshop on Conditional Term Rewriting Systems*, Eds S. Kaplan, & J.P. Jouannaud, LNCS 308, Springer-Verlag, 1987.



- [DF89] Docker, T.W.G., & France, R.B., 'Flexibility and Rigour in Structured Analysis', in *Information Processing 89*, Ed G.X. Ritter, Elsevier Science Publishers (North-Holland), 1989, 89-94.
- [DeM78] DeMarco, T., *Structured Analysis and System Specification*, Prentice-Hall, 1978.
- [Der87] Dershowitz, N., 'Termination of Rewriting', *Journal of Symbolic Computation*, Vol 3, 1987, 69-116.
- [Doc86] Docker, T. W. G., & Tate, G., 'Executable Data Flow Diagrams', in *Software Engineering '86*, Eds D. Barnes, & P. Brown, Peter Peregrinus, 1986, 352-370.
- [Doc87] Docker, T. W. G., 'A Flexible Software Analysis Tool', *Information and Software Technology*, Vol 29, No 1, Jan/Feb 1987.
- [Dro84] Drosten, K., 'Towards Executable Specifications Using Conditional Axioms', in *Symposium on Theoretical Aspects of Computer Science*, LNCS 166, Springer-Verlag, 1984, 85-96.
- [FD89] France, R.B., & Docker, T.W.G., 'Formal Specification using Structured Analysis', in *ESEC '89*, Eds C. Ghezzi, & J.A. McDermid, LNCS 387, Springer-Verlag, 1989, 292-310.
- [FP86] Finkelstein, A.C.W., & Potts, C., 'Structured Common Sense: The elicitation and formalization of system requirements', in *Software Engineering '86*, Eds P.J. Brown, & D.J. Barnes, Peter Peregrinus, 1986.
- [FREQ79] Position papers for panel session: 'What is a Formal Requirements Specification? What does it contain, How is it Structured and For What is it Useful', in *Formal Models and Practical Tools for Information Systems Design*, IFIP, Ed H. -J. Schneider, North-Holland, 1979, 281-287.
- [Gom84] Gomaa, H., 'A Software Design Method for Real-Time Systems', *Communications of the ACM*, Vol 27, No 9, Sept 1984, 938-949.
- [Gom86] Gomaa, H., 'Software Development of Real-Time Systems', *Communications of the ACM*, Vol 29, No 7, July 1986, 657-668.
- [Goo84] Good, D.I., 'Mechanical Proofs About Computer Programs', in *Proceedings of the Philosophical Transactions of the Royal Society, Mathematical Logic and Programming Languages*, Eds Sir Michael Atiyah, C.A.R. Hoare, & J.C. Shepherdson, Royal Society, 1984.
- [GS79] Gane, C., & Sarson, T., *Structured Systems Analysis: Tools and Techniques*, Prentice-Hall, 1979.

- [GHM78] Guttag, J. V., Horowitz, E., & Musser, D. R., 'The Design of Data Type Specifications', in *Current Trends in Programming Methodology*, Vol 4, Ed R. Yeh, Prentice-Hall, 1978, 60-79.
- [GTW78] Goguen, J. A., Thatcher, J. W., & Wagner, E. G., 'An Initial Algebra Approach to the Specification, Correctness, and Implementation of Abstract Data Types', in *Current Trends in Programming Methodology*, Vol 4, Ed R. Yeh, Prentice-Hall, 1978, 80-149.
- [HO80] Huet, G., & Oppen, D.C., 'Equations and Rewrite Rules: a survey', in *Formal Languages: Perspectives and Open Problems*, Ed R. Book, Academic Press, 1980.
- [Hoa78] Hoare, C. A. R., 'Communicating Sequential Processes', *Communications ACM*, Vol 21, No 8, Aug 1978, 666-677.
- [HP87] Hatley, D., & Pirbhai, I., *Strategies for Real-Time System Specification*, Dorset House, 1987.
- [Hue80] Huet, G., 'Confluent Reductions: Abstract Properties and Applications to Term Rewriting Systems', *Journal of the ACM*, Vol 27, No 4, Oct 1980, 797-821.
- [Jou87] Jouannaud, J. P., 'Reductive Conditional Term Rewriting Systems', in *Formal Descriptions of Programming Concepts-III*, IFIP, Ed M. Wirsing, North-Holland, 1987, 223-244.
- [Kap84] Kaplan, S., 'Fair Conditional Term Rewriting Systems: Unification, Termination and Confluence', Research Report, Universite de Paris-Sud, 1984.
- [Kap87] Kaplan, S., 'Positive/Negative Conditional Rewriting', in [CTRS87], 129-141.
- [KKZ88] Koymans, R., Kuiper, R., & Zijlstra, E., 'Paradigms for Real-Time Systems', in *Formal Techniques in Real-Time and Fault-Tolerant Systems*, LNCS 331, 1988, 159-174.
- [KP87] Kaplan, S., Pnueli, A., 'Specification and Implementation of Concurrently Accessed Data Structures: An Abstract Data Type Approach', in *Proceedings of 4th Annual Symposium on Theoretical Aspects of Computer Science*, LNCS 247, 1987.
- [Lam86] Lamport, L., 'A Simple Approach to Specifying Concurrent Systems', Digital Systems Research Center Report, Dec 1986.
- [LZ75] Liskov, B. H., & Zilles, S., 'Specification Techniques for Data Abstractions', *EEE Transactions on Software Engineering*, Vol SE-1, Jan 1975, 7-19.

- [LZ77] Liskov, B. H., & Zilles, S., 'An Introduction to Formal Specifications of Data Abstractions', in *Current Trends in Programming Methodology*, Vol 1, Ed R. Yeh, Prentice-Hall, 1977, 1-32.
- [Mil80] Milner, R., 'A Calculus of Communicating Systems', Springer-Verlag, 1980.
- [MP84] McMenamin, S.M., & Palmer, J.F., *Essential Systems Analysis*, Yourdon Press, 1984.
- [MS87] Mohan, C., & Srivas, M.K., 'Conditional Specifications with Inequational Assumptions', in [CTRS87], 161-178.
- [MW86a] Mellor, S. J., & Ward, P. T., *Structured Development for Real-Time Systems Vol 1: Introduction and Tools*, Yourdon Press, 1986.
- [MW86b] Mellor, S. J., & Ward, P. T., *Structured Development for Real-Time Systems Vol 2: Essential Modeling Techniques*, Yourdon Press, 1986.
- [Nau82] Naur, P., 'Formalization in Program Development', *Bit* 22, 1982, 437-453.
- [Nau85] Naur, P., 'Intuition in Software Development', in *Formal Methods and Software Development*, LNCS 186, Springer, 1985, 60-79.
- [Pet88] Peters, L., *Advanced Structured Analysis and Design*, Prentice-Hall, 1988.
- [Pete81] Peterson, J. L., *Petri Net Theory and the Modeling of Systems*, Prentice-Hall, 1981.
- [Pong86] Pong, L., 'Formal Data Flow Diagrams (FDFD): A Petri Net Based Requirements Specification Language', M.Phil. thesis, Centre of Computer Studies and Applications, University of Hong Kong, 1986
- [Pre87] Pressman, R. S., *Software Engineering: A Practitioner's Approach*, McGraw-Hill, 1987.
- [Ric86] Richter, C. A., 'An Assessment of Structured Analysis and Structured Design', *ACM SIGSOFT Software Engineering Notes*, Vol 11, No 4, Aug 1986, 41-45.
- [Ross77] Ross, D. T., 'Structured Analysis (SA): A Language For Communicating Ideas', *IEEE Transactions on Software Engineering*, Vol. SE-3, No. 1, Jan. 1977, 16-34.
- [RS77] Ross, D. T., & Schoman, K. E., 'Structured Analysis for Requirements Definition', *IEEE Transactions on Software Engineering*, Vol SE-3, No 1, Jan 1977, 6-15.
- [RZ84] Remy, J.-L., & Zhang, H., 'Reveur4: A System for Validating Conditional Algebraic Specification of Parameterized Abstract Data Types', in *Proceedings of the 2nd ECAI Conference*, 1984.

- [Sho88] Shoal, P., 'ADISSA: Architectural Design of Information Systems Based on Structured Analysis', *Information Systems*, Vol. 13, No. 2, 1988, 193-210.
- [SP88] Shoal, P., & Pliskin, N., 'Structured Prototyping: Integrating Prototyping into Structured System Development', *Information & Management*, 14, 1988, 19-30.
- [SSD87] *Proceedings of the Fourth International Workshop on Software Specification and Design*, IEEE Computer Society Press, 1987.
- [ST87] Sannella, D.T., & Tarlecki, A., 'On Observational Equivalence and Algebraic Specification', *Journal of Computer and System Sciences*, Vol 34, 1987, 150-178.
- [TePi85] Teague, L. C., & Pidgeon, C. W., *Structured Analysis Methods For Computer Information Systems*, Science Research Associates, 1985.
- [TP86a] Tse, T. H., Pong, L., 'An Examination of System Requirements Specification Languages', Technical Report, Centre of Computer Studies and Applications, University of Hong Kong, 1986.
- [TP86b] Tse, T. H., Pong, L., 'Towards a Formal Foundation for DeMarco Data Flow Diagrams', Technical Report, Centre of Computer Studies and Applications, University of Hong Kong, 1986.
- [Tse85a] Tse, T. H., 'An Algebraic Formulation for Structured Analysis Design and Models', Technical Report, Centre of Computer Studies and Applications, University of Hong Kong, 1985.
- [Tse85b] Tse, T. H., 'Towards a Unified Algebraic View of the Structured Analysis and Design Models', Technical Report, Centre of Computer Studies and Applications, University of Hong Kong, 1985.
- [Tse86] Tse, T. H., 'Integrating the Structured Analysis and Design Models: An Initial Algebra Approach', *Australian Computer Journal*, Vol 18, No 3, 1986, 121-127.
- [Tse87] Tse, T. H., 'Integrating the Structured Analysis and Design Models: A Category-Theoretic Approach', *Australian Computer Journal*, Vol 19, No 1, 1987, 25-31.
- [War86] Ward, T., 'The Transformation Schema: An Extension Of The Data Flow Diagram To Represent Control And Timing', *IEEE Transactions on Software Engineering*, Vol. SE-12, No.2, February 1986, 198-210.
- [WB82] Wirsing, M., & Broy, M., 'An Analysis of Semantic Models For Algebraic Specifications', in *Theoretical Foundations of Programming Methodology*, Eds M. Broy, & G. Schmidt, NATO Advanced Study Series, Series C, Vol. 91, D. Reidel, 1982, 351-412.

- [Wei78] Weinberg, V., *Structured Analysis*, Prentice-Hall, 1978.
- [Woo88] Woodman, M., 'Yourdon Dataflow Diagrams: A tool for disciplined requirements analysis', *Information and Software Technology*, Vol 30, No 9, Nov 1988, 515-533
- [WS79] Wasserman, A., & Stinson, S., 'A Specification Method for Interactive Information Systems', in *Proceedings of the Symposium on the Specification of Reliable Software*, IEEE, 1979, 68-79.
- [YC79] Yourdon, E., Constantine, L., *Structured Design: Fundamentals of a Discipline of Computer Program and System Design*, Second Edition, Yourdon Press, 1979.
- [YZCC84] Yeh, R. T., Zave, P., Conn, A.P., & Cole, G. E., 'Software Requirements: New Directions and Perspectives', in *Handbook of Software Engineering*, Eds C.R. Vick, & C.V. Ramamoorthy, Van Norstrand Reinhold, 1984, 519-543.
- [Zav82] Zave, P., 'An Operational Approach to Requirements Specification for Embedded Systems', *IEEE Transactions on Software Engineering*, Vol 8, No 3, May 1982, 250-269.
- [ZY81] Zave, P., & Yeh, R., 'Executable Requirements for Embedded Systems', in *Proceedings of the 5th IEEE Conference on Software Engineering*, IEEE Press, 1981, 285-304.

# APPENDIX I

## Conditional Term Rewriting Systems

This appendix is an introduction to term rewriting systems, in particular, conditional term rewriting systems.

### Properties of terms

Let  $T$  be the set of terms generated by a signature  $\Sigma = \langle S, F \rangle$ , where  $S$  is a set of sorts and  $F$  is a set of function symbols, and let  $X$  be a countable set of elements, called variables, which are denoted by  $x, y, z$ . The function  $\text{Var}$  on terms returns the set of variables occurring in a term and is defined as follows :

$\text{Var}: \text{Var}(x) = \{x\}$  where  $x \in X$

$\text{Var}(f(t_1, \dots, t_n)) = \text{Var}(t_1) + \dots + \text{Var}(t_n)$  where  $f \in F$  and  $t_i \in T$  ( $1 \leq i \leq n$ )

If  $V(t) = \emptyset$  then  $t$  is called a *ground term*.

An occurrence of a sub term in a term is defined in terms of a set of sequences of integers,  $N^*$ , including the empty sequence  $\lambda$ , and a concatenation operation,  $\cdot$ , on sequences. The elements of  $N^*$  are called *occurrences*. An ordering  $\leq$ , called the *prefix ordering*, is defined on occurrences as follows:

$u \leq v$  if and only if there exists  $w$  such that  $v = u \cdot w$ , where  $u, v, w \in N^*$ . Also  $v/u = w$  if and only if  $v = u \cdot w$ , where  $u, v, w \in N^*$ .

Intuitively  $u \leq v$  if  $u$  can be made equal to  $v$  by appending a sequence to it. Two occurrences,  $u, v$ , are said to be disjoint, denoted by  $u \setminus v$ , if and only if  $\text{NOT}(u \leq v)$  and  $\text{NOT}(v \leq u)$ , that is neither  $u$  or  $v$  can be made equal by appending sequences to them. Also  $u < v$  if and only if  $u \leq v$  and  $u \neq v$ , where  $u$  and  $v$  are occurrences.

The set of occurrences of a term,  $t$ , denoted by  $O(t)$ , and the sub term of  $t$  at occurrence  $u$ , denoted by  $t|_u$ , are defined as follows:

1. If  $t = x$  then  $O(t) = \{\lambda\}$  and  $t|_\lambda = t$ .
2. If  $t = f(t_1, \dots, t_n)$  then  $O(t) = \{\lambda\} + \{iu \mid i \leq n, u \in O(t_i)\}$ ,  $t|_\lambda = t$ , and  $t|_{iu} = t_i|_u$ .

For example, a term  $t = f(g(x, h(y)), k(x, z))$ , has an occurrence set  $O(t) = \{\lambda, 1, 11, 12, 121, 2, 21, 22\}$ , where  $t|_\lambda = f(g(x, h(y)), k(x, z))$ ,  $t|_1 = g(x, h(y))$ ,  $t|_{12} = h(y)$ ,  $t|_{121} = y$ , and  $t|_2 = k(x, z)$ . Also  $1 \leq 11$ ,  $1 \leq 12$ ,  $1 \leq 121$ , and  $12 \leq 121$ .

*Replacement* of a sub term at occurrence  $u$  of a term,  $t$ , by another term,  $t'$ , denoted by  $t[u \leftarrow t']$ , is defined as follows:

1.  $t[\lambda \leftarrow t'] = t'$ .
2. If  $t = f(t_1, \dots, t_n)$ ,  $t[iu \leftarrow t'] = f(t_1, \dots, t_{i-1}, t_i[u \leftarrow t'], \dots, t_n)$ ,  $i \leq n$ .

Replacements have the following properties.

$\forall t, t1, t' \in T; u \in O(t), v \in O(t1):$

- *Embedding* :  $t[u \leftarrow t']|u.v = t'|v,$
- *Associativity* :  $t[u \leftarrow t']|u.v \leftarrow t1 = t[u \leftarrow t']|v \leftarrow t1$ .

$\forall t, t1, t' \in T; u, v \in O(t),$  with  $u \setminus v:$

- *Persistence* :  $t[u \leftarrow t']|v = t|v,$
- *Commutativity* :  $t[u \leftarrow t']|v \leftarrow t1 = t|v \leftarrow t1[u \leftarrow t'].$

$\forall t, t1, t' \in T; u, v \in O(t),$  with  $u \leq v:$

- *Distributivity* :  $t[u \leftarrow t']|v = (t|v)[u/v \leftarrow t'],$
- *Dominance* :  $t[u \leftarrow t']|v \leftarrow t1 = t|v \leftarrow t1.$

*Substitution* of variables in a term is defined as follows:

A substitution is a mapping,  $\sigma$ , from  $X$ , the set of variables, to  $T$ , the set of terms, with  $\sigma(x) = x$  almost everywhere. They are extended to morphisms of  $T$  by  $\sigma(f(t1, \dots, tn)) = f(\sigma(t1), \dots, \sigma(tn))$ . The *domain* of a substitution  $\sigma$  is the finite set  $D(\sigma) = \{x \in X \mid \sigma(x) \neq x\}$ .

The *match* of a term  $t$  by another term  $t'$ , denoted by  $t::t'$ , is defined as follows:

$t::t'$  if and only if there exists a substitution  $\sigma$  such that  $t = \sigma(t')$ . Any such substitution is denoted by  $\sigma = t::t'$  in what follows.

Intuitively, the match of a term  $t$  by another term  $t'$  occurs when a substitution exists that when applied to  $t'$  makes it identical to the term  $t$ .

## Term rewriting systems

Term rewriting systems are formally defined below:

A *term rewriting system* (TRS) is a set  $R$  of pairs of terms  $\langle t \rightarrow t' \rangle$ , such that  $\text{Var}(t)$  is a subset of  $\text{Var}(t')$ . The pairs of terms are called (rewrite) *rules*.

The following are definitions associated with TRSs.

An occurrence,  $u$ , in a term  $t$  in a TRS,  $R$ , is called a *redex occurrence of  $R$  in  $t$*  if and only if there exists a rule  $\langle t1 \rightarrow t2 \rangle$  in  $R$  such that  $t1 \leq t|u$ .

A term  $t$  *reduces* (or rewrites) to  $t'$  in a TRS  $R$ , denoted by  $t \rightarrow_R t'$ , if there exists a rule  $\langle t1 \rightarrow t2 \rangle$  in  $R$ , and a substitution  $\sigma = (t|u)::t1$  (i.e.  $u$  is a redex occurrence of  $R$  in  $t$ ), and  $t' = t[u \leftarrow \sigma(t2)]$ .

If  $\rightarrow$  is a relation over  $T$ , then  $\rightarrow$  is

- (a) *stable* if and only if  $\forall \sigma; \forall t, t' \in T; t \rightarrow t' \Rightarrow \sigma(t) \rightarrow \sigma(t')$ ;
- (b) *compatible* if and only if  $\forall t, t', t1 \in T, \forall u \in O(t1); t \rightarrow t' \Rightarrow t1[u \leftarrow t] \rightarrow t1[u \leftarrow t']$ .

The reduction relation  $\rightarrow_R$  is the smallest compatible and stable relation containing  $R$  (see Huet [Hue80]). For a relation on terms let:

- $\rightarrow_R^*$  denotes the transitive closure of  $\rightarrow_R$ ,
- $t \downarrow t' \Leftrightarrow \exists t_1$  such that  $t \rightarrow_R^* t_1$  and  $t' \rightarrow_R^* t_1$ ,
- $t \uparrow t' \Leftrightarrow \exists t_1$  such that  $t_1 \rightarrow_R^* t$  and  $t_1 \rightarrow_R^* t'$

If  $t$  is minimal with respect to  $\rightarrow_R$  i.e. there is no  $t'$  such that  $t \rightarrow_R t'$ , then  $t$  is called a  $\rightarrow_R$ -normal form. For a term  $t$ , if there exists a  $\rightarrow_R$ -normal form  $t'$  such that  $t \rightarrow_R^* t'$  then  $t'$  is called a normal form of  $t$ .

Two important properties of  $\rightarrow_R$  are termination and confluence.

A relation,  $\rightarrow_R$ , is *terminating* (noetherian) if and only if there is no infinite sequence  $t_1 \rightarrow_R t_2 \rightarrow_R \dots \rightarrow_R t_n$ , that is,  $\rightarrow_R^*$  is well founded.

A relation,  $\rightarrow_R$ , is confluent if and only if  $\forall t, t' \in T; t \uparrow t' \Rightarrow t \downarrow t'$ .

Every term in a terminating and confluent relation possesses a unique normal form (see Huet [Hue80]). Rewriting systems which generate a terminating and confluent relation on terms provide an effective procedure for determining the equality of terms. In such systems, if two terms,  $t, t'$  reduce to the same normal term, then the terms are equal, also two equal terms in the equational theory corresponding to the TRS (obtained by replacing the symbol  $\rightarrow$  by  $=$ ), reduce to the same normal form in the TRS. Proof of this can be found in Huet [Hue80].

### Conditional term rewriting systems

Conditional term rewriting systems (CTRSs) are extensions to TRSs which allow conditions to be associated with reductions.

A *conditional term rewriting system* is a finite set of rules, called *conditional rewrite rules*, of the following form:  $u_1 = v_1 \wedge \dots \wedge u_n = v_n \Rightarrow \text{Lhs} \rightarrow \text{Rhs}$  where  $\text{Var}(u_i)$  is a subset of  $\text{Var}(\text{Lhs})$ ,  $\text{Var}(v_i)$  is a subset of  $\text{Var}(\text{Lhs})$ , and  $\text{Var}(\text{Rhs})$  is a subset of  $\text{Var}(\text{Lhs})$ , for  $1 \leq i \leq n$  [Kap84]. The formula before the implication symbol,  $\Rightarrow$ , is called the condition part or *antecedent*, while the reduction after the  $\Rightarrow$  is called the *consequence*.

Rewriting in a CTRS is defined as follows:

Given a CTRS,  $R$ , a term  $t$  *reduces* (or rewrites) to  $t'$ , denoted by  $t \rightarrow_R t'$ , if and only if there is a rule in  $R$ ,  $u_1 = v_1 \wedge \dots \wedge u_n = v_n \Rightarrow \text{Lhs} \rightarrow \text{Rhs}$  in  $R$  such that:



### Match and replace

there exists an occurrence  $u$  in  $t$  and a substitution  $\sigma = (t|u)::Lhs$ , and  $t' = t[u \leftarrow \sigma(Rhs)]$ , and

### Convergence of terms in antecedent

$u_i \downarrow v_i$ , where  $1 \leq i \leq n$ , that is  $u_i$  and  $v_i$  have a common reduct.

Thus rewriting in a CTRS involves *verifying* the condition, which involves further rewriting to determine common reducts. Such verification may give rise to infinite loops if the form of the laws are left unconstrained. For this reason Kaplan [Kap84] introduced the notion of a *simplification ordering* on terms which, ensures that conditions are simpler in some sense than their consequences, thus eliminating infinite loops in the evaluation of conditions.

A *simplification ordering* is a well founded ordering (terminating)  $<$  on terms in  $T$  such that:

- *Subterm property* -  $f(\dots t \dots) > t$
- *Compatibility* - if  $t > t'$  then  $f(\dots t \dots) > f(\dots t' \dots)$ .

The following theorem, given in Kaplan [Kap84], states how simplification orderings are used to eliminate infinite loops.

### Theorem 1

Given a CTRS, and a simplification ordering  $<$ , such that for every rule  $u_1 = v_1 \wedge \dots \wedge u_n = v_n \Rightarrow Lhs \rightarrow RHS$  in  $R$ ,  $\sigma(Lhs) > \sigma(Rhs)$  and  $\sigma(Lhs) > \sigma(u_i)$ ,  $Lhs > \sigma(v_i)$  for  $i = 1$  to  $n$ , and for all substitutions  $\sigma$ , then:

1.  $\rightarrow_R$  is terminating,
2. when  $\rightarrow_R$  is confluent,  $\rightarrow_R$  is decidable.

Proof of theorem 1 can be found in [Kap84]. The rules in the CTRSs defined above are usually called positive conditional equations, since only equalities between terms are allowed in the conditions. Both Kaplan [Kap87], and Mohan and Srivas [MS87] provide treatments of CTRSs which allow inequalities in the condition parts. The approach of Mohan and Srivas is used as the basis for the relational CTRS introduced in this thesis. An overview of their approach follows.

### **Conditional rewriting systems with inequational assumptions**

Mohan and Srivas define *Equational-Inequational CTRSs* (EI-CTRSs) as follows:

An EI-CTRS is a set of rules of the form  $(u_1 = v_1 \wedge \dots \wedge u_n = v_n) \wedge (s_1 \neq r_1 \wedge \dots \wedge s_p \neq r_p) \Rightarrow Lhs \rightarrow RHS$ , where  $Var(u_i)$  is a subset of  $Var(Lhs)$  and  $Var(v_i)$  is a subset of  $Var(Lhs)$ , for  $i = 1$  to  $n$ ,  $Var(s_i)$  is a subset of  $Var(Lhs)$  and  $Var(r_i)$  is a subset of  $Var(Lhs)$ , for  $i = 1$  to  $p$ , and  $Var(Rhs)$  is a subset of  $Var(Lhs)$ .

Rewriting in an EI-CTRS, called EI-reduction, is defined as follows:

A ground term  $t$  *EI-reduces* to another term  $t'$  using an EI-CTRS  $R$ , denoted by  $t \rightarrow_R t'$ , if there is a rule  $(u_1 = v_1 \wedge \dots \wedge u_n = v_n) \wedge (s_1 \neq r_1 \wedge \dots \wedge s_p \neq r_p) \Rightarrow \text{Lhs} \rightarrow \text{Rhs}$  in  $R$  such that:

Match and replace

there exists an occurrence  $u$  in  $t$  and a substitution  $\sigma = (tlu)::\text{Lhs}$ , and  $t' = t[u \leftarrow \sigma(\text{Rhs})]$ , and

Demonstrable convergence of terms in antecedent

it can be demonstrated that  $u_i \downarrow v_i$ , where  $1 \leq i \leq n$ , in a finite number of steps, and

Demonstrable non-convergence of terms in antecedent

it can be demonstrated that  $\text{NOT}(s_i \downarrow r_i)$ , where  $1 \leq i \leq p$ , that is  $s_i$  and  $r_i$  have no common reducts, in a finite number of steps.

Demonstrable non-convergence implies that every sequence of EI-reductions from  $s_i$  to  $r_i$  terminates and the set of all reducts from  $s_i$  is disjoint from that of  $r_i$ , thus termination of EI-reduction is very desirable property of EI-CTRSs.

An EI-CTRS can be viewed as an equational theory by replacing  $\rightarrow$  by  $=$  in the rules, and taking the derived rules to be universally quantified formulas. Given an EI-CTRS  $R$ , the derived equational theory is denoted by  $E(R)$ . The model semantics for the derived equational theory is based on determining a set of inequalities between ground terms called *inequational assumptions*. These assumptions state which inequalities hold in all desirable models of the equational theory, and are appended to the equational theory, thus the models satisfying with the theory must also satisfy the inequational assumptions. Reduction in an EI-CTRS,  $R$ , is sound and complete with respect to a set of inequational assumptions,  $\mu$ , if:

*Soundness* :  $t \rightarrow_R t' \Rightarrow E(R) + {}^1\mu \models t = t'$ ,

*Completeness* :  $E(R) + \mu \models t = t' \Rightarrow t \downarrow t'$ .

The inequational assumptions made by Mohan and Srivas concern ground constructor terms, that is terms built solely from the *constructors* of a signature, where constructors are symbols for functions which create new objects of a sort in the signature. All other functions are said to be *defined functions*. All ground constructor terms are assumed to be distinct, thus the set of inequational assumptions consists of all inequalities between ground constructor terms. In general confluence is not a sufficient condition for soundness and completeness of EI-reduction, requiring another property, sufficient completeness.

Let  $s(\mu)$  be a set of ground terms such that  $\forall c_1, c_2 \in s(\mu); c_1 \neq c_2 \in \mu$ . An EI-CTRS  $R$  is sufficiently complete with respect to  $\mu$  if every ground term  $t$  has a reduct  $t' \in s(\mu)$ .

---

<sup>1</sup> + denotes set union

The following theorem establishes the importance of the sufficient completeness property.

Theorem 2

EI-rewriting is sound and complete if  $R$  is sufficiently complete with respect to  $\mu$ .

Proof of theorem 2 can be found in [MS87]. Mohan and Srivas give a number of syntactic condition on EI-CTRSs which ensures that they are sound and complete. Such conditions are based on the notion of a function being *fully defined* by an EI-CTRS.

A function is *fully defined* by an EI-CTRS  $R$  if and only if every term of the form  $f(t_1, \dots, t_n)$ , where  $t_i$  is a ground term, is reducible by  $R$  to a unique constructor term.

Proposition 1

If every defined function is fully defined by  $R$  then EI-rewriting is sound and complete with respect to  $\mu$ , where  $\mu$  is the set of all inequalities between ground constructor terms.

Proposition 2

Every defined function is fully defined by  $R$  if

EI-rewriting is ground terminating,

EI-rewriting is ground confluent, and

every ground non-constructor term is reducible.

Syntactic conditions which ensure that an EI-CTRS is sound and complete, together with the proofs of the above theorems and propositions, are given in [MS87].

## Appendix II

### Proof of completeness and soundness of sufficiently complete R-CTRSs

Proof of Proposition 3.1 in Chapter 3 [MS87]:

1.  $<_h$  is obviously a partial ordering on  $F$ , the set of function symbols in a signature.
2. The ordering  $<_h$  induces a partial, well-founded ordering,  $<_g$ , on  $T(F)$ , the set of ground  $F$ -terms, defined as follows:  $g(t) <_g h(t')$  if and only if  $g <_h h$  or  $t <_h t'$ , where  $g(t), h(t') \in T(F)$ , and  $t$  and  $t'$  are tuples of ground terms. Rewriting of a term in a R-CTRS can be represented by a tree of terms, where the root is the term from which rewriting starts, and the children of each term in the tree are the suitably instantiated terms in the antecedent and the right hand side of the consequence of a rule whose consequence has a left hand side which matches with the term. Assume that there is a term,  $t$ , for which rewriting in the R-CTRS does not terminate. This means that the tree is either of infinite width or of infinite depth. It cannot be of infinite width since each rule has a finite number of terms in its antecedent. If the tree is of infinite depth this means that there is an infinite sequence of ground terms  $t, t_1, \dots, t_n$ , starting from  $t$ , such that  $t <_g t_1 <_g \dots <_g t_n$ . Since  $<_g$  is a well-founded ordering, this is impossible. Thus there can be no infinite sequence of rewritings of a ground term in a R-CTRS.

Proof of Proposition 3.2 [MS87]:

Since constructors are distinct from all other function symbols in a signature, condition 1 of the proposition ensures that in testing for confluence one only has to consider overlaps between rules whose consequences contain left hand sides with the same outermost symbols. If a ground term  $t$ , matches with the left hand side of the consequences of two rules, then either the suitably instantiated right hand sides are equal, or at most one of the rules can be used to reduce the term, from condition 2. Thus if  $t \rightarrow t_1$  and  $t \rightarrow t_2$ , where  $t_1$  and  $t_2$  are ground terms, then either  $t_1$  is the same terms as  $t_2$ , or the reductions were carried out on distinct sub terms of  $t$ , in which case the rules when applied in opposite order to  $t_1$  and  $t_2$  result in them being reduced to the same term.

Proof of proposition 3.3:

It is first shown that one step rewriting is sound, that is, for two defined ground terms,  $t, t', t \rightarrow_{Rt'} \Rightarrow \forall M \in M_{\Sigma, E+\alpha}, M \models t = t'$ , in a R-CTRS,  $R$ , where  $M_{\Sigma, E+\alpha}$  is the class of finitely-generated models for the RS,  $E(R)$ , which satisfy the laws in  $E+\alpha$ . The proof is by induction on the number of reduction steps needed to verify the antecedents of R-CTRS rules. The induction base is taken as the case where only one reduction step is needed in order to determine the definedness of ground terms in order to carry out a one-step reduction of a term  $t$  to  $t'$  in an R-CTRS,  $R$ . In such a case, an unconditional rule,  $ok(t1) \rightarrow TT$ , where  $ok$  is the  $ok$ -predicate for the ground constructor term  $t1$ , in an R-CTRS,  $R$ , means that  $ok(t1)$  holds in all the algebraic models of the corresponding RS, denoted by  $E(R)$ , that is,  $ok(t1) \rightarrow_R TT \Rightarrow \forall M \in M_{\Sigma, E+\alpha}, M \models ok(t)$ , where  $M_{\Sigma, E+\alpha}$  is the class of finitely-generated models for the RS,  $E(R)$ , which satisfy the laws in  $E+\alpha$ . If a term rewrites in one step to  $t'$  in  $R$ , that is  $t \rightarrow_{Rt'}$ , using a rule  $t1 \rightarrow t2$  where  $t1\pi = \sigma t1$ , and  $t' = t[\pi \leftarrow \sigma t2]$ , then  $ok(\sigma t1) \rightarrow_R TT$ , and  $ok(\sigma t2) \rightarrow_R TT$ . Thus there are unconditional rules in  $R$ ,  $ok(t1) \rightarrow TT$ , and  $ok(t2) \rightarrow TT$ , which imply that  $ok(t1)$  and  $ok(t2)$  hold in all models in  $M_{\Sigma, E+\alpha}$ . Since the law  $ok(t1), ok(t2) \Rightarrow t1 = t2$  is in  $E(R)$ , and each instantiated literal,  $ok(\sigma t1)$  and  $ok(\sigma t2)$ , in the antecedent holds in all models of  $M_{\Sigma, E+\alpha}$ , then by modus ponens  $\sigma t1 = \sigma t2$  holds in all models of  $M_{\Sigma, E+\alpha}$ , thus  $t = t[\pi \leftarrow \sigma t2] = t'$  holds in all models of  $M_{\Sigma, E+\alpha}$ . If a relation  $r(\sigma t1, \dots, \sigma tn)$  rewrites to  $TT$  in one step in  $R$ , by a rule  $r(t1, \dots, tn) \rightarrow TT$ , then  $ok(\sigma t1) \rightarrow_R TT, \dots, ok(\sigma tn) \rightarrow_R TT$ , via unconditional  $ok$ -predicate rules. Thus  $ok(\sigma t1), \dots, ok(\sigma tn)$  hold in all models of  $M_{\Sigma, E+\alpha}$ . Since  $ok(t1), \dots, ok(tn) \Rightarrow r(t1, \dots, tn)$  is in  $E(R)$ , and each instantiated literal,  $ok(\sigma t1), \dots, ok(\sigma tn)$ , in the antecedent holds in all models of  $M_{\Sigma, E+\alpha}$ , then by modus ponens  $r(\sigma t1, \dots, \sigma tn)$  holds in all models of  $M_{\Sigma, E+\alpha}$ .

Suppose a ground term,  $t$  rewrites in one step to a term  $t'$ , by a rule  $(u_i = v_i)_{i=1 \dots m}, (u'_i \neq v'_i)_{i=1 \dots n}, (r_i)_{i=1 \dots o}, (\neg r'_i)_{i=1 \dots p} \Rightarrow t1 \rightarrow t2$ , where  $t1\pi = \sigma t1$ ,  $t' = t[\pi \leftarrow \sigma t2]$ . For each equality in the antecedent,  $u_i = v_i$ ,  $ok(\sigma u_i) \rightarrow^*_R TT$ ,  $ok(\sigma v_i) \rightarrow^*_R TT$ , and either  $\sigma u_i \rightarrow^*_R u$  and  $\sigma v_i \rightarrow^*_R u$ , or  $\sigma u_i \rightarrow^*_R c1$  and  $\sigma v_i \rightarrow^*_R c2$ , and  $N(c1) = N(c2)$ , where  $c1$  and  $c2$  are ground constructor terms. By the induction hypothesis,  $ok(\sigma u_i)$ ,  $ok(\sigma v_i)$ , and either  $\sigma u_i = u$ , and  $\sigma v_i = u$ , or  $\sigma u_i = c1 = c2 = \sigma v_i$ , hold in all models of  $M_{\Sigma, E+\alpha}$ , ( $N(c1) = N(c2)$  means that  $c1 = c2$  holds in all models of  $M_{\Sigma, E+\alpha}$ ) thus  $\sigma u_i = u = \sigma v_i$  holds in all models of  $M_{\Sigma, E+\alpha}$ . For each inequality,  $u'_i \neq v'_i$ ,  $ok(\sigma u'_i) \rightarrow^*_R TT$ ,  $ok(\sigma v'_i) \rightarrow^*_R TT$ , and every finite sequence of rewrites starting from  $\sigma u'_i$  and  $\sigma v'_i$  resulted in no common reducts. Since every defined term is reducible to ground constructor term, then  $\sigma u'_i \rightarrow^*_R c1$  and  $\sigma v'_i \rightarrow^*_R c2$  where  $N(c1) \neq N(c2)$ , thus  $c1 \neq c2$  holds in all models of  $M_{\Sigma, E+\alpha}$ . By

the induction hypothesis  $\sigma u_i = c_1 \neq c_2 = \sigma v_i$  holds in all models of  $M_{\Sigma, E+\alpha}$ . For each relation,  $r_i = r(t_1, \dots, t_n)$ ,  $ok(\sigma t_1) \rightarrow^*_R TT$ , ...,  $ok(\sigma t_n) \rightarrow^*_R TT$ , and  $\sigma t_1 \rightarrow^*_R t_1'$  or  $\sigma t_1 \rightarrow^*_R c_1$ ,  $\sigma t_1' \rightarrow^*_R c_1'$  and  $N(c_1) = N(c_1')$ , and ...,  $\sigma t_n \rightarrow^*_R t_n'$  or  $\sigma t_n \rightarrow^*_R c_n$ ,  $\sigma t_n' \rightarrow^*_R c_n'$  and  $N(c_n) = N(c_n')$ , and  $r(t_1', \dots, t_n') \rightarrow_R TT$ . By the induction hypothesis  $t_1 = t_1'$ , ...,  $t_n = t_n'$ , and  $r(t_1', \dots, t_n')$  holds in all models of  $M_{\Sigma, E+\alpha}$ . Thus  $r(t_1, \dots, t_n)$  holds in all models of  $M_{\Sigma, E+\alpha}$ . For each n-relation  $\sim r_i = r'(s_1, \dots, s_n)$ ,  $ok(\sigma s_1) \rightarrow^*_R TT$ , ...,  $ok(\sigma s_n) \rightarrow^*_R TT$  and thus  $\sigma s_1 \rightarrow^*_R c_1, \dots, \sigma s_n \rightarrow^*_R c_n$ , where  $c_i$  ( $1 \leq i \leq n$ ) is a ground constructor term, and  $NOT(r'(c_1, \dots, c_n) \rightarrow TT)$ , thus  $\sim r'(c_1, \dots, c_n)$  is a n-relation assumption. By the induction hypothesis  $t_1 = c_1$ , ...,  $t_n = c_n$ , and  $\sim r(c_1, \dots, c_n)$  holds in all models of  $M_{\Sigma, E+\alpha}$ , thus  $\sim r(t_1, \dots, t_n)$  holds in all models of  $M_{\Sigma, E+\alpha}$ . Since the rule  $(u_i = v_i)_{i=1 \dots m}, (u_i \neq v_i)_{i=1 \dots n}, (r_i)_{i=1 \dots o}, (\sim r_i)_{i=1 \dots p} \Rightarrow t_1 = t_2$  is in  $E(R)$ , and the instantiated antecedent holds in all models of , then, by modus ponens,  $\sigma t_1 = \sigma t_2$  holds in all models of  $M_{\Sigma, E+\alpha}$ , and thus  $t = t[\pi \leftarrow \sigma t_2] = t'$  holds in all models of  $M_{\Sigma, E+\alpha}$ .

It is now shown that for any defined ground term,  $t$ ,  $\forall M \in M_{\Sigma, E+\alpha}, M \models t = c$ , where  $c$  is a ground constructor term and  $t \rightarrow^*_R c$ . This done by induction on the depth of the rewrite relation from a defined ground term to a ground constructor term. The base case is the case where the depth is 0.

Consider the case where a defined ground term  $t$  rewrites to a ground constructor term,  $t \rightarrow_{R'} t' \rightarrow^*_R c$ . By the induction hypothesis,  $t' \rightarrow^*_R c \Rightarrow \forall M \in M_{\Sigma, E+\alpha}, M \models t' = c$ , and by the soundness of one step rewriting  $t \rightarrow_{R'} t' \Rightarrow \forall M \in M_{\Sigma, E+\alpha}, M \models t = t'$ . Thus  $t \rightarrow_{R'} t' \rightarrow^*_R c \Rightarrow \forall M \in M_{\Sigma, E+\alpha}, M \models t = c$ .

A homomorphism,  $h$ , from the set of defined ground terms to elements in the carrier sets of the models in  $M_{\Sigma, E+\alpha}$  can be defined as follows: for a model  $M$  in  $M_{\Sigma, E+\alpha}$ ,  $h(t) = c^M$ , where  $t \rightarrow^*_R c$ . This homomorphism is well-defined from above, and is unique since the models in  $M_{\Sigma, E+\alpha}$  are finitely generated. Thus ground rewriting in is sound and complete.

# Errata

This section lists the errors identified in this thesis. Page numbers are preceded by "p." followed by line numbers. A negative line number indicates that counting is from the bottom up, for example line -3 is the third line from the bottom of the page. Omissions are underlined>.

## CHAPTER 0

p.1, line 14

"nfeasible" should be "infeasible".

p.2, line -5

The reference "FP" should be "FP86".

p. 3, line 16

"guaranted" should be "guaranteed".

p.4, line -7

The reference "[Woo78]" should read "[Woo88]".

line -3

The reference "[YC78]" should read "[YC79]".

p.7, line 9

The reference "Ros77" should read "Ross77".

line 10

"diagraming" should be "diagramming".

line 11

"diagram" should be "diagrams".

## CHAPTER 1

p. 14, line -3

Delete "consists" after "cust\_order".

p. 15, line -8

Should read "there are sufficient parts".

p. 21, line 4

Should read "control aspects of DFDs".

line -12

Should read "In a TS data flows can be combined".

p. 28, lines 10, 11

Should read "Hatley's extensions ... Furthermore, they provide tools".

p. 29, lines -5, -4

Should read "The types of information".

p. 35, lines 5,7

Should read "A Petri net ... is a useful tool for".

## CHAPTER 2

p. 42, line -3

"if the book is a copy of the library" should read "if the book belongs to the library".

p. 50, line -17

Replace "CheckouBook" by "CheckoutBook"

p.64, line -9

The reference "Hat88" should read "HP87".

line -6

The reference "KK88" should read "KKZ88".

p. 67, line -17

The reference "KK88" should read "KKZ88".

p. 69, line 5

Should read "processes which the outgoing data flows".

## CHAPTER 3

p. 73, line 6

Should read "to support the range".

p. 76, line -8

$V_A(r(t_1, \dots, t_n)) = (V_A(t_1) \dots V_A(t_n))$  should read  $V_A(r(t_1, \dots, t_n)) = TT$ , where TT is a special value indicating the validity (equivalently, definedness) of the predicate. Relations are



associated with default semantics where it is assumed that if an interpretation does not evaluate to TT then the relation does not hold (that is, it is undefined).

p. 77, line 15

Should read " and  $D_A$  is the".

p.78, line7

" $r_{di}(w) \in r_{diA}$ " should read " $\forall_A(r_{di}(w)) = TT$ ".

line 8

" $r'_{ei}(w) \in r'_{eiA}$ " should read " $\forall_A(r'_{ei}(w)) = TT$ ".

p. 81, line 12

Should read "Example 3.4 defines sets of natural numbers"

p.82, line -19

Delete law S15. This situation can be handled using a normalizing function which gets rid of any duplicates.

p. 83, line 5

insert below line 5 the following:

**ok-predicate**

okset : set

p. 84, line 23

Delete law S15 (see erratta for p. 82, line -19 above).

#### CHAPTER 4

p. 96, line 8

Remove "." after "*correct*".

p. 111, line -25

Insert below line -25 the following law:

$getsubstructs(\emptyset) = \emptyset$

(This error also causes an obvious change in the law numbering).

p. 116, line 21

Delete law PS25.

CHAPTER 5

p. 126, line -2

"Example 2.7" should be "Example 2.6".

p. 132, line 7

"observeration" should read "observation".

line 13

"add(e,deleteq(q))" should read "addq(e,deleteq(q))"

line -6

Should read "access functions ... are determined".

p. 134, line 12

The reference Hoa85 should read Hoa78.

line 13

The reference Lam88 should read AL88.

line 17

The reference AGR87 should read AGR88.

p. 138, line 5

The form of the law is actually more general than is expressed. Any events which affect a disjoint subset of states may be synchronized.

p. 139, line 18

Insert below line 18 the following law:

3.  $\text{getfinesrec}(\text{emptylist}, t1) = \text{emptylist}$

line 28

Insert below line 28 the following law:

2.  $\text{sum}(\text{emptylist}) = 0$

line -2

Should read "Receive1p6(borrid)".

p. 140, line 21

Should read "Receive2p9(borrid)".

p. 142, line -1, -2

Replace laws 1. and 2. by  $q == \text{ADD}(e) ==> \text{addq}(e,q)$

p. 144, line 9

Should read "can determine wich pairs".

p. 146, 19

Add "; ckmess:asynch(checkout\_mess)" to line  
line -13

Insert ", ckmess" after "ckinfo".

p. 148, line -11

Should read "Such an automated system".

line -9

Should read "This would involve".

line -4

Should read "as far as their inputs".

p. 149, line 15

Should read "State transitions".

## CHAPTER 6

p. 154, Figure 6.2

Redirect the arrow emanating from the state ACCELERATING and directed to the state RUNNING3 (labeled brake\_on), to the state BRAKING.

Insert an arrow from CRUISING to RUNNING2 labeled less\_30.

Insert an arrow from BRAKING to IDLE labeled engine\_on/off.

Insert an arrow from BRAKING to RUNNING2 labeled cruise\_on/off.

Insert an arrow from RUNNING2 to IDLE labeled engine\_on/off.

Insert an arrow from CRUISING to IDLE labeled engine\_on/off.

p. 156, line 15

Delete "+ P1substate ".

p. 157, line 2

Replace "Siganture" by "Signature".

p. 158, line -9

Replace "Siganture" by "Signature".

p. 162, line 18

Add "; + Asynch(Number) +" to line.

line 30

Replace "<\_,\_,\_,\_>" by "<\_,\_,\_,\_,\_,\_>", and replace "asynch2" by "asynch1".

line 36

Replace "as1: asynch1, as2:asynch2" by "as1, as2: asynch1".

p. 164, line 4

Add "+ Syslabel" to line.

line 11

Add "; A1, A2: syslabel" to line.

p. 165, lines 11 to 20

Delete laws 8 and 9.

line 16

Should read "p5==Sendp5(pos)".

p. 167, line -2, -4

Replace "Readprate" by "Readprate(c)".

line -6, -8

Replace "Readchrate" by "Readchrate(c)".

p. 169

Laws 38 to 41 handle only some cases in which parallel events can occur. In general, events which affect mutually exclusive parts of an application's state can be carried out in parallel.

## CHAPTER 7

p. 196, line -2

Should read "characterizes the behaviour of the ExtDFD".

p. 197, line -15

Should read " Earlier work ... provides".

lines -8, -9

Should read "which stores the representations generated by".

p. 199, line -5

The reference "Doc88" should read "Doc87".

p. 211, line 14

Should read "is a very desirable property".

lines 21, 22

Should read "satisfying the theory".

Addition to bibliography

[San88] Sanella, D., 'A Survey of Formal Software Development Methods', Expository Report, Department of Computer Science, University of Edinburgh, July 1988.