Correctness Preserving Transformations on a Multipass Occam Compiler

Karin Glindtvad, Hanne Riis Nielson
Aarhus University *

Oktober 1991

Abstract

The verification of a compiler may be a substantial task. However, by introducing correctness preserving program transformations some automated assistance becomes available. The idea is to specify an initial multipass compiler, to verify it in the usual way and then, while preserving the overall correctness result, to transform it into a more efficient single pass compiler. This transformation process may be performed using the fold/unfold framework of Burstall and Darlington and automation is provided by the Flagship Programming Environment. We illustrate this transformation process on a compiler for a subset of Occam.

1 Introduction

The compilation of an object program into target code is a rather complicated process in which many aspects have to be considered. To obtain an efficient compiler, it is often important that all aspects are treated in a single pass. Unfortunately, this may cause the correctness proof to be extremely

*Computer Science Department, Aarhus University, Ny Munkegade 116, 8000 Aarhus C, Denmark, tel: +45 86 12 71 88, e-mail: hrnielson@daimi.aau.dk

complicated. However, if the compiler is split into several passes, each handling one specific aspect, then the correctness proof may become manageable. Having proved each pass correct, the goal is to transform the composition of the passes into a single pass compiler while preserving the overall correctness result.

We illustrate this transformation process for a small subset of Occam [In88a]. The initial compiler consists of four different passes whose purposes are

- to replace variables by addresses,
- to assign temporary addresses to subexpressions,
- to generate code for the transputer [In88b], and
- to compute the space required to execute the code on a transputer.

We transform this four pass compiler into a single pass compiler using the fold/unfold framework of Burstall and Darlington [BuDa77]. The Flagship Programming Environment [Flag90] can be used to automate this process. In addition to supporting the fold/unfold framework, the Flagship Programming Environment has facilities for composing and guiding the transformation steps by goal oriented scripts, and by different kinds of higher order transformation rules. Detailed knowledge of the transformation process is required to specify the scripts whereas the higher order transformation rules are fairly easy to use but more limited in their applicability.

The structure of the paper is as follows. In Section 2 we illustrate the idea behind the fold/unfold framework by transforming the expression part of the Occam subset. Then the Flagship Programming Environment is introduced and in Section 3 we demonstrate how the same transformation may be performed automatically by presenting a goal oriented script for this. With this gentle introduction behind us, the transformation of the compiler for the considerably larger Occam subset PL_0 is described in Section 4; full details are given in the appendices. Finally, in Section 5 we give some concluding remarks.

2 A Simple Example of Fold/Unfold

To illustrate the concepts of the fold/unfold framework we give an example showing how the four pass compiler for the expression language can be transformed into a single pass compiler. Considering the automation later on, the compiler is given as a `Hope+` program [Perry89].

The syntax of the expressions is given by the type definition

```
data exp_0 == con_0 num ++ var_0 id ++ exp_0 add_0 exp_0;
```

To perform the compilation we need an environment, being a list of coherent identifiers and addresses:

```
data v_env == list(id # adr);
```

Assuming that the environment contains addresses for all free variables of the expression the first step is to eliminate the environment by moving the information from the environment to an intermediate datatype, `exp_1`:

```
data exp_1 == con_1 num ++ adr_1 adr ++ exp_1 add_1 exp_1;
```

This is ensured by the function:

```
dec E_1 :  exp_0 # v_env -> exp_1;
```

defined in Table 1. The next step is to introduce temporary addresses for the computation of expressions. To do that the datatype `exp_2` is introduced:

```
data exp_2 == push_2(num # adr)
              ++ fetch_2(adr # adr)
              ++ add_2 (adr # adr # adr)
              ++ trip_2(exp_2 # exp_2 # exp_2);
```

The result of this pass is a program specified as a sequence of register operations. The pass is specified by the function:

```
dec E_2 :  exp_1 # adr -> exp_2;
```

defined in Table 1. The second parameter contains information about the next free address to be used. In the third pass the sequence of register operations is translated into a sequence of abstract transputer instructions [In88b]. We define:

```
data instr == LDC(num) ++ LDL(adr) ++ STL(adr) ++ ADD;
```

and the pass is specified by the function:

```
dec E_3 :  exp_2 -> list instr;
```

of Table 1. Finally, in the fourth pass the required workspace is computed as expressed by the function:

```
dec E_4 :  exp_0 -> num;
```

also defined in Table 1.

The result of applying `E_1, E_2` and `E_3` to an object program is the translation of the object program into target code, as specified by the definition of `E_trans`:

```
dec E_trans :  exp_0 # v_env # adr -> list instr;

--- E_trans(e,en,t) <= E_3(E_2(E_1(e,en),t));
```

The translation of the object program *and* the computation of required workspace is captured in the definition of `E_comp`:

```
dec E_comp :  exp_0 # v_env # adr -> (list instr # num);

--- E_comp(e,en,t) <= (E_trans(e,en,t)),E_4 e);
```

```
E_1 (con_0 c,en)        <= con_1 c;
E_1 (var_0 x,en)        <= adr_1 lookup(x,en);
E_1 (e1 add_0 e2,en)    <= E_1(e1,en) add_1 E_1(e2,en);

E_2 (con_1 c,t)         <= push_2(c,t);
E_2 (adr_1 a,t)         <= fetch_2(a,t);
E_2 (e1 add_1 e2,t)     <= trip_2(E_2(e1,t),E_2(e2,t+1),
                                  add_2(t,t+1,t));

E_3 (push_2(c,t))       <= [LDC c,STL t];
E_3 (fetch_2(a,t))      <= [LDL a,STL t];
E_3 (add_2(t1,t2,t3))   <= [LDL t1,LDL t2,ADD,STL t3];
E_3 (trip_2(e1,e2,e3))  <= E_3 e1<>(E_3 e2<>E_3 e3);

E_4 (con_0 c)           <= 1;
E_4 (var_0 x)           <= 1;
E_4 (e1 add_0 e2)       <= max(E_4 e1,E_4(e2)+1);
```

Table 1: The expression compiler.

2.1 The Transformation Process

The purpose of the transformation process will be to transform the four pass compiler of Table 1 into a single pass compiler. This means that we want a definition of `E_comp` that may call itself recursively, but neither calls `E_1`, `E_2`, `E_3`, `E_4` nor `E_trans`, as shown in Table 2. The above definition of `E_comp` will be considered as the basis of the transformation process; in the terminology of the fold/unfold framework it is called a *Eureka definition*. In order to obtain the new definition of the present Eureka definition, each instance of the Eureka definition is transformed separately like a kind of case analysis. The specification of the current instance to be transformed is done by *instantiation*, that is by introducing a substituting instance of the Eureka definition.

However, in order to transform `E_comp` the definition of `E_trans` should be transformed first. Therefore the transformation will be performed in two stages; first the `E_trans` stage and next the `E_comp` stage.

```
E_comp(con_0 c,en,t)
  <= ([LDC c,STL t], 1)
E_comp(var_0 x,en,t)
  <= ([LDL lookup(x,en),STL t], 1)
E_comp(e1 add_0 e2,en,t)
  <= let (c1,w1)==E_comp(e1,en,t) in
       let (c2,w2)==E_comp(e2,en,t+1) in
        (c1<>(c2<>[LDL t,LDL t+1,ADD,STL t]),
         max(w1,w2+1))
```

Table 2: The single pass compiler, `E_comp`, as produced manually.

2.1.1 The `E_trans` Stage

The transformation of `E_trans(con_0 c,en,t)` is given as a sequence of steps starting with the instantiation of `E_trans` followed by the unfolding of each of the compilation functions, `E_1,..,E_3`. To *unfold* is to rewrite the left hand side of the definition by the right hand side of it. The transformation of the instance `E_trans(var_0 x,en,t)` is performed by a similar sequence of transformation steps, whereas the transformation of the last instance of `E_trans` is more complex. As in the two previous cases the three compilation functions are unfolded, but the result still contains calls of the compilation functions. However, this problem is solved by folding against `E_trans`. To *fold* is to recognize a subterm as the right hand side of the definition and rewrite the subterm by the left hand side. Finally the remaining call of `E_3` is removed by unfolding `E_3` once more and now `E_trans` has reached the desired form. The transformation of `E_trans(e,en,t)` is shown in Table 3.

2.1.2 The `E_comp` Stage

The transformation of `E_comp(con_0 c,en,t)`, is given as a sequence of steps starting with the instantiation of `E_comp` followed by the unfolding of the compilation functions, `E_trans` and `E_4`. The transformation of the instance `E_comp(var_0 x,en,t)` is performed by an equivalent sequence of transformation steps, whereas the transformation of the last instance of `E_comp` is more tricky. As in the two previous cases the two compilation functions

are unfolded. However, the result still contains unwanted applications. By *abstraction*, that is by introducing a `let` clause, it is possible to obtain a situation where `E_comp` may be folded and then `E_comp` has reached the desired form. This combination of abstraction and folding is often referred to

```
E_trans(con_0 c,en,t)
  <= E_3(E_2(E_1(con_0 c,en),t)))                    (inst)
  <= E_3(E_2(con_1 c,t))                             (unf E_1)
  <= E_3(push_2(c,t))                                (unf E_2)
  <= [LDC c,STL t]                                   (unf E_3)

E_trans(var_0 x,en,t)
  <= E_3(E_2(E_1(var_0 x,en),t))                     (inst)
  <= E_3(E_2(adr_1 lookup(x,en),t))                  (unf E_1)
  <= E_3(fetch_2(lookup (x,en),t))                   (unf E_2)
  <= [LDL lookup(x,en),STL t]                        (unf E_3)

E_trans(e1 add_0 e2,en,t)
  <= E_3(E_2(E_1(e1 add_0 e2,en),t))                   (inst)
  <= E_3(E_2((E_1(e1,en) add_1 E_1(e2,en)),t))         (unf E_1)
  <= E_3(trip_2(E_2(E_1(e1,en),t),                     (unf E_2)
     E_2(E_1(e2,en),t+1),add_2(t,t+1,t)))
  <= E_3(E_2(E_1(e1,en),t))                            (unf E_3)
     <>(E_3(E_2(E_1(e2,en),t+1))
     <>E_3(add_2(t,t+1,t)))
  <= E_trans(e1,en,t)<>(E_trans(e2,en,t+1)             (fold)
     <>E_3(add_2(t,t+1,t)))
  <= E_trans(e1,en,t)<>(E_trans(e2,en,t+1)             (unf E_3)
     <>[LDL t,LDL t+1,ADD,STL t])
```

Table 3: The manual transformation of `E_trans`.

as *forced folding* [BuDa77]. The transformation of `E_comp(e,en,t)` is shown in Table 4. The result of this transformation process is a function, `E_comp`, which implements a correct single pass compiler for the expression language, assuming that the original compilation functions are correct. The complete

function is like the one shown in Table 2.

```
E_comp(con_0 c,en,t)
 <=(E_trans(con_0 c,en,t),E_4(con_0 c))            (inst)
 <=([LDC c,STL t],E_4(con_0 c))                    (unf)
 <=([LDC c,STL t],1)                               (unf)

E_comp(var_0 x,en,t)
 <=(E_trans(var_0 x,en,t),E_4(var_0 x))            (inst)
 <=([LDL lookup(x,en),STL t],E_4(var_0 x))         (unf)
 <=([LDL lookup(x,en),STL t],1)                    (unf)

E_comp(e1 add_0 e2,en,t)
 <=(E_trans(e1 add_0 e2,en,t),E_4(e1 add_0 e2))  (inst)
 <=(E_trans(e1,en,t)<>(E_trans(e2,en,t+1)        (unf)
     <>[LDL t,LDL t+1,ADD,STL t]),
     E_4(e1 add_0 e2))
 <=(E_trans(e1,en,t)<>(E_trans(e2,en,t+1)        (unf)
     <>[LDL t,LDL t+1,ADD,STL t]),
     max(E_4 e1,E_4(e2)+1))
 <= let (c1,w1)==(E_trans(e1,en,t),E_4 e1) in    (abst)
     let (c2,w2)==(E_trans(e2,en,t+1),E_4 e2) in
      (c1<>(c2<>[LDL t,LDL t+1,ADD,STL t]),
       max(w1,w2+1))
 <= let (c1,w1)==E_comp(e1,en,t) in                (fold)
     let (c2,w2)==E_comp(e2,en,t+1) in
      (c1<>(c2<>[LDL t,LDL t+1,ADD,STL t]),
       max(w1,w2+1))
```

Table 4: The manual transformation of `E_comp`.

During this transformation process, most of the transformation rules of the fold/unfold framework have been used, the only exception is the application of *laws*.

3 Automation of the Simple Example using Scripts

One of the features of the Flagship Programming Environment is the ability to specify a transformation process in detail using the so-called scripts of the transformation language. Basically the language contains two kinds of constructions:

- control structures, specifying when and where to apply the transformation, and
- transformation operations like `fold, unfold, abstract`, and `replace`.

The scripts used to transform the two subdefinitions `E_trans` and `E_comp` are shown in Tables 5 and 8, and the structures and operations involved will be explained in detail below.

3.1 The Language of Scripts

To control the transformation process, we use three control structures:

- `choose_eqns` specifies the Eureka definition and transformation operations to be applied,
- `branch` ensures that every instance of the Eureka definition is transformed, and
- `compose` specifies a sequence of transformation operations to be applied to every instance of the Eureka definition.

To transform the Eureka definitions we use four transformation operations:

- `unfold`, which is equivalent to the unfold rule,
- `fold`, which is equivalent to the fold rule,
- `replace`, which may be viewed as a generalization of the application of laws, and

```
trans_script <=
 choose_eqns[
  (eqspec "E_trans e",
   branch[
    compose[
     unfold(any_term,eqspec "E_1 e"),
     unfold(any_term,eqspec "E_2 e"),
     unfold(any_term,eqspec "E_3 e"),
     fold(all_terms,eqspec "E_trans e"),
     unfold(any_term,eqspec "E_3 e")]])];
```

Table 5: A script transforming E_trans.

- `abstract`, which may be viewed as a specialization of the abstraction rule.

The `unfold` and `fold` operations take two parameters. The first determines whether several occurrences of an application, possibly with different arguments, are allowed or not. In the example of Table 5 the entire Eureka definition is subject to transformation implying that no exact context specification is required. Therefore, it is sufficient to use the two subterms `any_term` and `all_terms`. `Any_term` means that only one occurrence is allowed in the program, whereas `all_terms` means that several occurrences are permitted. The second parameter specifies the function to be unfolded/folded by the use of `eqspec "f x"`. If the `unfold` and `fold` operations cannot be performed, they correspond to a *noop* operation. This means that, whenever only these operations are used, it suffices to develop a script for the most complicated transformation of an instance of the Eureka definition. For the `E_trans` function this means that Table 3 is the guide to the definition of the script in Table 5. The control structure `branch` will ensure that all instantiations are considered. Therefore, contrary to the original framework, instantiation does not exist as an independent transformation operation. This is illustrated in Table 5 with the transformed program shown in Table 6.

The `replace` operation takes two parameters. The first one is a subterm specifying two `Hope_patterns` to be exchanged and the context for the re-

```
E_trans(con_0 v_z1,y_z1,x_z1)
  <= [LDC v_z1,STL x_z1];
E_trans(var_0 u_z1,y_z1,x_z1)
  <= [LDL lookup(u_z1,y_z1),STL x_z1];
E_trans(Z_z1 add_0 Y_z1,y_z1,x_z1)
  <= E_trans(Z_z1,y_z1,x_z1)<>(E_trans(Y_z1,y_z1,x_z1+1)
      <>[LDL x_z1,LDL(x_z1+1),ADD,STL x_z1]);
```

Table 6: `E_trans`.

placement. The second parameter verifies that the replacement is correctness preserving. In the current version of the system[1] the last parameter is ignored and thus cannot be used to ensure correctness. Therefore, this burden will rest on the user of the system. In this paper we shall use the second component as a "comment".

```
forced_fold_E context <=
 compose[
  abstract sbterm("E_4 x",context),
  abstract sbterm("E_trans(x,x1,x2)",context),
  replace(sbterm(
            "(let p1==e1 in let p2==E_4 e2 in e3," <>
            " let (p1,p2)==(e1,E_4 e2) in e3)",any),
          "tupling"),
  fold(all_terms,eqspec "E_comp e")];
```

Table 7: Forced folding.

The `abstract` operation takes one parameter that determines which subterm to abstract and it introduces a unique variable to replace each occurrence of this particular subterm. This means that only applications with

[1]Flagship Programming Environment version May 31, 1991.

common arguments are abstracted. If multiple applications with different arguments occur, as in `E_comp`, then each application must be abstracted separately by giving the appropriate context information.

To encapsulate the abstraction needed in the transformation of `E_comp`, we introduce a user defined `forced_fold` macro, as shown in Table 7. To accomplish the correct combination of the `let`-clauses, it is necessary to specify the application of `E_4`. Note that the `replace` operation used in this macro is correctness preserving.

Unlike `unfold` and `fold`, the `abstract` and `replace` operations must always be applicable and so do not have the option of corresponding to *noop* operations. Therefore, the transformation of each instance of `E_comp` must be explicitly specified. In the present version of the Flagship Programming Environment a script transforming `E_comp` is as shown in Table 8. Here we use the `branch` operation corresponding to the transformation illustrated in Table 4.

```
comp_script <=
 choose_eqns[
  (eqspec "E_comp e",
   branch[
    compose[
     unfold(any_term,eqspec "E_trans((e1 add_0 e2),en,t)"),
     unfold(any_term,eqspec "E_4 e"),
     forced_fold_E (context_match
                    ("--- E_comp(x add_0 _,_,_)<=r")),
     forced_fold_E any ],
    compose[
     unfold(any_term,eqspec "E_trans(con_0 c,en,t)"),
     unfold(any_term,eqspec "E_4 e")],
    compose[
     unfold(any_term,eqspec "E_trans(var_0 x,en,t)"),
     unfold(any_term,eqspec "E_4 e")]])];
```

Table 8: A script transforming `E_comp`.

3.2 Application of Scripts

A script is applied to a Eureka definition by the use of a metafunction, `apply_script`. Bearing the separation of the Eureka definition in mind, the transformation process may be expressed by:

```
‘‘doublepass’’:=apply_script(trans_script,‘‘multipass’’)
‘‘singlepass’’:=apply_script(comp_script,‘‘doublepass’’)
```

Here the `Hope+` module `‘‘multipass’’` contains the Eureka definitions and the contents of Table 1. The contents of the module `singlepass` is shown in Table 9. Note the close correspondance between the single pass compiler `E_comp` of Table 9 and the result of the manual transformation as shown in Table 2.

```
E_comp(con_0 V_z1,y_z1,x_z1)
  <=([LDC V_z1,STL x_z1],1);
E_comp(var_0 U_z1,y_z1,x_z1)
  <=([LDL lookup(U_z1,y_z1),STL x_z1],1);
E_comp(Z_z1 add_0 Y_z1,y_z1,x_z1)
  <= let (Y_z2,X_z2)==E_comp(Y_z1,y_z1,x_z1+1) in
     let (S_z1,R_z1)==E_comp(Z_z1,y_z1,x_z1) in
      (S_z1<>(Y_z2<>
       [LDL x_z1,LDL(x_z1+1),ADD,STL x_z1]),
       max(R_z1,X_z2+1));
```

Table 9: `E_comp` as produced by the Flagship Programming Environment.

4 Transforming the PL_0 Compiler

The expression language described in the previous section is a small subset of the PL_0 language presented by [LøvJen89] which is itself a subset of Occam [In88a]. A PL_0 program is a sequential process that may interact with the environment using two predefined channels, `in` and `out`. Apart from this,

PL_0 is quite similar to Dijkstra's language of Guarded Commands [Dijk75].

To specify the addresses of the channels, a channel environment is defined as:

```
data c_env == adr # adr;
```

and the necessary information is then contained in:

```
data env == v_env # c_env;
```

where `v_env` is defined as in Section 2. The PL_0 language contains the syntactic categories *program*, *block*, *process*, *guard* and *expression* as described by the BNF grammar:

```
p   ::= b
b   ::= x:int;b | sp
sp  ::= SKIP | STOP | x:=e | ch?  x | ch!  e |
        SEQ(sp1,...,spn) | IF(g1,...,gn) | WHILE(e,sp)
g   ::= e -> sp
e   ::= c | x | TRUE | FALSE | mop e | e1 dop e2
    where mop ∈ {−, NOT}, and
          dop ∈ {+, −, ∗, /, REM, =, <>, >, <, ≥, ≤, AND, OR}
```

For each syntactic category we specify four compilation functions. The purpose of the *program compiler* is to provide an initial environment specifying the channels to be used in the object program and to activate the *block compiler*. The *block compiler* will then update the current variable environment and activate the *process compiler*. The *process compiler* provides some of the compilation of the object program, and activates the *guard compiler* and the *expression compiler*. The initial function specifications are given in Appendix A.2. The relationship between translation of the object program into target code *and* computation of required workspace is captured by the following specifications:

```
C_comp(prog)          <= (C_3(C_2(C_1 prog)),C_4 prog)
B_comp(block,en)      <= (B_3(B_2(B_1(block,en))),B_4 block)
P_comp(proc,en,t)     <= (P_3(P_2(P_1(proc,en),t)),P_4 proc)
G_comp(guard,en,t)    <= (G_3(G_2(G_1(guard,en),t)),G_4 guard)
E_comp(exp,en,t)      <= (E_3(E_2(E_1(exp,en),t)),E_4 exp)
```

Given these Eureka definitions the transformation process using scripts is very similar to the one performed in Section 3. However, a couple of interesting points arise.

4.1 A Strategy for the Transformation

When considering the transformation of the increased number of Eureka definitions it should be clear that we need a strategy depending on the structure of the initial compiler. The structure of the compilation functions is similar for all four passes, and the calling structure of each pass in the compiler may be characterized as a kind of hierarchy of functions as displayed in Figure 1 for the composition functions.

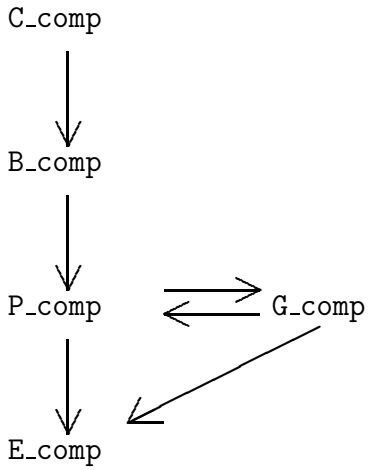

Figure 1: The hierarchy of definitions

In order to specify a strategy for transforming this hierarchy of definitions,

a number of issues should be discussed. The first issue is whether the mutual recursion of `P_comp` and `G_comp` may prevent transformation. Fortunately, only instances of the Eureka definitions are transformed during the transformation process leaving the Eureka definition itself unchanged, so whenever the righthand side of a Eureka definition is recognized, a `fold` operation may be applied regardless of whether this Eureka definition has been subject to transformation or not. This ensures that both `P_comp` and `G_comp` may be transformed, and it also implies that e.g. `B_comp` may be transformed before `E_comp`. When the transformation process has terminated, the original Eureka definitions have become redundant and they may be removed from the result of the transformation.

The next issue is how to separate the transformation process into stages. The obvious way is to split each definition into two subdefinitions `X_trans` and `X_comp`. Unfortunately, such a separation will introduce restrictions upon the transformation order because for each `X`, the `X_trans` stage must be carried out before the corresponding `X_comp` stage. However, as explained above the definitions of the five syntactic categories can be transformed in any order. This means, for example, that `E_trans` must be transformed before `E_comp`, `G_trans` before `G_comp` but `G_comp` may be transformed before `E_trans`. Therefore, some kind of control of the transformation order is necessary.

Taking the above issues into account, an appropriate strategy will be:

- To divide the transformation process in two stages specified by `X_trans` and `X_comp`,
- to transform the `X_trans` stage before the `X_comp` stage,
- to transform each stage bottom up starting with `E_, G_,..., C_`, and
- to remove the original Eureka definitions, which are now redundant.

4.2 The Pattern of Scripts

The first step of the strategy above is to split each Eureka definition into two subdefinitions of the following forms:

```
X_trans( obj_prog, ...  )
         <= X_3(X_2(X_1( obj_prog,..  ),..))
```

and

```
X_comp( obj_prog,...  )
         <= ( X_trans(obj_prog,...), X_4 obj_prog )
```

and then transform each stage separately.

4.2.1 The `X_trans` Stage

When the separation of the Eureka definitions has been done the next step of the strategy is to transform the `X_trans` stage. The `X_trans` Eureka definitions are, as shown above, specified by the composition of three compilation functions. The only difference between the forms of these five definitions is the number of arguments they take (see Appendix B.1.1). This implies that scripts used to transform these definitions may have some common structures, and specifying these in a tactic may help to derive a general pattern for the scripts. The tactic for transforming an `X_trans` definition is given by the following three steps:

- Unfold the functions used in the definition, always starting with the innermost application,
- fold against the Eureka definition, and
- unfold certain functions to remove additional and unwanted applications of functions, or fold against other Eureka definitions.

The first two steps are often referred to as the *composition* or *fusion* tactic [Fea87]. The last step is needed for two problem specific reasons: Because of the hierarchical structure of the compiler it may be necessary to fold against other Eureka definitions and because of the special structure of `E_3` and `exp_2` it will be necessary to perform extra unfolding as we saw in Sections 2 and 3. Therefore this last step may vary, depending on the structure of the functions used in the present Eureka definition.

When transforming the `X_trans` definitions, only `unfold` and `fold` operations are necessary which means that the transformation may be performed using implicit instantiation only. Furthermore, the number of arguments that a definition takes is unimportant in this connection. Hence the general pattern for a script transforming an `X_trans` definition using the above tactic is of the form given in Table 10. Considering the script of Table 10, it is obvious

```
trans_script <=
 choose_eqns[
  (eqspec "X_trans x",
   branch[
    compose[
     unfold(any_term,eqspec "X_1 x"),
     unfold(any_term,eqspec "X_2 x"),
     unfold(any_term,eqspec "X_3 x"),
     fold(all_terms,eqspec "X_trans x"),
     unfold(all_terms,eqspec "X_n x"),
      ...
     fold(all_terms,eqspec "Y_trans y")
      ... ]])];
```

Table 10: A pseudo trans_script.

that the `trans_script` of Section 3 is an instance of this general pattern. In fact, even though some of the scripts shown in Appendix B.1.2 may appear to contain an unexpected large number of `fold` operations against other definitions, they are all instances of the general script pattern of Table 10.

The abundance of `fold` operations in some of the scripts is due to the implicit instantiation where each instance of a given Eureka definition is transformed using one particular script. Therefore, every operation necessary at some state in the transformation process must be present in the script, implying that not all parts of the script will be used when transforming any single instance of the Eureka definition. As shown in Appendix B.1.2, the five scripts are aggregated into one script performing the first stage of the transformation process. The result of the first tranformation stage is shown

in Appendix B.1.3.

4.2.2 The `X_comp` Stage

According to our strategy, the next step is to transform the `X_comp` stage. The `X_comp` Eureka definitions are, as previously shown, specified by the tupling of two compilation functions and the only difference between the forms of these five definitions is the number of arguments each definition takes (see Appendix B.2.1). Therefore, a tactic for transforming the `X_comp` definitions may be specified by the following three steps:

- Unfold the functions used in the definition,
- forced_fold against the Eureka definition, and
- forced_fold against other Eureka definitions.

The first two steps are often referred to as the *tupling tactic* [Fea87]. As in the previous stage the last step is due to the hierarchical structure of the compilation functions.

The abstraction needed in the forced_fold steps uses the two operations `abstract` and `replace`. This implies that for each instance of `X_comp` the first argument must be explicitly specified in the script, whereas the last two arguments are unimportant. Therefore, the general pattern for a script performing this stage for one `X_comp` definition is as shown in Table 11.

The first step must be performed whereas the last two steps may be applied several times or not at all. Considering the script of Table 11, it is easy to see that the `comp_script` of Section 3 is an instance of this general pattern. As shown in Appendix B.2.2, the scripts for the `X_comp` definitions are aggregated into one script performing the second step of the transformation process, and the result is shown in Appendix B.2.3.

The forced_fold operations used in the last two steps are encapsulated in `forced_fold` macros which may also be described by a general pattern, as illustrated in Table 12. Notice, that in order to achieve the correct combination of the `let`-clauses introduced by the `abstract` operations, parts of the

```
comp_script <=
 choose_eqns[
  (eqspec "X_comp x",
   branch[
    compose[
     unfold(any_term,eqspec "X_trans(inst_1 x,en,t)"),
     unfold(any_term,eqspec "X_4 x"),
     forced_fold_X,
     forced_fold_Y, ... ],
    compose[
     unfold(any_term,eqspec "X_trans(inst_2 x,en,t)"),
     unfold(any_term,eqspec "X_4 x"),
     forced_fold_X,
     forced_fold_y, ... ],
     ⋮
    compose[
     unfold(any_term,eqspec "X_trans(inst_n x,en,t)"),
     unfold(any_term,eqspec "X_4 x"),
     forced_fold_X,
     forced_fold_Y, ... ])];
```

Table 11: A pseudo comp_script.

Hope_patterns used in the replace operations must be explicitly specified. Considering the macro of Table 12, it is easy to see that the macro of Section 3 is an instance of this general pattern. The macros used in the transformation of the X_comp definitions are shown in Appendix B.2.2.

4.2.3 Application of Scripts

The procedure for applying these scripts is as described in Section 3.2. This means that trans_script is applied first and then comp_script which is concordant with the transformation strategy. The single pass compiler for PL_0 is shown in Appendix B.2.3.

```
forced_fold_X context <=
 compose[
  abstract sbterm ("X_4 x",context),
  abstract sbterm ("X_trans x",context),
  replace(sbterm(
            "(let p1==e1 in let p2==X_4 e2 in e3,"<>
            " let (p1,p2)==(e1,X_4 e2) in e3 )", any),
          "tupling" ),
  fold( all_terms, eqspec "X_comp x" ) ];
```

Table 12: A pseudo forced_fold macro.

4.3 The Problem of Multiple Arguments

When specifying the *process compiler* an interesting problem arises. The SEQ and IF constructors take an arbitrary number of arguments rather than some fixed number. The obvious way to specify compilation functions for these constructors is therefore to use a general map function, as briefly illustrated by:

```
P_1(seq_0 plist,en)
  <= seq_1(map(lambda x => P_1(x,en)end)plist);
P_2(seq_1 plist,t)
  <= seq_2(map(lambda x => P_2(x,t)end)plist);
```

where `map` is defined by:

```
map f nil     <= nil;
map f(p::ps)  <= (f p)::(map f ps);
```

Consider now the Eureka definition:

```
P_ex(plist,en,t) <= P_2(P_1(plist,en),t)
```

Instantiation gives:

```
P_ex(seq_0 p::plist,en,t)
   <= P_2(P_1(seq_0 p::plist,en),t)
```

Following the tactic of Section 4.2.1, the innermost application must be unfolded:

```
<= P_2(seq_1(map(lambda x => P_1(x,en)end)p::plist),t)
```

And unfolding the `map` function then gives:

```
<= P_2(seq_1(P_1(p,en)::
              map(lambda x => P_1(x,en)end)plist),t)
```

Unfolding `P_2` gives:

```
<= seq_2(map(lambda x => P_2(x,t)end)
             (P_1(p,en)::
                map(lambda x => P_1(x,en)end)plist)))
```

Unfolding `map` then gives:

```
<= seq_2(P_2(P_1(p,en),t)::
          (map(lambda x => P_2(x,t)end)
               (map(lambda x => P_1(x,en)end)plist)))
```

Then it is possible to fold the head of the list against `P_ex`:

```
<= seq_2(P_ex(p,en,t)::
          (map(lambda x => P_2(x,t)end)
               (map(lambda x => P_1(x,en)end)plist)))
```

When trying to fold applications in the tail of the list, the problem becomes obvious. In order to fold `map(lambda x => P_1(x,en) end) plist` against `P_1`, the constructor `seq_1` is required. Obviously, this conctructor is no longer available and we cannot perform the operation. The same problem arises when trying to fold the second application of the map function against `P_2`, so the tail of the list will never be folded against `P_ex`. There-

fore, although the semantics of both P_1 and P_2 is correct the transformation process will not succeed and P_ex will still be a "two pass compiler".

At first sight one may think that the problem is due to the use of higher-order functions. However, this is not the case. The problem arises because a recursive datatype (e.g. a list of processes) is encapsulated by a constructor (e.g. seq). When a recursive function is applied to a value of the recursive datatype, one will loose track of the constructor after the first unfolding and the transformation process is stuck. There are at least two solutions to the problem; to rewrite the datatype or to rewrite the functions.

We shall choose the latter and introduce a number of specialized map functions, thereby moving the necessary information from the constructors to the map functions. In the example above it would imply that two map functions M_1 and M_2 are introduced and P_1 and P_2 are redefined to:

```
P_1(seq_0 plist,en) <= seq_1 M_1(plist,en);
P_2(seq_1 plist,t)  <= seq_2 M_2(plist,t);
```

where M_1 and M_2 are specialized map functions:

```
M_1 (nil,en)       <= nil;
M_1 (p::plist,en)  <= P_1(p,en)::M_1(plist,en);

M_2 (nil,t)        <= nil;
M_2 (p::plist,t)   <= P_2(p,t)::M_2(plist,t);
```

The relationship between the specialized map functions is the same as between the compilation functions and given the definition:

```
M_ex(plist,en,t) <= M_2(M_1(plist,en),t)
```

the transformation of `P_ex (seq_0 p::plist,en,t)` may now be performed as shown in Table 13. When the transformation has terminated, the number of map functions as well as the number of compilation functions has decreased.

```
P_ex(seq_0 p::plist,en,t)
 <= P_2(P_1(seq_0 p::plist,en),t)                          (inst)
 <= P_2(seq_1(M_1(p::plist,en)),t)                         (unf)
 <= P_2(seq_1(P_1(p,en)::M_1(plist,en)),t)                 (unf)
 <= seq_2(M_2(P_1(p,en)::M_1(plist,en)),t)                 (unf)
 <= seq_2(P_2(P_1(p,en),t)::M_2(M_1(plist,en),t))          (unf)
 <= seq_2(P_ex(p,en,t)::M_2(M_1(plist,en),t)               (fold)
 <= seq_2(P_ex(p,en,t)::M_ex(plist,en,t))                  (fold)
```

Table 13: Transformation of P_ex.

To transform the entire PL_0 compiler four specialized map functions are needed, and specifications of these functions are given in Appendix A.2. As for the compilation functions, the relationship between the map functions may be described by a Eureka definition:

```
M_comp(plist,en,t) <= (M_3(M_2(M_1(plist,en),t)), M_4 plist)
```

This Eureka definition is of the same form as the definitions given in Section 4.1. Therefore, it may be transformed using the same strategy and the same general script patterns as used in Section 4.2. This means that the four map functions are reduced to one, just like the four pass compiler is reduced to a single pass compiler. The expansion of the hierarchy of Eureka definitions is shown in Figure 2.

5 Conclusion

Using the fold/unfold framework and the Flagship Programming Environment, we have succesfully transformed a correct multipass compiler for an Occam subset into a correct single pass compiler. In connection with this transformation process the language of Scripts has shown to be a powerful tool for controlling and performing program transformations. Developing a script requires detailed knowledge of the transformation process so it will be an advantage if a higher level of transformation rules is available. One

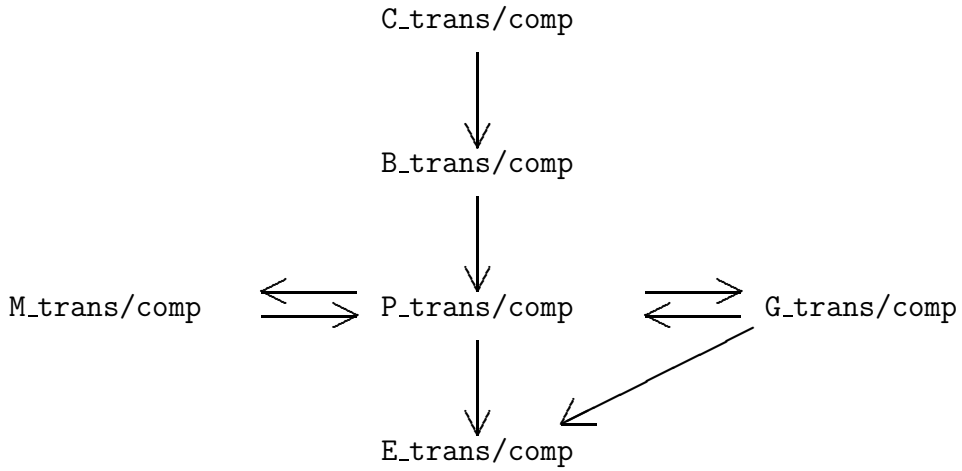

Figure 2: The extended hierarchy of definitions

approach to a more abstract level of transformation rules is to introduce *tactics* [Fea87]. Tactics are provided for transformation of classes of Eureka definitions.

The Flagship Programming Environment does supply a number of predefined general tactics including both the *composition* and the *tupling* tactics. However, these tactics are not sufficient to transform the PL_0 compiler entirely. Therefore, in order to reduce the amount of details necessary for developing goal oriented scripts, the transformation process is separated further by specifying subgoals depending on the predefined tactics. This makes the ability to combine tactics and scripts rather important. Unfortunately, the tactics in the version of the Flagship Programming Environment we have used are not fully developed so we have not been able to apply them to the PL_0 multipass compiler.

We should also like to mention that only a few of the facilities of the Flagship Programming Environment have been described in this paper. Several kinds of higher order transformation rules and transformation methods are included and we understand that more are planned for later versions of the system.

The main issue of the present paper has been the elimination of intermediate data structures arising in a multipass compiler. A similar development has been performed by Feather [Fea82] for a smaller language in the ZAP system. However, due to the larger complexity of our language PL_0 we need more complicated nested data structures for representing programs and this is the source of the intricate problems discussed in Section 4. Also the use of tupling does not arise in Feather's work. The general problem of eliminating intermediate data structures has been discussed in various contexts. In [Wad84, Wad85] and [Wad88], Wadler gives algorithms for eliminating intermediate lists and trees for a restricted domain of functional programs. Ganzinger and Giegerich [GaGie84] are concerned with the composition of passes of a multipass compiler and show how the so-called attribute coupled grammars can be used to specify the individual passes and subsequently facilitate the elimination of intermediate data structures.

A more general perspective on program transformations is taken in projects as e.g. CIP [Partsch84] and ProSpecTra [Krieg86]. Program transformations are here used in the development of programs from specifications, in program optimization and in program implementation. We anticipate that a development similar to ours can be performed in these settings as well.

Acknowledgement

We would like to thank John Darlington for making the Flagship Programming Environment available to us and Keith Sephton for general help with using the system. Flemming Nielson, Anders Gammelgaard and Torben Amtoft commented on a previous version of this paper. This work was supported by ProCoS, Esprit BRA project no. 3104.

References

[BuDa77] R.M.Burstall, J.Darlington : *A Transformation System for Developing Recursive Programs,* JACM Vol. 24 No.1 1977.

[Dijk75] E.Dijkstra : *Guarded Commands, Nondeterminacy and Formal Derivation of programs,* CACM Vol. 18 No.8 1975.

[Fea82] M.S.Feather : *A System for Assisting Program Transformation,* ACM Transactions on Programming Languages and Systems, Vol.4, No.1, January 1982.

[Fea87] M.S.Feather : *A Survey and Classification of some Program Transformation Approaches and Techniques,* Program Specification and Transformation, L.G.L.T. Meertens (Editor), Elsevier Science Publishers B.V. (North Holland), IFIP, 1987.

[Flag90] K.M.Sephton, W.N.Chin, L.M.McLoughlin, H.M.Pull, R.L.J.While : *Using "tuples" the Flagship Programming Environment,* Technical Report, Department of Computing, Imperial College, London, 1990.

[FrKaLa89] M.Fränzle, B.v.Karger, Y.Lakhneche : *Compilation Specification and Verification,* ProCos Technical Report, Christian-Albrechts University, Kiel Germany, October 26, 1989.

[GaGie84] H.Ganzinger, R.Giegerich : *Attribute Coupled Grammars,* Proceedings of the ACM SIGPLAN '84 Symposium on Compiler Construction, SIGPLAN Notices Vol.19, No.6, June 1984.

[In88a] INMOS Ltd. : *Occam 2 Reference Manual*, Prentice Hall, 1988.

[In88b] INMOS Ltd. : *TRANSPUTER INSTRUCTION SET : A compiler writer's guide*, Prentice Hall, 1988.

[Krieg86] B.Krieg-Brückner et al. : *PROgram Development by SPECification and TRAnsformation,* ESPRIT '86: RESULTS AND ACHIEVEMENTS, Elsevier Science Publishers B.V. (North-Holland).

[LøvJen89] H.H.Løvengreen, K.M.Jensen : *Definition of the ProCoS Programming Language Level 0,* ProCos Technical Report, Technical University of Denmark, October 25, 1989.

[Partsch84] H.Partsch: *The CIP Transformation System,* Program Transformation and Programming Environments, NATO ASI Series, Series F: Computer and Systems Science 8, Springer-Verlag, 1984.

[Perry89] N.Perry : *Hope+,* IC/FPR/LANG/2.5.1/7, Department of Computing, Imperial College, London, 1989.

[Wad84] P.Wadler: *Listlessness is better than laziness: Lazy evaluation and garbage collection ar compile-time,* Proceedings of the ACM Symposium on Lisp and Functional Programming, August 1984.

[Wad85] P.Wadler: *Listlessness is better than laziness II: Composing listless functions,* Proceedings of the Workshop on Programs as Data Objects, LNCS 217, Springer-Verlag, 1985.

[Wad88] P.Wadler: *Deforestation: Transforming programs to eliminate trees,* ESOP '88, LNCS 300, Springer-Verlag, 1988.

A The Initial PL_0 Compiler

This appendix contains the Hope+ definitions of the datatypes and functions used to implement the PL_0 multipass compiler.

A.1 Auxiliary Functions

```
module aux;
pubtype v_env,c_env,env,id,adr,label;
pubfun max,maxelm,length,chanin,chanout,noofchan,
       initenv,nextfree,lookup,update;

type id == list char;
type adr == num;
type label == list char;
type v_env == list(id # adr);
type c_env == adr # adr;
type env == v_env # c_env;

dec max         : num # num          -> num;
dec maxelm      : list num           -> num;
dec length      : list alpha         -> num;
dec initenv     : env;
dec chanin      : env                -> adr;
dec chanout     : env                -> adr;
dec noofchan    : num;
dec nextfree    : env                -> num;
dec lookup      : id # env           -> adr;
dec update      : env # id # adr     -> env;
dec updatev_env : v_env # id # adr   -> v_env;

--- max(x,y) <= if(x>y) then x else y;

--- maxelm(n::ns) <= max(n,maxelm ns);
--- maxelm nil    <= 0;

--- length nil    <= 0;
--- length(x::xs) <= 1 + length xs;
```

```
--- initenv <= (nil,(1,2));

--- chanin(s,(i,o)) <= i;

--- chanout(s,(i,o))<= o;

--- noofchan <= 2;

--- nextfree(s,(i,o)) <= length(s) +(noofchan + 1);

--- lookup(x,(nil,(i,o)))
            <= if (x ="in") or (x = "out")
               then error [(1,"predefined channame")]
               else error [(2,"undefined identifier")];
--- lookup(x,((y,a)::s,(i,o)))
            <= if(x = y) then a
               else lookup(x,(s,(i,o)));

--- update((s,(i,o)),x,a) <=(updatev_env(s,x,a),(i,o));

--- updatev_env(nil,x,a) <= [(x,a)];
--- updatev_env((y,b)::s,x,a)
                          <= if x = y
                             then(x,a)::s
                             else(y,b)::updatev_env(s,x,a);
end;
```

A.2 The Multipass Compiler

```
module multipass;
uses aux;
pubtype exp_0,exp_1,exp_2,proc_0,proc_1,proc_2,
        guard_0,guard_1,guard_2,block_0,block_1,block_2,
        prog_0,prog_1,prog_2,instr;
```

```
pubconst dop_0,mop_0,con_0,var_0,true_0,false_0,skip_0,
         stop_0,assign_0,seq_0,in_0,out_0,if_0,while_0,
         blk_0,pro_0,gc_0,dop_1,mop_1,con_1,adr_1,true_1,
         false_1,skip_1,stop_1,assign_1,seq_1,in_1,out_1,
         if_1,while_1,pro_1,gc_1,push_2,fetch_2,bool_2,
         dop_2,trip_2,mop_2,tup_2,gc_2,assign_2,seq_2,in_2,
         out_2,stop_2,skip_2,if_2,while_2,pro_2,LDC,LDL,STL,
         IN,OUT,TESTERR,STOPERR,DOP,MOP,STOP,BLOCK,DEF,CJ,J,
         LOOP,NOT,EXIT;

pubfun E_1,E_2,E_3,E_4,P_1,P_2,P_3,P_4,G_1,G_2,G_3,G_4,
       M_1,M_2,M_3,M_4,B_1,B_2,B_3,B_4,C_1,C_2,C_3,C_4;

infix dop_0 : 1;
data exp_0 == true_0 ++ false_0 ++ con_0 num
           ++ var_0 id ++ mop_0 exp_0 ++ exp_0 dop_0 exp_0;

infix dop_1 : 1;
data exp_1 == true_1 ++ false_1 ++ con_1 num
           ++ adr_1 adr ++ mop_1 exp_1 ++ exp_1 dop_1 exp_1;

data exp_2 == push_2(num # adr) ++ fetch_2(adr # adr)
           ++ bool_2(num# adr) ++ mop_2(adr # adr)
           ++ dop_2(adr # adr # adr) ++ tup_2(exp_2 # exp_2)
           ++ trip_2(exp_2 # exp_2 # exp_2);

data proc_0;
data guard_0 == gc_0(exp_0 # proc_0);

data proc_1;
data guard_1 == gc_1(exp_1 # proc_1);

data proc_2;
data guard_2 == gc_2(exp_2 # proc_2);

data proc_0 == skip_0 ++ stop_0 ++ assign_0(id # exp_0)
            ++ in_0 id ++ out_0 exp_0 ++ seq_0 list proc_0
            ++ if_0 list guard_0 ++ while_0(exp_0 # proc_0);
```

```
data proc_1 == skip_1 ++ stop_1 ++ assign_1(adr # exp_1)
            ++ in_1(adr # adr) ++ out_1(exp_1 # adr)
            ++ seq_1 list proc_1 ++ if_1 list guard_1
            ++ while_1(exp_1 # proc_1);

data proc_2 == skip_2 ++ stop_2
            ++ assign_2(adr # exp_2 # adr)
            ++ in_2(adr # adr) ++ out_2(exp_2 # adr # adr)
            ++ seq_2 list proc_2 ++ if_2 list guard_2
            ++ while_2(exp_2 # proc_2);

data block_0 == blk_0(id # block_0)
             ++ pro_0 proc_0;

data block_1 == pro_1(adr # proc_1);

data block_2 == pro_2 proc_2;

type prog_0  == block_0;

type prog_1  == block_1;

type prog_2  == block_2;

data instr == LDC(num) ++ LDL(adr) ++ STL(adr)
           ++ IN ++ OUT ++ TESTERR ++ STOPERR ++ STOP
           ++ BLOCK(list instr) ++ DEF(label) ++ CJ(label)
           ++ J(label) ++ LOOP(list instr) ++ NOT ++ EXIT
           ++ MOP ++ DOP;

dec E_1         : exp_0 # env        -> exp_1;
dec E_2         : exp_1 # adr        -> exp_2;
dec E_3         : exp_2              -> list instr;
dec E_4         : exp_0              -> num;
dec G_1         : list guard_0 # env -> list guard_1;
dec G_2         : list guard_1 # adr -> list guard_2;
dec G_3         : list guard_2       -> list instr;
```

```
dec G_4         : list guard_0       -> list num;
dec P_1         : proc_0 # env       -> proc_1;
dec P_2         : proc_1 # adr       -> proc_2;
dec P_3         : proc_2             -> list instr;
dec P_4         : proc_0             -> num;
dec M_1         : list proc_0 # env  -> list proc_1;
dec M_2         : list proc_1 # adr  -> list proc_2;
dec M_3         : list proc_2        -> list instr;
dec M_4         : list proc_0        -> list num;
dec B_1         : block_0 # env      -> block_1;
dec B_2         : block_1            -> block_2;
dec B_3         : block_2            -> list instr;
dec B_4         : block_0            -> num;
dec C_1         : prog_0             -> prog_1;
dec C_2         : prog_1             -> prog_2;
dec C_3         : prog_2             -> list instr;
dec C_4         : prog_0             -> num;

--- E_1(true_0,en)        <= true_1;
--- E_1(false_0,en)       <= false_1;
--- E_1(con_0 c,en)       <= con_1 c;
--- E_1(var_0 x,en)       <= adr_1 lookup(x,en);
--- E_1(mop_0 e,en)       <= mop_1 E_1(e,en);
--- E_1(e1 dop_0 e2,en)   <= E_1(e1,en) dop_1 E_1(e2,en);

--- E_2(true_1,t)         <= bool_2(1,t);
--- E_2(false_1,t)        <= bool_2(0,t);
--- E_2(con_1 c,t)        <= push_2(c,t);
--- E_2(adr_1 a,t)        <= fetch_2(a,t);
--- E_2(mop_1 e_1,t)      <= tup_2(E_2(e_1,t),mop_2(t,t));
--- E_2(e1_1 dop_1 e2_1,t)<= trip_2(E_2(e1_1,t),E_2(e2_1,t+1),
                                    dop_2(t,t+1,t));

--- E_3(bool_2(bool,t))        <= [LDC bool,STL t];
--- E_3(push_2(con,t))         <= [LDC con,STL t];
--- E_3(fetch_2(a,t))          <= [LDL a,STL t];
--- E_3(mop_2(t1,t2))          <= [LDL t1,MOP,STL t2];
--- E_3(tup_2(e1_2,e2_2))      <= E_3 e1_2<>E_3  e2_2 ;
```

```
--- E_3(dop_2(t1,t2,t3))       <= [LDL t1,LDL t2,DOP,STL t3];
--- E_3(trip_2(e1_2,e2_2,e3_2))<= E_3 e1_2
                                  <>(E_3  e2_2<>E_3 e3_2);

--- E_4(true_0)     <= 1;
--- E_4(false_0)    <= 1;
--- E_4(con_0 c)    <= 1;
--- E_4(var_0 x)    <= 1;
--- E_4(mop_0 e)    <= E_4 e;
--- E_4(e1 dop_0 e2)<= max(E_4 e1,E_4(e2)+1);

--- G_1((gc_0(b,p)::gs),en)<= gc_1(E_1(b,en),P_1(p,en))::
                              G_1(gs,en);
--- G_1(nil,en)            <= nil;

--- G_2((gc_1(b,p)::gs),t) <= gc_2(E_2(b,t),P_2(p,t))::
                              G_2(gs,t);
--- G_2(nil,t)             <= nil;

--- G_3(gc_2(b,p)::gs) <= [BLOCK
                           ([TESTERR]<>(E_3 b<>
                            ([STOPERR,CJ("fail")]<>
                              (P_3 p
                              <>[J("exitif"),DEF("fail")]))
                               <>(G_3 gs<>[DEF("exitif")])))];
--- G_3 nil           <= [STOP];

--- G_4(gc_0(b,p)::gs) <= max(E_4 b,P_4 p)::G_4 gs;
--- G_4 nil             <= nil;

--- M_1(nil,en)   <= nil;
--- M_1(p::ps,en) <= P_1(p,en)::M_1(ps,en);

--- M_2(nil,t)    <= nil;
--- M_2(p::ps,t)  <= P_2(p,t)::M_2(ps,t);

--- M_3  nil      <= nil;
--- M_3(p::ps)    <= P_3 p<>M_3 ps;
```

```
--- M_4 nil       <= nil;
--- M_4(p::ps)    <=(P_4 p)::(M_4 ps);

--- P_1(skip_0,en)        <= skip_1;
--- P_1(stop_0,en)        <= stop_1;
--- P_1(assign_0(x,e),en) <= assign_1(lookup(x,en),E_1(e,en));
--- P_1(in_0 x,en)        <= in_1(lookup(x,en),chanin en);
--- P_1(out_0 e,en)       <= out_1(E_1(e,en),chanout en);
--- P_1(seq_0 ps,en)      <= seq_1 M_1(ps,en);
--- P_1(while_0(b,p),en)  <= while_1(E_1(b,en),P_1(p,en));
--- P_1(if_0(gs),en)      <= if_1 G_1(gs,en);

--- P_2(skip_1,t)         <= skip_2;
--- P_2(stop_1,t)         <= stop_2;
--- P_2(assign_1(a,e),t)  <= assign_2(a,E_2(e,t),t);
--- P_2(in_1(a,i),t)      <= in_2(a,i);
--- P_2(out_1(e,o),t)     <= out_2(E_2(e,t),o,t);
--- P_2(seq_1 ps,t)       <= seq_2 M_2(ps,t);
--- P_2(while_1(b,p),t)   <= while_2(E_2(b,t),P_2(p,t));
--- P_2(if_1 gs,t)        <= if_2 G_2(gs,t);

--- P_3(skip_2)          <= [];
--- P_3(stop_2)          <= [STOP];
--- P_3(assign_2(a,e,t)) <= [TESTERR]
                            <>(E_3 e
                             <>[STOPERR,LDL t,STL a]);
--- P_3(in_2(a,i))       <= [LDC a,LDC i,IN];
--- P_3(out_2(e,o,t))    <= [TESTERR]<>(E_3 e<>
                           [STOPERR,LDC t,LDC o,OUT]);
--- P_3(seq_2 ps)        <= M_3 ps;
--- P_3(while_2(b,p))    <= [LOOP
                            ([TESTERR]<>(E_3(b)
                              <>([STOPERR,NOT,
                                  CJ("cont"),EXIT,
                                  DEF("cont")]
                               <>P_3(p))))];
--- P_3(if_2(gs))         <= G_3 gs;
```

```
--- P_4(skip_0)           <= 0;
--- P_4(stop_0)           <= 0;
--- P_4(assign_0(x,e))    <= E_4 e;
--- P_4(in_0 x)           <= 0;
--- P_4(out_0 e)          <= max(E_4 e,1);
--- P_4(seq_0 ps)         <= maxelm(M_4 ps);
--- P_4(while_0(b,p))     <= max(E_4 b,P_4 p);
--- P_4(if_0 gs)          <= maxelm(G_4 gs);

--- B_1(blk_0(x,b),en) <= B_1(b,update(en,x,nextfree(en)));
--- B_1(pro_0 p,en)    <= pro_1(nextfree(en),P_1(p,en));

--- B_2(pro_1(a,p))    <= pro_2 P_2(p,a);

--- B_3(pro_2 p)       <= P_3 p;

--- B_4(blk_0(x,b))    <= 1 + (B_4 b);
--- B_4(pro_0 p)       <= P_4 p;

--- C_1 p <= B_1(p,initenv);

--- C_2 p <= B_2 p;

--- C_3 p <= B_3 p;

--- C_4 p <= noofchan + (B_4 p);
end;
```

B The Transformation Process

This appendix contains the specifications, the scripts and the result of the transformation process using the Flagship Programming Environment. As described in Section 4 there are two stages in the transformation process; the first is described in Appendix B.1 and the second in Appendix B.2.

B.1 The First Stage

B.1.1 The Eureka Definitions

```
module transdef;
uses aux,multipass;
pubfun E_trans,P_trans,G_trans,M_trans,B_trans,C_trans;

dec E_trans : exp_0 # env # adr -> list instr;
--- E_trans(e,en,t) <= E_3(E_2(E_1(e,en),t));

dec G_trans : list guard_0 # env # adr -> list instr;
--- G_trans(g,en,t) <= G_3(G_2(G_1(g,en),t));

dec P_trans : proc_0 # env # adr -> list instr;
--- P_trans(p,en,t) <= P_3(P_2(P_1(p,en),t));

dec M_trans : list proc_0 # env # adr -> list instr;
--- M_trans(p,en,t) <= M_3(M_2(M_1(p,en),t));

dec B_trans : block_0 # env -> list instr;
--- B_trans(b,en) <= B_3(B_2(B_1(b,en)));


dec C_trans : prog_0  -> list instr;
--- C_trans p <= C_3(C_2(C_1 p));
end;
```

B.1.2 The Script

```
module transscript;
pubfun trans_script;
uses scripts,aux,multipass,transdef;
dec trans_script : script;
```

```
--- trans_script
     <= choose_eqns[
          (eqspec "E_trans e",
            branch[
              compose[unfold(all_terms,eqspec "E_1 e"),
                unfold(all_terms,eqspec "E_2 e"),
                unfold(all_terms,eqspec "E_3 e"),
                fold(all_terms,eqspec "E_trans e"),
                unfold(all_terms,eqspec "E_3 e")]]),
          (eqspec "P_trans p",
            branch[
              compose[unfold(any_term,eqspec "P_1 p"),
                unfold(any_term,eqspec "P_2 p"),
                unfold(any_term,eqspec "P_3 p"),
                fold(all_terms,eqspec "P_trans p"),
                fold(all_terms,eqspec "M_trans m"),
                fold(all_terms,eqspec "G_trans g"),
                fold(all_terms,eqspec "E_trans e")]]),
          (eqspec "G_trans g",
            branch[
              compose[unfold(any_term,eqspec "G_1 g"),
                unfold(any_term,eqspec "G_2 g"),
                unfold(any_term,eqspec "G_3 g"),
                fold(all_terms,eqspec "G_trans g"),
                fold(all_terms,eqspec "P_trans p"),
                fold(all_terms,eqspec "E_trans e")]]),
          (eqspec "M_trans m",
            branch[
              compose[unfold(any_term,eqspec "M_1 m"),
                unfold(any_term,eqspec "M_2 m"),
                unfold(any_term,eqspec "M_3 m"),
                fold(all_terms,eqspec "M_trans m"),
                fold(all_terms,eqspec "P_trans p")]]),
          (eqspec "B_trans b",
            branch[
              compose[unfold(any_term,eqspec "B_1 b"),
                unfold(any_term,eqspec "B_2 b"),
                unfold(any_term,eqspec "B_3 b"),
```

```
                fold(all_terms,eqspec "B_trans b"),
                fold(all_terms,eqspec "P_trans p")]]),
          (eqspec "C_trans c",
            branch[
              compose[unfold(any_term,eqspec "C_1 c"),
                unfold(any_term,eqspec "C_2 c"),
                unfold(any_term,eqspec "C_3 c"),
                fold(all_terms,eqspec "B_trans c")]])];
end;
```

B.1.3 The Result of the First Stage

```
module newtrans;
pubfun B_trans,C_trans,E_trans,G_trans,M_trans,P_trans;
uses aux,multipass;
dec B_trans : block_0 # env -> list(instr);
dec C_trans : prog_0 -> list(instr);
dec E_trans : exp_0 # env # adr -> list(instr);
dec G_trans : list(guard_0) # env # adr -> list(instr);
dec M_trans : list(proc_0) # env # adr -> list(instr);
dec P_trans : proc_0 # env # adr -> list(instr);
--- B_trans(pro_0 T_z1,r_z1)
     <= P_trans(T_z1,r_z1,nextfree r_z1);
--- B_trans(blk_0(y_z2,x_z2),r_z1)
     <= B_trans(x_z2,update(r_z1,y_z2,nextfree r_z1));
--- C_trans t_z1
     <= B_trans(t_z1,initenv);
--- E_trans(z_z3 dop_0 y_z3,y_z1,x_z1)
     <= E_trans(z_z3,y_z1,x_z1)<>(E_trans(y_z3,y_z1,x_z1+1)
        <>[LDL x_z1,LDL(x_z1 + 1),DOP,STL x_z1]);
--- E_trans(mop_0 X_z3,y_z1,x_z1)
     <= E_trans(X_z3,y_z1,x_z1)<>[LDL x_z1,MOP,STL x_z1];
--- E_trans(var_0 Y_z3,y_z1,x_z1)
     <=[LDL lookup(Y_z3,y_z1),STL x_z1];
--- E_trans(con_0 Z_z3,y_z1,x_z1)
     <=[LDC Z_z3,STL x_z1];
--- E_trans(false_0,y_z1,x_z1)
     <=[LDC 0,STL x_z1];
```

```
--- E_trans(true_0,y_z1,x_z1)
     <=[LDC 1,STL x_z1];
--- G_trans(gc_0(w_z2,v_z2) :: u_z2,v_z1,u_z1)
     <=[BLOCK([TESTERR]<>(E_trans(w_z2,v_z1,u_z1)<>
          ([STOPERR,CJ "fail"]<>(P_trans(v_z2,v_z1,u_z1)<>
             [J "exitif",DEF "fail"]))
              <>(G_trans(u_z2,v_z1,u_z1)
               <>[DEF "exitif"]))))];
--- G_trans(nil,v_z1,u_z1)
     <=[STOP];
--- M_trans(Y_z2 :: X_z2,V_z1,U_z1)
     <= P_trans(Y_z2,V_z1,U_z1)<>M_trans(X_z2,V_z1,U_z1);
--- M_trans(nil,V_z1,U_z1)
     <= nil;
--- P_trans(while_0(r_z2,W_z2),Y_z1,X_z1)
     <=[LOOP([TESTERR]<>(E_trans(r_z2,Y_z1,X_z1)<>
               ([STOPERR,NOT,CJ "cont",EXIT,DEF "cont"]<>
                    P_trans(W_z2,Y_z1,X_z1))))];
--- P_trans(if_0 V_z2,Y_z1,X_z1)
     <= G_trans(V_z2,Y_z1,X_z1);
--- P_trans(seq_0 s_z2,Y_z1,X_z1)
     <= M_trans(s_z2,Y_z1,X_z1);
--- P_trans(out_0 t_z2,Y_z1,X_z1)
     <=[TESTERR]<>(E_trans(t_z2,Y_z1,X_z1)<>
           [STOPERR,LDC X_z1,LDC chanout Y_z1,OUT]);
--- P_trans(in_0 R_z2,Y_z1,X_z1)
     <=[LDC lookup(R_z2,Y_z1),LDC chanin Y_z1,IN];
--- P_trans(assign_0(T_z2,S_z2),Y_z1,X_z1)
     <=[TESTERR]<>(E_trans(S_z2,Y_z1,X_z1)<>
           [STOPERR,LDL X_z1,STL lookup(T_z2,Y_z1)]);
--- P_trans(stop_0,Y_z1,X_z1)
     <=[STOP];
--- P_trans(skip_0,Y_z1,X_z1)
     <= nil;
end;
```

B.2 The Second Stage

B.2.1 The Eureka Definitions

```
module compdef;
uses aux,multipass,newtrans;
pubfun E_comp,P_comp,G_comp,M_comp,B_comp,C_comp;

dec E_comp : exp_0 # env # adr -> list instr # num;
--- E_comp(e,en,t) <=(E_trans(e,en,t),E_4 e);

dec G_comp : list guard_0 # env # adr -> list instr # list num;
--- G_comp(g,en,t) <=(G_trans(g,en,t),G_4 g);

dec P_comp : proc_0 # env # adr -> list instr # num;
--- P_comp(p,en,t) <=(P_trans(p,en,t),P_4 p);

dec M_comp : list proc_0 # env # adr -> list instr # list num;
--- M_comp(p,en,t) <=(M_trans(p,en,t),M_4 p);

dec B_comp : block_0 # env -> list instr # num;
--- B_comp(b,en) <=(B_trans(b,en),B_4 b);

dec C_comp : prog_0 -> list instr # num;
--- C_comp  p <=(C_trans p,C_4 p);
end;
```

B.2.2 The Script

```
module compscript;
pubfun comp_script;
uses scripts,aux,multipass,newtrans,compdef;
dec comp_script : script;
dec forced_fold_B : script;
dec forced_fold_E : context -> script;
dec forced_fold_G : script;
dec forced_fold_M : script;
dec forced_fold_P : script;
```

```
--- comp_script
     <= choose_eqns[
        (eqspec "E_comp e",
          branch[
           compose[
            unfold(any_term,eqspec "E_trans((e1 dop_0 e2),en,t)"),
            unfold(any_term,eqspec "E_4 e"),
            forced_fold_E context_match
                          "--- E_comp(x dop_0 _,_,_) <= r",
            forced_fold_E any],
           compose[
            unfold(any_term,eqspec "E_trans((mop_0 e),en,t)"),
            unfold(any_term,eqspec "E_4 e"),
            forced_fold_E any],
           compose[
            unfold(any_term,eqspec "E_trans(false_0,en,t)"),
            unfold(any_term,eqspec "E_4 e")],
           compose[
            unfold(any_term,eqspec "E_trans(true_0,en,t)"),
            unfold(any_term,eqspec "E_4 e")],
           compose[
            unfold(any_term,eqspec "E_trans(con_0 c,en,t)"),
            unfold(any_term,eqspec "E_4 e")],
           compose[
            unfold(any_term,eqspec "E_trans(var_0 x,en,t)"),
            unfold(any_term,eqspec "E_4 e")]]),
        (eqspec "P_comp p",
          branch[
           compose[
            unfold(any_term,eqspec "P_trans(while_0(e,p),en,t)"),
            unfold(any_term,eqspec "P_4 p"),
            forced_fold_P,
            forced_fold_E any],
           compose[
            unfold(any_term,eqspec "P_trans(if_0 glist,en,t)"),
            unfold(any_term,eqspec "P_4 p"),
            forced_fold_G],
           compose[
```

```
    unfold(any_term,eqspec "P_trans(seq_0 plist,en,t)"),
    unfold(any_term,eqspec "P_4 p"),
    forced_fold_M],
   compose[
    unfold(any_term,eqspec "P_trans(out_0 ex,en,t)"),
    unfold(any_term,eqspec "P_4 p"),
    forced_fold_E any],
   compose[
    unfold(any_term,eqspec "P_trans(in_0 x,en,t)"),
    unfold(any_term,eqspec "P_4 p")],
   compose[
    unfold(any_term,eqspec "P_trans(assign_0(a,ex),en,t)"),
    unfold(any_term,eqspec "P_4 p"),
    forced_fold_E any],
   compose[
    unfold(any_term,eqspec "P_trans(stop_0,en,t)"),
    unfold(any_term,eqspec "P_4 p")],
   compose[
    unfold(any_term,eqspec "P_trans(skip_0,en,t)"),
    unfold(any_term,eqspec "P_4 p")]]),
(eqspec "G_comp g",
  branch[
   compose[
    unfold(any_term,eqspec "G_trans((g::glist),en,t)"),
    unfold(any_term,eqspec "G_4 g"),
    forced_fold_G,
    forced_fold_P,
    forced_fold_E any],
   compose[
    unfold(any_term,eqspec "G_trans(nil,en,t)"),
    unfold(any_term,eqspec "G_4 g")]]),
(eqspec "M_comp m",
  branch[
   compose[
    unfold(any_term,eqspec "M_trans((p::plist),en,t)"),
    unfold(any_term,eqspec "M_4 p"),
    forced_fold_M,
    forced_fold_P],
```

```
           compose[
            unfold(any_term,eqspec "M_trans(nil,en,t)"),
            unfold(any_term,eqspec "M_4 p")]]),
        (eqspec "B_comp b",
          branch[
           compose[
            unfold(any_term,eqspec "B_trans (blk_0(x,b),en)"),
            unfold(any_term,eqspec "B_4 b"),
            forced_fold_B],
           compose[
            unfold(any_term,eqspec "B_trans (pro_0 p,en)"),
            unfold(any_term,eqspec "B_4 b"),
            fold(all_terms,eqspec "P_comp p")]]),
        (eqspec "C_comp c",
          branch[
           compose[
            unfold(any_term,eqspec "C_trans c"),
            unfold(any_term,eqspec "C_4 c"),
            fold(all_terms,eqspec "B_comp b")]])];


--- forced_fold_B
     <= compose[
          abstract fn "B_4",
          abstract fn "B_trans",
          replace(sbterm("(let p1==e1 in let p2==e2 in e3,"<>
                         " let (p1,p2)==(e1,e2) in e3)",any),
                  "tupling"),
          fold(all_terms,eqspec "B_comp b")];

--- forced_fold_E context
     <= compose[
          abstract sbterm("E_4 x",context),
          abstract sbterm("E_trans(x,x1,x2)",context),
          replace(sbterm("(let p1==e1 in let p2==E_4 e2 in e3,"<>
                         " let (p1,p2)==(e1,E_4 e2) in e3)",any),
                 "tupling"),
          fold(all_terms,eqspec "E_comp e")];
```

```
--- forced_fold_G
     <= compose[
          abstract fn "G_4",
          abstract fn "G_trans",
          replace(sbterm("(let p1==e1 in let p2==G_4 e2 in e3,"<>
                         " let (p1,p2)==(e1,G_4 e2) in e3)",any),
                  "tupling"),
          fold(all_terms,eqspec "G_comp m")];

--- forced_fold_M
     <= compose[
          abstract fn "M_4",
          abstract fn "M_trans",
          replace(sbterm("(let p1==e1 in let p2==M_4 e2 in e3,"<>
                         " let (p1,p2)==(e1,M_4 e2) in e3)",any),
                  "tupling"),
          fold(all_terms,eqspec "M_comp m")];

--- forced_fold_P
     <= compose[
          abstract fn "P_4",
          abstract fn "P_trans",
          replace(sbterm("(let p1==e1 in let p2==P_4 e2 in e3,"<>
                         " let (p1,p2)==(e1,P_4 e2) in e3)",any),
                  "tupling"),
          fold(all_terms,eqspec "P_comp p")];
end;
```

B.2.3 The Result of the Second Stage

```
module newcomp;
pubfun B_comp,C_comp,E_comp,G_comp,M_comp,P_comp;
uses aux,multipass,newtrans;
dec B_comp : block_0 # env -> list(instr) # num;
dec C_comp : prog_0 -> list(instr) # num;
dec E_comp : exp_0 # env # adr -> list(instr) # num;
dec G_comp : list(guard_0) # env # adr -> list(instr) # list(num);
```

```
dec M_comp : list(proc_0) # env # adr -> list(instr) # list(num);
dec P_comp : proc_0 # env # adr -> list(instr) # num;
--- B_comp(pro_0 z_z2,r_z1)
     <= P_comp(z_z2,r_z1,nextfree r_z1);
--- B_comp(blk_0(x_z2,T_z1),r_z1)
     <= let(T_z5,u_z5)==
             B_comp(T_z1,update(r_z1,x_z2,nextfree r_z1))
           in (T_z5,1+u_z5);
--- C_comp t_z1
     <=(B_trans(t_z1,initenv),noofchan + B_4 t_z1);
--- E_comp(r_z3 dop_0 W_z3,y_z1,x_z1)
     <= let(x_z8,t_z7)==E_comp(W_z3,y_z1,x_z1 + 1)
           in let(v_z6,R_z5)==E_comp(r_z3,y_z1,x_z1)
                 in (v_z6<>(x_z8<>[LDL x_z1,LDL(x_z1 + 1),DOP,
                        STL x_z1]),max(R_z5,t_z7 + 1));
--- E_comp(mop_0 t_z3,y_z1,x_z1)
     <= let(w_z6,S_z5)==E_comp(t_z3,y_z1,x_z1)
           in (w_z6<>[LDL x_z1,MOP,STL x_z1],S_z5);
--- E_comp(var_0 z_z4,y_z1,x_z1)
     <=([LDL lookup(z_z4,y_z1),STL x_z1],1);
--- E_comp(con_0 x_z4,y_z1,x_z1)
     <=([LDC x_z4,STL x_z1],1);
--- E_comp(false_0,y_z1,x_z1)
     <=([LDC 0,STL x_z1],1);
--- E_comp(true_0,y_z1,x_z1)
     <=([LDC 1,STL x_z1],1);
--- G_comp(gc_0(V_z2,U_z2) :: w_z2,v_z1,u_z1)
     <= let(V_z8,U_z8)==E_comp(V_z2,v_z1,u_z1)
           in let(S_z7,r_z7)==P_comp(U_z2,v_z1,u_z1)
                 in let(y_z6,U_z5)==G_comp(w_z2,v_z1,u_z1)
                       in ([BLOCK([TESTERR]<>(V_z8<>([STOPERR,
                                CJ "fail"]<>(S_z7<>[J "exitif",
                                  DEF "fail"]))<>(y_z6<>
                                     [DEF "exitif"])))],
                             max(U_z8,r_z7) :: U_z5);
--- G_comp(nil,v_z1,u_z1)
     <=([STOP],nil);
```

```
--- M_comp(Z_z2 :: Y_z2,V_z1,U_z1)
     <= let(R_z7,W_z7)==P_comp(Z_z2,V_z1,U_z1)
           in let(x_z6,w_z5)==M_comp(Y_z2,V_z1,U_z1)
                 in (R_z7<>x_z6,W_z7 :: w_z5);
--- M_comp(nil,V_z1,U_z1)
     <=(nil,nil);
--- P_comp(while_0(t_z2,s_z2),Y_z1,X_z1)
     <= let(T_z7,s_z7)==E_comp(t_z2,Y_z1,X_z1)
           in let(z_z6,V_z5)==P_comp(s_z2,Y_z1,X_z1)
                 in ([LOOP([TESTERR]<>(T_z7<>([STOPERR,NOT,
                          CJ "cont",EXIT,DEF "cont"]<>z_z6)))],
                     max(s_z7,V_z5));
--- P_comp(if_0 S_z2,Y_z1,X_z1)
     <= let(X_z6,W_z5)==G_comp(S_z2,Y_z1,X_z1)
           in (X_z6,maxelm W_z5);
--- P_comp(seq_0 x_z3,Y_z1,X_z1)
     <= let(Y_z6,r_z5)==M_comp(x_z3,Y_z1,X_z1)
           in (Y_z6,maxelm r_z5);
--- P_comp(out_0 z_z3,Y_z1,X_z1)
     <= let(Z_z6,s_z5)==E_comp(z_z3,Y_z1,X_z1)
           in ([TESTERR]<>(Z_z6<>[STOPERR,LDC X_z1,
                  LDC chanout Y_z1,OUT]),max(s_z5,1));
--- P_comp(in_0 Y_z3,Y_z1,X_z1)
     <=([LDC lookup(Y_z3,Y_z1),LDC chanin Y_z1,IN],0);
--- P_comp(assign_0(v_z3,u_z3),Y_z1,X_z1)
     <= let(u_z6,t_z5)==E_comp(u_z3,Y_z1,X_z1)
           in ([TESTERR]<>(u_z6<>[STOPERR,LDL X_z1,
                  STL lookup(v_z3,Y_z1)]),t_z5);
--- P_comp(stop_0,Y_z1,X_z1)
     <=([STOP],0);
--- P_comp(skip_0,Y_z1,X_z1)
     <=(nil,0);
end;
```