Seminar Th4:

Definitive Methods for Programming and Parallel Programming

Th4.1: Prototyping from definitive specifications

In the definitive paradigm, there are many types of activities but only one type of operation – redefinition. A redefinition may produce both the effect of i) changing the model and ii) testing (or simulating) the model. For example, in the Jugs program, redefining widthA changes the layout design but redefining contA is part of the simulation process. This shows that definitive programming encapsulates design and simulation in the same process; when the programmer is satisfied with the design, a program is ready for use [CW89].

Is it a good idea to merge design and simulation? In other words, should the user be allowed to exert such power to change the design of a program? Although there is no distinction between data and program in conventional computer architecture, a program in a conventional high-level programming languages is not usually changed during its execution. The simulation referred to here is part of the software development process. In a software developer's role, one has the right to modify one's software. But to a user, the development of the software is supposedly frozen. Only certain ways of interaction to the script is expected. Therefore, it is more appropriate to ask whether there are any convenient ways of restricting the user's power to modify a definitive script.

There is a danger in this discussion of giving the reader an impression that a definitive script is a program. Since a set of definitions is meant to model a state but not to handle inputs and the transitions to the state according to the inputs, we should not generally treat a definitive script as a program, at least not a complete program. However, since a definitive state may contain complex relationship between variables, a change in the value of a variable may induce a large amount of value changes to other variables and to the output. For some applications where dynamic change of relationship between variables is not required, for example the vehicle cruise control simulation example, simulating the application is essentially changing the input parameters in the conventional programming sense. For this kind of application, a definitive script may be considered as a program with a different input specification. While the expected program may accept a number entered in a particular dialog box, the definitive script accepts a redefinition of a variable to that number.

In order to turn a definitive script into a program, an interface transforming the user input into redefinitions is needed. A few possibilities are explored in this thesis:
1. Extend definitive notations to a complete language with transition control. Lsd is one such language.
2. Write a simple interface program which fulfils the required input specification and generates appropriate redefinitions for the definition evaluator. Some software tools such as tooltool [Musciano88] exist to assist the creation of this interface. The current Scout system has also been extended in such a way that user-input can be captured and transformed into required definitions.
3. Transform the definitive script into a conventional language, where the transformed program only allows changes to certain variables. Since conventional programming separates the development of a program from its simulation, the transformation effectively freezes the development of the program. The rest of the chapter will discuss this transformation process.

Attempts at transforming a definitive script into an imperative program were made by Michael [Michael89] and Hui [Hui90]. Trident is a software tool designed to convert an EDEN program into C program. EDEN and C are chosen because they have a large common

subset of data types and operators.

```
/*declare                  /* X and target are the variables to be changed */
change(X, target);
*/

X = 0;              /* this redundant assignment conveys type information */
target = 36;
sqrx is X * X;
correct is sqrx == target;

proc problem : target {          /* action for visualising target */
        writeln("X is the square-root of ", target, ". What is X?);
}

proc result : correct { /* action for visualising correct */
        if (correct)
                writeln("You've got it");
        else
                writeln("Then square of ",X," is ",sqrx," not ",target);
}
```

Listing 5.4: Square-root Guessing Program in EDEN

Listing 5.4 is a simple EDEN program for guessing the square-root of a given number. The definitions of X, target, sqrx and correct form a definitive state. The two actions are used for visualising the definitive state; they serve the same function as Scout definitions. On redefining the variable X, a message stating whether or not X is the square-root of the target, initially 36, will be displayed; when the variable target is redefined, a message stating the new goal of the problem will be displayed. In EDEN, a richer range of interaction is allowed (for example change the problem to solving cube-root instead of square-root), but as indicated in the first three lines of EDEN comments, only X and target are the intended input of the program.

```
int correct, sqrx, target, X;

_user_input() {
        char name[80];
        while (!feof(stdin)) {
                scanf("%s", name);
                if (!strcmp(name, "X")) _X();
                if (!strcmp(name, "target")) _target();
        }
}

_target() {
        scanf("%d", &target);
        problem();
        correct = sqrx == target;
        result();
}

_X() {
        scanf("%d", &X);
        sqrx = X * X;
        correct = sqrx == target;
        result();
}
```

```
result() {
        sqrx = X * X;
        correct = sqrx == target;
        if (correct) {
                printf("%s\n", "You've got it");
        } else {
                printf("%s%d%s%d%s%d\n",
                                "The square of ", X, " is ", sqrx, " not ", target);
        }
}

problem() {
        printf("%s%d%s\n", "X is the square-root of ", target, ". What is X?");
}

main() {
        X = 0;
        target = 36;
        problem();
        result();
        _user_input();
}
```

Listing 5.5: Translated Square-Root Guessing Program in C

Given the definitive state, the dependency between the variables can be calculated. Therefore, having specified which variables are subject to change (X and target in this case), the Trident translator can generate procedures to emulate the effect of redefining those variables in EDEN. In the procedural program constructed by the Trident translator constructs, each definitive variable is emulated by an imperative variable. The value of such a variable is maintained by repeated re-assignments. These assignments ensure that all the variables essential for calculating the formula associated with that variable are evaluated before the variable is assigned the value of the formula. Listing 5.5 is the transformed C program.

In its present state, Trident is a highly restricted translator. The generated program has a restricted form of input: it can only accept input of the form:

```
variable-name value
```

This is usually not the required input format. A better version of Trident should allow the specification of the expected input format.

Despite the fact that the input format is restrictive and that the current Trident translator is only able to translate very small examples, there is still an essential difference between the translated program and a 'normal' guessing program. Normally, a user cannot alter the target until the correct answer is given and he is not supposed to keep on changing X when he has achieved the correct answer. At some stage, a guessing program would usually provide a channel for exit. Because the EDEN program in Listing 5.3 allows X and target to be redefined at any stage and never terminates, the translated program also inherited these properties. It would not be difficult to enhance the Trident translator to generate a more appropriate program. The change construct informs the translator what the user is privileged to act upon a definitive state. It is not hard to imagine a version of Trident translator which can grant conditional privileges. The user might start with the privilege to change X; if correct becomes true, the user might be privileged to change target; termination of the program means that the user has no longer any privilege to change the definitive state.

In the discussion of Trident above, we are in effect exploring a non-traditional way of software development. It is not writing a higher level program satisfying the specification, then translating it into a program in the target language, as in the case of writing a C++ program and then translated it into a C program for execution. It is first writing a program which has a different input specification (a specification that allows a wider range of input, and where the input formats are also different), then developing a program satisfying the original specification by restricting the range of the input and converting their formats back to the original specification. The situation can be depicted in Figure 5.4, where a larger area means a higher degree of freedom in implementation.

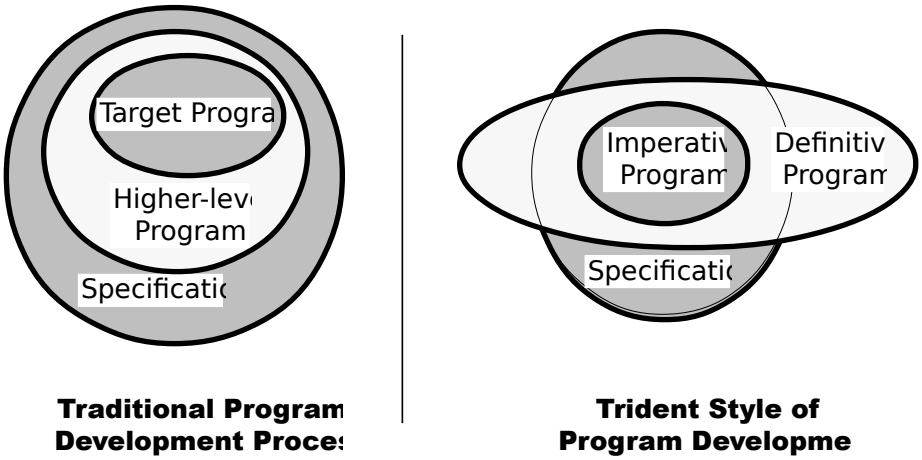

Figure 5.4: The Trident Way of Software Development

During the development of a definitive script, the design and simulation processes are interleaved. This shortens the editing stage of the exploratory software development cycle. Trident shows that it is possible to freeze the development of a definitive script and use the script to develop an executable program. In this way, the final program has a better run-time performance.

This summary is not needed?

5.4. Summary

Writing a script of definitions and performing on-line modification of the script is the simplest way of using definitive notations. The main use of these interactions is to develop a definitive script which models the state or part of the state of a system. The on-line modification facility favours exploratory development of the definitive state model. There are many more factors of definitive notations that are advantages for exploratory development of definitive state. To help the programmer to comprehend the state, definitive notations have domain-specific data types and operators. The way of expressing dependency in a definition provides a strong but modifiable link between variables. This allows a neat separation of internal state and its presentation. It is particularly helpful in visualising abstract information such as the speed of a vehicle. Also tools may be built to rearrange the definitions to provide useful insight for the programmer. To help in the editing phase of exploratory design, definitions have potential for building up a good undo facility. In addition, redefinition can serve both to effect redesign and simulation. This means that the development of a definitive state can be done in a continuously executing environment.

Th4.2: Definitive Programming for Parallelism