1

Introduction

1.1. Programming Language and Software Development

Programming is more than *translating* what we want the computer to do into a computer program; it involves the whole process of *determining* what basic information the computer needs to possess, *determining* what we want the computer to do, *transforming* the specification into a program and *evaluating* the specification and the implementation. Implementation is only a small step in the software development cycle. Implementation in the JSD software development process, for example, only contributes to one of the six development steps [Jackson83]. What is more, some authors claim that the hard thing about software construction is deciding what one wants to say, not saying it (cf [Brooks86, Sommerville89]).

Although historically programming languages have been concerned with implementation, some kind of programming language is essential as the fundamental communication medium between the participants in the software development process. It is difficult to discount the role of research in high-level languages in solving the

essence of the problem of complex software development [Harel92]. Research on programming language design should focus not only on implementation issues but also on the relationship between a programming language and the whole software development cycle.

Perhaps we can learn a lesson from the development of object-oriented programming (OOP). The idea of OOP (viz programming as object-based modelling) was first brought out by Simula, and can be traced back to the sixties [Naur63, BDMN79]. It was not widely known until the early 80's, when the object-oriented language Smalltalk [GR83] and later C++ [Stroustrup86] were launched. They triggered lots of interest in the programming community. "Suddenly everybody is using it, but with such a range of radically different meanings that no one seems to know exactly what the other is saying", Cox commented [Cox86]. It is difficult to define OOP. Wegner attempted to define it by "object-oriented = objects + classes + inheritance" [Wegner87], but this definition fails to address those object-oriented languages that have no classes. In some object-oriented languages, properties of objects are inherited from other individual objects rather than from classes. Hence, Nelson modifies the definition of OOP to "object-oriented = (objects + classes + inheritance) OR (objects + delegation)" [Nelson90]. Because of the diversity of practice in OOP, Nelson claims that we are creating an object-oriented "Tower of Babel" [Nelson91]. When discussing OOP, many, such as [SB86] and [DT88], refer to techniques like inheritance, message passing and data encapsulation but neglect the object modelling principle. Noticeably from the late 80's, the underlying programming methodology of object-oriented languages is emphasised. For instances, Booch discusses object-oriented *design* (OOD) rather than what an object-oriented *language* can do [Booch91]; Meyer rightly states that the first principle of object-oriented program design is "ask not first what the system does: ask WHAT it does it to!" [Meyer88]. The history of OOP indicates that it is best to understand the application

software design issues in order to give a clear direction to the development of a programming paradigm.

Two important aspects of software design are: it is a sort of design and it is dealing with computation. One difficulty in design is that the software requirement is often unclear or a specification may not be available because the domain of application is poorly understood [Trenouth91]. As Whitefield puts it: "*design is more of a dialectic between the generation of possible solutions and the discovery of the constraints operating on the solution space*" [Whitefield89]. Also Fisher and Boecker state that "*design is best understood as an incremental activity that makes use of existing prototypical solutions to gain a deeper understanding of a problem*" [FB83]. These motivate the exploratory software development process.

One of the earliest proponents of exploratory software development was Sheil. By showing some cases in which any attempt to obtain an exact specification from the client is bound to fail (because the client does not know and cannot anticipate exactly what is required), Sheil concludes that "*no amount of interrogation of the client or paper exercises will answer these questions; one just has to try some designs to see what works*" [Sheil83]. This statement characterises exploratory software development.

One of the examples used in this thesis can illustrate this. During the simulation of a train departure protocol, we discover that it is possible for the train to move while a passenger has opened a door and is attempting to board the train. It is a dangerous act. When the departure protocol is examined, the protocol between the driver, station-master and guard works well in isolation, the passengers also make correct decisions for alighting and boarding. A problem arises only when these two sets of protocol interact. It is therefore difficult to foresee the problem before simulation.

Since there may not be any expected problematic areas in the protocol, it would require a formidable analysis in order to understand the source of the problem before

any remedy can be suggested. In the train example, for instance, a possible remedy might leave the protocol of both the station-master and the passenger unchanged, but add locks to the doors. Because 'understanding' seems to be the bottle-neck of software design, exploratory software development put its emphasis on 'understanding'. Exploratory software development employs a run-understand-debug-edit (RUDE) cycle [Partridge86, PW87]. In this RUDE cycle, experiments are conducted and observations are made in order to understand the behaviour of the software prototype. A main objective in programming paradigm development is therefore to shorten the observation and understanding processes.

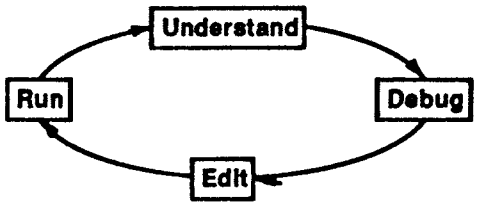

Figure 1.1: The Exploratory Software Development (RUDE) Cycle

There is a common characteristic between experimentation and computation – they are both state-based. Experimentation gives rise to a state-based interpretation of the prototype – what is the state of the prototype when something becomes the input to the experiment? Computation is also state-based. Computation concerns states and the interactions between them. It is natural, therefore, to develop a state-based exploratory programming paradigm.

In addition to being state-based, an exploratory programming paradigm should respect four principles for exploratory software development laid out by Trenouth in [Trenouth91]. Exploratory software must always be: continuously executable, easily extendible, conveniently explorable and usefully explainable. In this thesis, we consider a new approach to programming, and seek to justify the claim that this programming paradigm is suitable for exploratory software development.

1.2. Dependency and Observation

There is a close correspondence between observation and dependency. The reason that we do experiments is because we believe that the input/output relationship of an experiment will be consistent throughout different observations. Therefore, the relationship can be identified or verified through experiments.

The properties we might be interested in changing and observing in an experiment characterise an object. Our approach to software experiment is to capture the dependency information of the properties within an object and between objects by means of definitions. This is why we have called our approach *definitive programming*, meaning *definition-based* programming. A set of definitions, or what we have called a *definitive script*, then records the current state of experiment. In order to change the current state (or to perform an experiment), part of the definitive script is to be redefined. By a definition we mean a formula of the form:

$$x = f(y_1, y_2, \ldots)$$

The value of the variable x is always equal to the evaluation of the formula $f(y_1, y_2, \ldots)$. By defining variables using formulae rather than explicit values, the data dependency of the variables is recorded. The value of x depends upon the values of $y_1, y_2, \ldots$ where $y_1, y_2, \ldots$ may themselves be functionally dependent upon other variables.

A definitive script of this nature is restrictive; it can only capture uni-directional relationships. That is, in a set of definitions, no circular dependency is allowed. On the other hand, this generally guarantees that a set of definitions can be evaluated. Moreover, we believe that the study of 'definitions' will establish a better foundation for more complex relationships such as constraints. This is evident from the fact that some constraint systems, such as ThingLab, Procol and RL/1 [BD86, MBF89, LV91, van Denneheuvel91, CP87], define constraints explicitly or implicitly by sets of methods that can be invoked to satisfy the constraints. Each of these methods serves a

similar function to a definition in our sense. By the appropriate selection of methods, one from each set, the constraints are resolved. This process can be understood as establishing and evaluating a definitive script.

1.3. Motivating Ideas

It is our belief that there is a way of programming that is rooted in modelling dependency between observations. This belief is supported by the following evidence:

1) Research done on definition-based systems

Several definitive notations have been designed and implemented. A definitive notation is a programming notation that can be used for formulating a set of definitions. It is described as a "programming notation" rather than a "programming language" because it only represents part of the information needed for general-purpose programming. DoNaLD and ARCA are two examples of definitive notations. DoNaLD is a definitive notation for 2-D line drawing [BABH86] and ARCA is a definitive notation for displaying and manipulating a class of combinatorial diagrams [Beynon86a]. The data types in DoNaLD and ARCA are application-oriented. For example, DoNaLD has *shape*, *point*, *line* and *circle* whereas ARCA has *diagram*, *colour* and *vertex*.

Definitive systems such as the DoNaLD system ease our understanding and observation of the application in at least two ways. Firstly, the definitive notation is closer to the application than a general-purpose language. The gap in translating between the programming model and the real world is narrowed. Secondly, definitive systems provide immediate feedback. If a box is defined in DoNaLD by lines joining its four corners and the positions of three corners are defined relative to the south-west corner (box/SW), repositioning of the box by redefining the DoNaLD variable box/SW will have an immediate effect on the positions of the four lines on the screen. A short feedback cycle allows a large number of experiments to

be done on the current state in a short time. Later in the thesis, it will be shown that all the qualities of an exploratory programming paradigm – continuous executability, extendibility, explorability and explainability – are present in definitive systems.

In addition to the research in definitive state representation, methods of specifying transitions between states are also investigated. The EDEN definitive language[1] is the contribution of Edward Yung to specifying definitive state transitions in a sequential fashion [Yung89]; [Slade89] provides a thorough study of the LSD specification language for concurrent systems modelling and the ADM programming language for the implementation of LSD. These show that definitive programming is capable of specifying general state-transition models. Hence, definitive programming is an all-purpose programming paradigm which captures data dependency.

2) *Connections between definitive programming and other programming paradigms*

We are actively developing an agent-oriented definitive system. This kind of programming partitions a system into sub-systems according to the agents involved. Every agent has a knowledge of its environment and has its own variables. All these are represented by definitions. An agent will act upon its own understanding of the environment by typically redefining some variables. In this programming paradigm, programming using definitive state representation is similar to functional programming and the agent partitioning is similar to object-oriented decomposition.

---

[1] The term *definitive language* is used to refer to any programming language in which we can formulate definitive scripts.

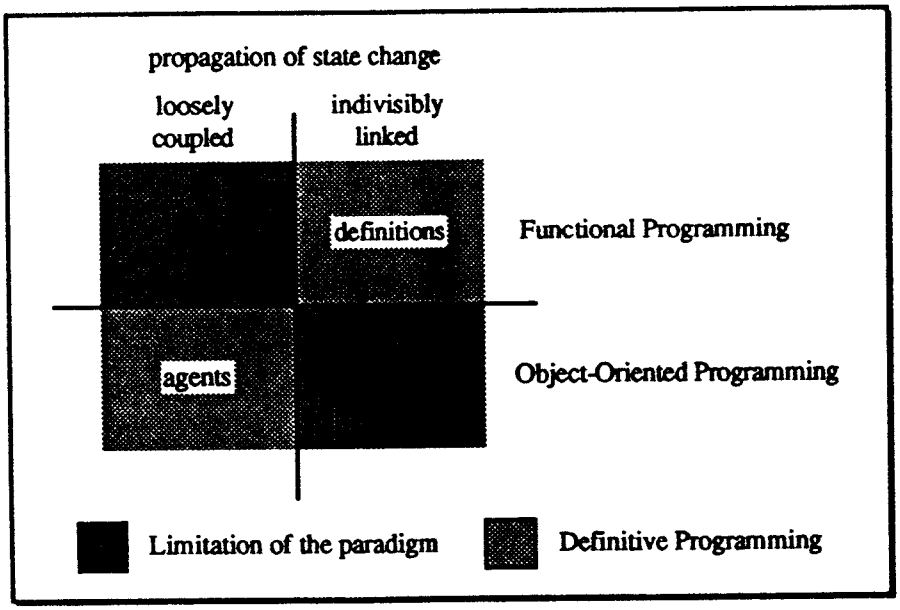

Figure 1.2: Programming Paradigms vs Propagation of State Change

Definitions and agents are associated with two complementary kinds of propagation of state change. Definitive scripts are associated with indivisible propagation (e.g. as in a mechanical linkage) and agents with loosely coupled propagation (e.g. as in asynchronous communication), as illustrated in Figure 1.2.

In a functional program, everything is a function. Writing a functional program is expressing the input/output relationship of an application in functional terms. There is no concept of state in functional programming. This means that the relations expressed in a functional program are of a static nature.

The primary use of functional abstractions in a historical sense is to represent relationships between observations made in the same context. These relationships are associated with modelling indivisible propagation of state change – they correspond to 'definitions' in Figure 1.2. A functional abstraction is not the most appropriate way to model propagation of state change that is loosely coupled, such as commonly arises in interactive programming (see §2.2.2 and §7.1).

Object-oriented programming, in contrast, models propagation of state change in a dynamic fashion through explicit communication between objects. When an

operation performed on one object requires corresponding operations to other objects, this is modelled by means of message passing.

Object-oriented programming simplifies the representation of loosely coupled state changes by reducing the problem of programming a system to that of programming the objects within the system. This is closely connected with the role of agents as in Figure 1.2. Object-oriented programming is less successful in representing indivisible propagation of state change such as is required for synchronisation in concurrent object-oriented models [Baldwin87].

Figure 1.2 indicates that a definitive program which combines scripts of definitions and agent specification exploits the best qualities of the functional and object-oriented paradigms.

3) *Broad programming practice*

There are few programming paradigms in use in the commercial world but there are many programming paradigms used or under development in research laboratories. Procedural languages dominate commercial computing, but increasing program complexity and improved parallel hardware technology lead us to question their suitability for applications in the future [Turner83, Landin66, Baldwin87]. Many programming paradigms are invented. But will any one of them be *the* future programming paradigm, if there is one?

Baldwin, Hillis and Steele have argued that data parallelism is the key to maximising the utility of parallel hardware [Baldwin87, HS78]. Data parallelism is closely connected with the identification of data dependency. Baldwin compares several programming paradigms with respect to their suitability for multi-processor machines [Baldwin87]. By his analysis, neither conventional procedural programming, object-oriented programming, functional programming nor logic

programming is good for specifying data dependency. In contrast, definitive programming explicitly describes dependency relationships between data.

Baldwin suggests that constraint programming may be the best candidate for parallel programming. However, constraint satisfaction is generally recognised to be a time-consuming exercise. Most of the existing constraint systems either accept only a restricted kind of constraint or use constraint management methods that are given explicitly by the programmer. There are clear connections between definitive scripts and systems of constraints (cf §1.2). Definitive programming may be an appropriate compromise where efficiency, expressive power and convenience are concerned.

In [Smith87], Smith argues that the relation between a program and the outside world should guide the development of new foundations for programming – the traditional account of the semantics of programs is not adequate. Programming paradigm development should also focus on "the semantics of the semantics" of programs. Definitive programming, a paradigm founded on describing the relationship between observations obtained from experiment, may address the essence of the problem [Beynon92, BR92].

The above points indicate that the most widely known and well established programming paradigms have significant limitations. It is entirely possible that a new programming paradigm which is based on modelling dependency would shake the whole programming world.

4) *Use of dependency in current systems*

Although dependency is rarely formally studied, it is not difficult to identify systems which make heavy use of dependency. The most prominent one is the famous spreadsheet. An electronic spreadsheet is a table of cells in which the relationships between the cells are explicitly written down for the calculation of the

values of the cells. A graphical user interface tool is another example. A stylesheet in word processors is yet another. Spreadsheets and style-based word processors may be the most commonly used software tools, and most probably the explicit use of dependency is a crucial reason for their success.

1.4. What Has to be Done

Our research in definitive programming can be logically divided into two sub-areas: representation of states and modelling of transitions. In the area of representation of states, definitive notations are designed and implemented to evaluate and explore the advantages and disadvantages of definitions. In the area of modelling transitions, higher-level specification and control languages are designed to govern the transitions of states. Furthermore, a theoretical framework needs to be developed. Practical examples are also required in order to evaluate the research in every stage.

1.4.1. Brief History of Definitive Programming Research

It would be helpful to present a short history of the research in definitive programming to give some flavour of the scope the research involved. Whilst the definitive paradigm has a relatively short history, definitive principles have been used informally to maintain relationships between values of variables for a very long time. Early examples include the specification of machining sequences in numerical machine tools in APT [ITT67], Wyvill's interactive graphics language [Wyvill75] and the electronic spreadsheets of the early 70's. The first paper to describe the abstract concept of a definitive notation was [Beynon85], published in 1985. Independent definitive notations and an agent privilege specification language were then developed in parallel.

In the area of developing definitive notations, there were only two definitive notations designed before 1987. They were ARCA and DoNaLD. Amongst them only ARCA was implemented. One reason for the slow development of definitive notations was that implementing a definitive notation was a time-consuming job. Since the

design and implementation of the definitive language EDEN in 1987, the implementation job of definitive notations is greatly eased. The name EDEN is, in fact, an abbreviation of "an Engine for DEfinitive Notations". The implementation of a definitive notation becomes a task of producing a translator from the specific notation to EDEN and writing an associated EDEN library for the simulation of the data types and underlying algebra in the notation. This is a much simpler job than writing the evaluation engine for a new family of definitions. However, definitive notations were still running in isolated environments. Within a session, one could only interact with a single definitive notation other than EDEN.

Much of the development of an agent privilege specification language was carried out by Mike Slade. The LSD notation, first defined in 1986 [Beynon86b] and subsequently modified in 1989 [Slade89], was a result of this research. LSD is a specification language for the behaviour of multi-agent reactive systems. An LSD specification is not executable; it has to be transformed manually into the ADM language before execution [BSY88]. The ADM definitive language was designed both for the interpretation of LSD and to give a more satisfactory abstract account of the hybrid programming paradigm used in EDEN, and subsequently implemented for the former role by Slade. The main problem with ADM is its limited data types. This restricts the usefulness of ADM and hence hinders the development of LSD.

1.4.2. The Future

Ideally, the ultimate system will be very large. In that system, many definitive notations will describe different parts of a program. Possibly these definitive notations will be defined within a more powerful and general language. The LSD specification language has plenty of scope for improvement as well. Ideally, we should like to be able to specify the methods of conflict resolution for agent actions in LSD. Also, the transformation from LSD to an executable program should be simpler. A possible solution is the use of hidden text to annotate an LSD specification. By the addition of

simulation decisions in this way, the annotated LSD specification will be executable, in principle. The ability to specify conflict resolution also implies the ability to model higher level data dependencies, such as constraints.

Although we have some hints of what this ultimate system will be like, it still seems to be a long way before a preliminary version can be prototyped. More research on the LSD notation itself, the transformation process and the linkage between definitive notations has to be done beforehand.

1.5. Contribution of This Thesis

This thesis is not intended to overcome all the obstacles to the ultimate system. Its main objective is to merge previous research efforts in definitive programming around a unified theme. It becomes apparent that 'Programming as Modelling' is one of the major contributions of definitive programming to the software development process [BRY90, BBY92, BY92], and it is in this context that previous researches merge. Definitive notations, by having their specific domains and being definitive in nature, are suitable for modelling states of the real world, while the agent privileges described by LSD are suitable for modelling the dynamic behaviour of the real world. Because of its strong modelling orientation, definitive programming satisfies the requirements for exploratory programming – it is state-based, continuously executable, easily extendible, conveniently explorable and usefully explainable. Apart from abstract discussion of the potential of definitive programming, my practical contribution is to combine several definitive notations, and to some extent LSD, into a single programming environment based on definitions. This brings us practically a step forward to our vision of programming.

This thesis will describe both practical work done and the philosophical advancement in definitive programming. The areas covered are:

Practical work

1. *In the area of definitive representation of state:* A definitive system, Scout, in which several definitive notations can be used cohesively, is designed and implemented. This involves modifications to the existing implementation of definitive notations as well as developing a general interface program to the window system.

2. *In the area of transition of states:* Firstly, the source of definitions of the system is widened. Originally, the only way of introducing new definitions was via textual input; now definitions can be generated by mouse events as well as generated by the system itself. Additional sources of input allow the system to respond to richer form of interaction with its environment. Secondly, an ADM-to-EDEN translator is prototyped. Since ADM is an implementation language for LSD and EDEN is the underlying language of the Scout system, the ADM-to-EDEN practically links the previous research works together. This work also indicates how the practical power of EDEN can, in principle, be expressed in the purer programming paradigm of the ADM where *all changes* of state are represented by redefinitions.

3. *In the understanding of definitive notations:* The definitive notation Admira is prototyped. Since the evaluation mechanism of Admira makes use of the functional programming system Miranda [Turner86], studying Admira is a means to understand the relationship between definitive programming and functional programming.

Philosophical advancement

This thesis:

1. links up much previous work in the entire definitive research programme. Sets of definitions to define state and agent-oriented programs to describe transition both have a strong modelling foundation. This serves as the link.

2. develops the idea of using definitions for modelling the real world. By means of definitions, the gap between a computer programming model and the real world is narrowed.

3. advocates the new concept that definitive programming is good for exploratory software development.

4. evaluates, by means of illustrative examples, the advantages and limitations of current definitive system. In the course of this discussion, it will be demonstrated that definitive notations can play a significant part in general-purpose definitive programming.

1.6. Outline of the Thesis

This thesis is organised so as to defend the claim that definitive programming is a programming paradigm that is suitable for exploratory software development. In the next chapter, the heart of definitive programming – the Definition-based State-Transition (DST) model – is introduced. The abstract virtues of the DST model will also be explained. Chapter 3 shows that some commonly used software tools are already using concepts and techniques close to our notion of definitions. Chapter 4 describes my design and implementation of the Scout definitive notation. Scout is a definitive notation for describing screen layout. By means of an illustrative example, Chapter 5 demonstrates how the Scout notation assists exploratory screen layout design. Chapter 6 describes my work on integrating several definitive notations. Through integration of definitive notations, we can broaden the domain for our exploration. Chapter 7 stands between the discussion of definitive representation of state and specification of transitions over such a representation. It discusses the ways in which the power of single-agent definitive systems, such as Scout, may be enhanced. The discussion prompts us to introduce more general agents into definitive systems. Chapter 8 describes an agent-oriented specification language (LSD). By

describing a software tool for assisting the implementation of LSD and giving practical suggestions for improving LSD, this chapter advocates that agent-oriented definitive programming can not only deal with all-purpose programming but is also suitable for exploratory development of software. Chapter 9 summarises the thesis and concludes that definitive programming is a good paradigm for exploratory programming.