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

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 concensus 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:

SP0 --> SP1 --> ... --> S

where SP0 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 SP1 can be thought of as an implementation of SP0. If each individual step can be proved correct, that is, if it can be proved that SPi implements SPi-1, then S itself is guaranteed to be correct with respect to SP0. 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 SPi 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 depenedent (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.