9

Summary and Conclusion

This thesis aims to justify the claim that definitive programming is a good paradigm for exploratory programming. The exploratory software development method is employed when the specification of a problem is not known or unclear. Exploratory software development employs a Run-Understand-Debug-Edit (RUDE) cycle. To make exploratory software development efficient, exploratory software should be continuously executable, easily extendible, conveniently explorable and usefully explainable. Many of our previous publications encourage us to consider definitive programming as a paradigm for exploratory programming. This thesis discusses in detail the relationship between definitive programming and exploratory software development.

The Definition-based State Transition (DST) model is the essence of the definitive paradigm. The DST model is a state-transition model in which a state is represented by a set of definitions and the transitions are represented by redefinitions. From the operational point of view, a definitive state provides data dependency

information and methods of maintaining the state. Hence, definitive programs should be, in principle, highly parallelisable. From the semantic point of view, definitive programming is different from conventional programming in that a variable stores a formula rather than a concrete value. A definition denotes a value (the evaluation of the formula), gives meaning to the value (the formula) and specifies the relationships between the variables in the formula and the variable that appears in the left-hand-side of the definition. Therefore, the overall state change can be predicted when some of the variables get redefined. This means that the potential changes to the state are captured in the definition of the state itself. This makes a definitive paradigm useful for modelling applications.

Many common software tools use concepts similar to the definitive principle. This indicates that definitive programming has great potential for use in developing realistic applications. The Jugs screen layout design exercise further strengthens the belief that definitive notations are particular well-suited for design applications. Definitive programming uses domain-specific underlying algebra, allows flexible definition arrangement and integrates the design and simulation processes. All these features enable convenient modelling of states and redesigning of the models.

The relationship between the value of a variable and the values of variables on which it depends is analogous to a mechanical linkage. There is inseparable propagation of value changes within a single transition of state. Therefore, definitive notations such as Scout provide a neat way of separating the presentation of state from the definition of the state. This allows the programmer to develop the definitive state model without bothering too much about the issues concerning the realisation of states.

Each definitive notation is specifically designed for a class of applications. This is an advantage where modelling is concerned. On the other hand, it is a disadvantage with respect of general-purpose programming. To define a state, we need several

definitive notations to describe different aspects of that state. So there is a need to integrate definitive notations.

The Scout project was the response to this demand for integration. The Scout project addresses this problem in two ways: through the design and implementation of the Scout notation and through deriving a scheme for communication between definitive variables. The Scout project can be viewed as a constructive solution to the integration problem in two ways:

- While DoNaLD and ARCA concentrate on how to define a model, Scout concentrates on how to present the model. When generating a screen display is the common goal for different definitive notations, Scout can be the link between those definitive notations.

- A bridging definition is a channel through which definitive notations can communicate. This generally establishes a connection between different definitive notations independent of the assumption that screen display is the common ground.

The representation of states by sets of definitions must be complemented by some way of specifying state transitions. We have considered an agent-oriented definition-based specification language – LSD. Clearly, LSD adopts modelling as a programming strategy – it models the perception and reaction of the agents involved. This strategy matches the rich modelling property of definitions in specifying states.

In connection with exploratory software development, the modelling orientation of definitive programming surely makes a program highly explainable. This alone is not enough to justify definitive programming as exploratory programming paradigm. Other principles of exploratory software development must also be observed. These relate to:

1) how easy is it to change a specification,

2) how easy is it to transform a specification into an executable program and

3) how efficient is the execution.

In responding to these issues, we would argue that:

1) Changing a definition or an agent specification is, in principle, simple. The most difficult aspect is when a change in the specification requires an extension to the underlying algebra or even a new definitive notation. However, this thesis has shown a systematic way of implementing a definitive notation, and this scheme is proven to be simple by our experience in implementing existing definitive notations.

2) Transforming definitions in other definitive notations into executable (EDEN) definitions can be done on-line and is a fully-automatic process; transforming an LSD specification into an executable program is non-trivial. But with the improvement of notations suggested in chapter 8, hidden-text annotation and the ADM-to-EDEN translator, it is reasonable to believe that transformation of LSD into an executable program can be close to fully, if not completely, automated.

3) Many publications mentioned in this thesis have already discussed the potential for concurrency in definitive programming. On this basis, we can be confident that the execution of definitive program can be highly efficient.

Current definitive systems are far from perfect, but many suggestions in the thesis have yet to be implemented, and the evidence presented in this thesis is sufficient to justify the assertion that "definitive programming is a good paradigm for exploratory programming".