Chapter

3 *Using SDL Extensions*

This chapter describes some of the extensions to SDL that are available in the SDL suite, and that are not documented outside Telelogic.

The extensions covered here are the Own and ORef generators, and the algorithmic extensions to SDL. There are other Telelogic-specific extensions to SDL supported in the SDL suite, mainly concerning data types, generators and operators. These extensions are covered in chapter 2, *Data Types*.

Own and ORef Generators

Introduction

A major problem to obtain fast applications generated from SDL is the data model, that requires copying of data in a number of places. In an interchange of a signal between two processes, the signal parameters are first copied from the sending process to the signal and then again copied from the signal to the receiver. If the two processes have access to common memory, it would be possible only to pass a reference to the data via the signal, and in that way there would be no need to perform the two copy actions.

The generator Ref can be used for this purpose (see "The Ref Generator" on page 112 in chapter 2, *Data Types*), but there is a number of important problems when using the Ref generator:

- The user has to deallocate memory when it is no longer used. If the user forgets this in any circumstance, memory will be lost.
- It is easy to, intentionally or unintentionally, access the same memory from several process instances. This is very bad practice in real-time programming (without protection for simultaneous access to the memory) and might cause unwanted behavior. These kinds of errors are usually very difficult to find.

Basic Properties of the Own Generator

The purpose of the Own generator is to solve the situation described above, i.e. it should be possible to limit the number of copy operations that are needed, at the same time as the user should not need to worry about memory deallocation, and simultaneous access to the memory from several processes should not be possible.

The basic property of the Own generator that makes this possible is that only one Own pointer at a time can refer to the same memory. This variable (of Own type) is referred to as the owner of the memory. Ownership is passed to another variable by assignment.

Example 37: Own variables ––––––––––––––––––––––––––––––––––

```
newtype Data struct
  a, b integer;
endnewtype;

newtype Own_Data Own(Data)
endnewtype;

dcl
  v1 Own_Data,
  v2 Own_Data;

task v1 := v2;
```

This assignment is interpreted as follows:

- If v1 refers to some memory this memory is deallocated.
- v1 is assigned the value of v2, i.e. refers to the same memory as v2.
- v2 is set to Null.

––

By this scheme the basic properties of the Own generator is preserved, i.e. all memory no longer accessible is deallocated and there is only one reference to the data.

To handle more complex cases, the order in which these operations are performed is a bit more complicated. With the same types and variables as in the example above and the procedure P, taking three Own_Data parameters:

```
task v1 := P(v2, v1, v2);
```

we get the following execution:

Evaluation of the right-hand side is performed from left to right, i.e. starts with the first actual parameter of P. The first formal parameter of P is assigned the value of v2 and takes the ownership of this memory. The variable v2 is assigned the value Null. The same thing happens for the second formal parameter and the variable v1. The third formal parameter of P get the value Null as v2 is Null at this point.

Now the procedure P is called and its return value is obtained. Before assigning this value to the variable on the left hand side, i.e. to v1, the memory currently referred to by v1 is deallocated. In this case v1 is Null at this point as the second formal parameter of P already have taken the ownership of this memory. Last the variable v1 is assigned the value returned by P.

Ownership is, as can be seen in the example above, passed not only in assignments, but in every case where the reference is assigned to some other place. This happens for example in assignments, input, output, set, reset, procedure call, and create. The only places where ownership is **not** passed is:

- when the explicit copy function is used, for example
  `task v1 := copy(v2);`
- in import, export, and view operations, which are interpreted as containing implicit copy operations.
- in calls to the standard functions '=' and "/=".

Note that `copy(v2)` is a "deep" copy, i.e. any Own pointers in the copied data structure are also copied. Otherwise we would end up with several references to the same memory.

Definition of Own Generator

The Own generator is defined in SDL according to the following:

```
GENERATOR Own (TYPE Itemsort)
  LITERALS
     Null;
  OPERATORS
    "*>"  : Own, ItemSort -> Own;
    "*>"  : Own -> ItemSort;
    make! : ItemSort -> Own;
  DEFAULT Null;
ENDGENERATOR Own;
```

Basically the Own generator is a way to introduce pointers to allocated memory. The Null value is as usual interpreted as "a reference to nothing". The operators "*>" are the Extract! and Modify! operators, i.e. the way to reference or modify the memory referred to by the pointer. Using the type and variables in the previous example the following statements are correct:

```
task v1*>!a := 1;
task i := v2*>!b;
   /* integer assignment, i is integer variable */
task d := v2*>;
   /* struct level assignment, d is of type Data */
```

The "*>" operator have the same properties as the '*' in C, i.e. "v1*>" has the same meaning as "*v1" in C. To make the syntax a bit easier

there is a possibility to let the SDL Analyzer implicitly insert the "*>" in the expressions where it is needed. The example above would then become:

```
task v1!a := 1;
task i := v2!b;
   /* integer assignment, i is integer variable */
task d := v2;
   /* struct level assignment, d is of type Data */
```

which is a bit easier to read. More details about the implicit type conversions can be found in "Implicit Type Conversions" on page 134.

Before it is possible to start working with components in the data referenced by the Own pointer, the Own pointer must be initialized with a complete value (default is Null as can be seen in the definition). The Make! operator is a suitable way to initialize a variable. As usual the concrete syntax for Make! is "(. x .)", where x should be replaced by a value of the ItemSort for the current Own pointer.

Example of intializations using the data definitions in the examples above:

```
dcl v1 Own_Data := (. (. 1, 2 .) .);
task v1 := (. (. 5, 5 .) .);
```

The inner "(. .)" is for the constructing the struct value and the outer "(. .)" is for the Own make! function. It is, however, possible to avoid the double parentheses as there is an implicit type conversion from a type T to Own(T), by implicitly inserting "(. .)" around a value of type T. So the examples above could (and probably should) be written as

```
dcl v1 Own_Data := (. 1, 2 .);
task v1 := (. 5, 5 .);
```

Again, please see "Implicit Type Conversions" on page 134. The other operations available for own pointers, apart from "*>" and make!, are assignment, test for equality, and copy. The assign operator has already been described above. Test for equality ('=' and "/=") does NOT test for pointer equality as two Own pointers cannot be equal. Instead equality is "deep" equality, i.e. the values referred to by the Own pointers are compared.

An implicit copy operator has been inserted for every type. It takes a value and returns a copy of that value. For all types that are not Own pointers or contain Own pointers, this operator is meaningless as it just

returns the same value. For Own pointers or for structured values containing Own pointers, the copy function, however, copies the values referenced by the Own pointers.

The ORef Generator

The ORef generator is intended to be used together with the Own generator to provide a way to temporary refer to owned data during some algorithm, without affecting the ownership of the memory. If, for example, Own pointers is used to create a linked list and we would like to write a procedure that calculates the length of the list, then we need a temporary pointer going through the list. If that pointer was a Own pointer the list would be destroyed while we traverse the list, as there may be only one Own pointer referring to the same memory.

Example 38: Own and ORef ––––––––––––––––––––––––––––––––

```
newtype ListElem struct
  Data MyType;
  Next ListOwn;
endnewtype;

newtype ListOwn Own(ListElem)
endnewtype;
newtype ListRef ORef(ListElem)
endnewtype;

procedure ListLength; fpar Head ListRef;
returns integer;
dcl
  Temp ListRef,
  Len  Integer;
start;
  task Len := 0, Temp := Head;
  again :
  decision Temp /= null;
  (true) :
    task Len := Len+1, Temp := Temp!Next;
    join again;
  (false) :
  enddecision;
  return Len;
endprocedure;

dcl
  MyList ListOwn,
  L integer;

task L := call ListLength(MyList);
```

––

Note the use of the ListRef type both in the formal parameter Head and in the local variable Temp. If Head would be of ListOwn type, the variable MyList would be null after the call of ListLength, which is not what we intended. If ListOwn was used as type for the variable Temp, as the statement `Temp := Temp!Next` would unlink the complete list.

Another example of a typical application of ORef, is to introduce backward pointer in a linked list, to make it doubly linked. If the forward pointers are Own pointers then the backward pointers cannot be Own pointers as then we would have two Own pointers on the same object.

The ORef generator is defined as:

```
GENERATOR ORef (TYPE Itemsort)
  LITERALS
     Null;
  OPERATORS
    "*>"  : ORef, ItemSort -> ORef;
    "*>"  : ORef -> ItemSort;
    "&"   : ItemSort -> ORef;
    "="   : ORef, ItemSort -> Boolean;
    "="   : ItemSort, ORef -> Boolean;
    "/="  : ORef, ItemSort -> Boolean;
    "/="  : ItemSort, ORef -> Boolean;
  DEFAULT Null;
ENDGENERATOR;
```

Where “*>” is used for dereferencing and ‘&’ is an address operator.

Run-Time Errors

There are four situations, concerning Own and ORef, that can lead to a run-time error. These situations are:

- Dereferencing of a null pointer.
- An ORef pointer that refers to an object that has been deallocated.
- An ORef pointer that refers to an object that is owned by another process.
- A cycle of Own pointers is created, as this memory can never be deallocated.

These problems are all found during simulation and validation, except that if an ORef pointer refers to a data area that is first deallocated and then allocated again, the ORef pointer is no longer invalid.

Examples of run-time error situations assuming the data types in the previous section.

```
dcl
  L1, L2 ListOwn,
  R1, R2 ListRef,
  I      Integer;

task
  L1 := null,
  R1 := null,
  L1!Data := 1,
    /* ERROR: Dereferencing of Null pointer */
  I := R1!Data;
    /* ERROR: Dereferencing of Null pointer */

task
  L1 := (. 1, null .),
  R1 := L1,
  L1 := null,
  I := R1!Data;
    /* ERROR: Reference to deallocated memory */

task
  L1 := (. 1, null .),
  R1 := L1;
output S(L1) to sender;
task
  I := R1!Data;
    /* ERROR: Reference to memory not owned by
       this process */

task
  L1 := (. 1, null .),
  L1!Next := (. 2, null .),
  L1!Next!Next := L1;
    /* ERROR: Loop of own pointers created */
```

Implicit Type Conversions

Note:

Implicit type conversions are by default off in the SDL Analyzer. It can be turned on in the Analyze dialog in the Organizer. Implicit type conversions will, however, influence the time needed for semantic analysis. The more complex an expression is, the more effect on time the implicit type conversions will have, as the number of possibilities increases (often exponentially) with the length of the expression.

The purpose of the implicit type conversions is to simplify the use of the Own generator. The code that operate on data structures should be the same if you use a data structure T or if you use a own pointer to T, own(T). The only thing that a user has to think about is if ownership should be passed or if a copy should be made, when passing data to somewhere else.

The implicit conversions never change the type of, for example, an assignment. If there is an assignment:

```
task R1 := L1;
```

no implicit type conversions are applied on R1, as that would change the type of the assignment. Type conversions might be applied on L1 in the right hand side, to obtain the correct type. In an assignment:

```
task Q(a) := ...;
```

the implicit type conversions might also be applied to the index expression, i.e. to a.

In a test for equality and in similar situations, e.g. in:

```
L1 = R1
```

implicit type conversions are first applied to the left expression, i.e. to L1. If that yields a correct interpretation, that one is selected. Otherwise implicit type conversions are tried on the right expression, i.e. to R1.

Assume a type T and a two generator instantiations `Own_T = Own(T)` and `ORef_T = ORef(T)`. Assume also the variables:

```
dcl
  t1 T,
  v1 Own_T,
  r1 ORef_T;
```

Then the following implicit type conversions are possible:

```
1. Own_T  -> T,       by v1 -> v1*>
2. T      -> Own_T,   by t1 -> (. t1 .)
3. Own_T  -> ORef_T,  by v1 -> demote(v1)
4. ORef_T -> Own_T,   by r1 -> (. r1*> .)
5. T      -> ORef_T,  by t1 -> &t1
6. ORef_T -> T,       by r1 -> r1*>
```

Type conversions 1 and 6 make it possible to exclude "*>" in component selections. Instead of writing

```
a*>!b*>(10)*>!c
```

it is possible to write

```
a!b(10)!c
```

This possibility also exists for ordinary Ref pointers.

Type conversion 2 makes it possible to assign an Own pointer a new value of the Own pointer component type. If A is a Own pointer to a struct containing two integers, then it is possible to write:

```
A := (. 1, 2 .);
```

which means the same as

```
A := (. (. 1, 2 .) .);
```

where the inner "(. .)" is the make! function for the struct and the outer "(. .)" is the make! function for the Own pointer.

This possibility also exists for ordinary Ref pointers.

Type conversion 3 makes it possible to assign an ORef pointer to an Own pointer. This is already used in the examples above, but is not directly possible, as an ORef and an Own pointer are two distinct types. The `demote` operator converts a Own pointer to the corresponding ORef pointer. (Corresponding means the first ORef with the same component type in the same scope unit as the Own pointer type is defined).

Type conversion 4 makes it possible to construct a new Own value, starting from a ORef value. The conversion is performed in two steps, first going from ORef_T to T by applying conversion 6, and then from T to Own_T by applying conversion 2.

Type conversion 5 makes it possible to let a ORef_T pointer refer to a DCL variable by writing:

```
task r1 := t1;
```

which means the same thing as

```
task r1 := &t1;
```

where '&' is the address operator (as in C).

Algorithms in SDL

A former problem in SDL is the lack of support for writing algorithms. For pure calculations, not involving communication, the graphical form for SDL tends to become ordinary flow charts, which is usually not a good way to describe advanced algorithms. Such, often large, parts of an SDL diagram also hide other, from the SDL point-of-view, more important parts of the diagram, namely the state machine and the communication aspects.

The algorithmic extensions described here addresses these problems by introducing the possibility to write algorithms in textual form within a Task symbol, and also to define procedures and operators in textual form in text symbols. There are two major advantages with this approach, compared to ordinary SDL:

- The algorithms are written in a compact form, similar to ordinary programming languages, and will therefore not hide other important aspects of an SDL diagram.
- The language used within the algorithms contains more powerful algorithmic constructs than ordinary SDL, like if-then-else, and loop statements.

In addition, the algorithmic extensions make it possible to now define procedures and operators in textual form in text symbols in SDL/GR.

These algorithmic extensions to SDL have been approved by ITU Study Group 10 to be incorporated into the official Master List of Changes that will affect the next ITU recommendation for SDL. There are a few minor differences in the support for SDL algorithms in the SDL suite compared with the ITU definition – these are noted in the descriptions below.

The constructs that are part of the extensions are:

- Compound Statement
- Local Variables
- If Statements
- Decision Statements
- Loop Statements
- Label Statements
- Jump Statements
- Empty Statements

Compound Statement

The basic concept in the algorithmic extensions is the *compound statement*. A compound statement starts with a '{', which is followed by a sequence of variable declarations and a sequence of statements, and it then ends with a '}'.

A compound statement may be used in three places:

- as the contents of a TASK,
- as the body of a procedure or operator definition in a text symbol,
- as a statement within an enclosing compound statement.

Note:

According to the ITU language definition the body of a procedure or operator is allowed to be a statement. In the SDL suite, however, a compound statement is required. This means that if the body consists of only one statement, the enclosing "{ }" are required anyhow.

Note also that the enclosing "{ }" should not be included in a Task symbol in the SDL Editor. These braces will be added when the SDL system is converted to SDL/PR for analysis.

Example 39 ----------

Contents in Task symbol in SDL/GR:

```
a := b+1;
if (a>7) b := b+1;
```

Corresponding code in SDL/PR:

```
task {
  a := b+1;
  if (a>7) b := b+1;
};
```

Example 40 ----------

A procedure in a text symbol in SDL/GR, or in SDL/PR:

```
procedure p fpar i integer returns integer
{
  if (i>0)
    i := i+1;
  else
    i := i-1;
  return i;
}
```

Local Variables

Within a compound statement it is allowed to define a number of local variables. These variables will be created when the compound statement is entered and will be destroyed when the compound statement is left. The semantics of a compound statement is very much like a procedure without parameters, which is defined and called at the place of the compound statement.

A variable declaration within a compound statement looks that same as ordinary variable declarations, except that “exported” and “revealed” are not allowed. Example:

```
dcl
  a, b integer := 0,
  c    boolean;
```

Statements

A statement within a compound statement my be of any of the following types:

- compound statement
- output statement
- create statement
- set statement
- reset statement
- export statement
- return statement (only in procedures and operators)
- procedure call statement
- assignment statement
- if statement
- decision statement
- loop statement
- label statement
- jump statement
- empty statement

Note that each statement (and each variable declaration statement) ends with a ‘;’. The following statement types use the same syntax as in ordinary SDL/PR:

output, create, set, reset, export, return, call, assignment

Example 41: Ordinary SDL/PR statements

```
output s1(7) to sender;
output s2(true, v1) via sr1;
create p2(11);
set(now+5, t);
reset(t);
export(v1);
return a+3;
call prd1(a, 10);
a := a+1;
```

> **Note:**
> According to the ITU language definition the keyword `call` in a procedure call is optional. In the SDL suite it is, however, required.

If Statements

The structure of an if statement is:

```
if ( <Boolean expr> )
  <Statement>
else
  <Statement>
```

where the else part is optional. The Boolean expression is first calculated. If it has the value true, the first statement is executed, otherwise the else statement, if present, is executed.

Example 42

```
if (a>0)
  a := a+1;

if (a=0) {
  a := 100;
  b := b+1;
} else {
  a := a+1;
  b := 0;
}
```

If there are several possible if statements for an else path (the “dangling else” problem), the innermost if is always selected.

Example 43 ----------

```
if (a>0)
  if (b>0)
    a := a+1;
  else
    a := a-1;
```

means:

```
if (a>0) {
  if (b>0)
    a := a+1;
  else
    a := a-1;
}
```

Decision Statements

A decision statement has much in common with the ordinary decisions found in SDL, i.e. it is a multi-branch statement. The major differences between decision statements and ordinary statements is that all paths in a decision statement ends at the enddecision.

Example 44 ----------

```
decision (a) {
(1:10) : {call p(a); a := a-5;}
(<=0)  : a := a+5;
else   : a := a-5;
}
```

The decision question and the decision answers follows the same syntax and semantics as in ordinary decisions. Following an answer there should be a statement, which might be a compound statement.

Loop Statements

A loop statement is used to repeat the execution of a statement (or usually a compound statement), a number of times. The loop is controlled by a loop variable, which either can be locally defined in the loop or defined somewhere outside of the loop.

The loop control part contains three fields:

- the loop variable indicator with a start value,
- the loop test expression,
- the new loop variable value.

Example 45 ––

```
for (a := 1, a<10, a+1)
  sum := sum+a;
```

should be interpreted as (in C-like syntax):

```
a = 1;
while (a<10) {
  sum = sum+a;
  a = a+1;
}
```

––

Note the difference between SDL and C when it comes to the variable update. In C this is a statement, in SDL it is an expression to be assigned to the variable mentioned in the loop variable indicator.

In the loop variable indicator, either a new variable can be defined or a previously defined variable can be used. Example:

```
for (a := 1, ...
for (dcl a integer := 1, ...
```

Other possibilities in loop statements:

- One or more of the loop control part fields can be empty. If, however the loop variable indicator (first field) is empty, then the loop variable update field (third field) must also be empty. Example:

```
for (a := 1, , a+1) ..
for ( , , ) ...
```

- A loop may contain several loop control parts. Example:

```
for (a := 1, a<10, a+1; b := 1, b<5, b+1)
  sum := sum+a+b;
```

This should be interpreted as (in C like syntax):

```
a = 1;
b = 1;
while ( (a<10) and (b<5) ) {
  sum = sum+a;
  a = a+1;
  b = b+1;
}
```

- Break statements can be used to break out of a loop. See "Label Statements" on page 143 and "Jump Statements" on page 144.
- A loop statement may end with a "then" statement, which is executed if the loop is terminated because of the loop test expression becomes false. The "then" statement is not executed if the loop is terminated due to a break statement. Example:

```
ok := false;
for (a:=1, a<10, a+1) {
  sum := sum+arr(a);
  if (sum > limit) break;
}
then
  ok := true;
```

Label Statements

A label statement is just a label followed by a statement. These labels are only of interest if the statement following the label is a loop statement. The label name can be used in break statements (see below) to break out of a loop statement. Example:

```
L:
for (i:=0, i<10, i+1)
  sum := sum+a(i);
```

Note:

There are no "join" or "goto" statements allowed in the algorithmic extensions to SDL.

Jump Statements

Jump statements, i.e. `break` and `continue`, are used to change the execution flow within a loop.

A continue statement, which only may occur within a loop, is defined as: “skip the remaining part of the loop body and continue with updating the loop variable to its next value.”

Example 46 ----------

```
for (a:=1, a<10, a+1) {
  if (sum > limit) continue;
  sum := sum+arr(a);
}
```

should be interpreted as (in C like syntax):

```
a = 1;
while (a<10) {
  if (sum > limit) goto cont;
  sum = sum + arr[a];
cont :
  a = a+1;
}
```

A break statement can be used to stop the execution of the loop and directly goto the statement after the loop.

Example 47 ----------

```
ok := false;
for (a:=1, a<10, a+1) {
  sum := sum+arr(a);
  if (sum > limit) break;
}
then
  ok := true;
```

should be interpreted as (in C like syntax):

```
ok = false;
a = 1;
while (a<10) {
  sum = sum + arr[a];
  if (sum > limit) goto brk;
  a = a+1;
}
ok = true;
brk:
```

A break statement breaks out of the innermost loop statement. By using labeled loop statements breaks out of outer loops can be achieved.

Example 48 ----------

```
L: for (x:=1, x<10, x+1) {
     a := 0;
     for (y:=1, y<10, y+1) {
       a := a+y;
       if (call test(x,y)) break L;
     }
   }
```

The break statement in the inner loop breaks out from both loops as it mentions the label for the outer loop.

Empty Statements

It is allowed to have an empty statement, represented by just writing nothing. This is sometimes useful, for example as loop statement:

```
for (i:=1, Arr(i)/=0 and i<Limit, i+1) ;
  /* This loop sets i to the index of the first zero
     element in the Array Arr. */
```

Grammar for the Algorithmic Extensions

Meta grammar:

`'dcl', ')', ';'`	are examples of terminal symbols.
`<Stmt>, <Name>`	are examples of non-terminal symbols.
`::=`	means defined as.
`$`	means used for empty.
`*`	means 0 or more occurencies.
`+`	means 1 or more occurencies.
`|`	means or.

Start of Grammar ----------

```
<CompoundStmt> ::=
   '{' <VarDefStmt>* <Stmt>* '}'

<VarDefStmt>   ::=
   'dcl'     <Name> (',' <Name>)* <Sort> (':=' <Expr> | $)
       ( ',' <Name> (',' <Name>)* <Sort> (':=' <Expr> | $) )*
   ';'

<Stmt>         ::=
   <CompoundStmt>       |
   <Outputx> ';'        |
   <CreateRequest> ';'  |
   <Setx> ';'           |
   <Resetx> ';'         |
   <Export> ';'         |
   <Return> ';'         |
   <ProcedureCall> ';'  |
   <IfStmt>             |
   <LabelStmt>          |
   <AssignmentStatement> ';' |
   <DeciStmt>           |
   <LoopStmt>           |
   <JumpStmt> ';'       |
   <EmptyStmt> ';'

<IfStmt>       ::=
   'if' '(' <Expr> ')' <Stmt> ('else' <Stmt> | $)

<DecisionStmt> ::=
   'decision' '(' <Expr> ')' '{'
   ( <Answer> <Stmt> )+
   ( 'else'   <Stmt> | $)
   '}'
```

```
<Answer>        ::= same as answer in ordinary decisions

<LoopStmt>      ::=
   'for' '(' (<LoopClause> (';' <LoopClause>)* | $) ')'
     <Stmt>
   ('then' <Stmt> | $)

<LoopClause>    ::=
   (<LoopVarInd> | $) ',' (<Expr> | $) ',' (<Expr> | $)

<LoopVarInd>    ::=
   'dcl' <Name> <Sort> ':=' <Expr> |
   <Identifier> (':=' <Expr> | $)

<LabelStmt>     ::=
   <Label> <Stmt>

<JumpStmt>      ::=
   <Break> (<Name> / $) |
   <Continue>

<EmptyStmt>     ::=
   $
```

End of Grammar ––

Algorithms in Simulator/Validator

The textual trace in the SDL Simulator and the SDL Validator for the new algorithmic extensions will be according to the table below.

Statement	Textual trace	Comment
Compound		No trace
If	`IF (true)` `IF (false)`	
Decision	`DECISION  Value: 7`	Same trace as for ordinary decisions

Loop	`LOOP variable b := 3`
	`LOOP test TRUE`
	`LOOP test FALSE`
		For loop variable assignments
		For loop tests

Jump	`CONTINUE`
	`BREAK`
	`BREAK LoopName`	

Empty		No trace

A compound statement without variables declarations is seen as just a sequence of statements, while a compound statement with variable declarations is seen as a procedure call of a procedure with no name, without parameters. However, no trace information is produced for this implicit procedure call or procedure return.

When it comes to variables, these are available in the simulator interface just in the same way as if compound statements where true procedures. That is, the commands *Up* and *Down* can be used to view variables in different scopes. Note that a variable defined in a loop variable indicator introduces a scope of its own.

There is one exception of this general treatment of variables in local scopes and that is procedures defined as a compound statement.

In this procedure:

```
procedure p
  fpar in/out a integer
{
  dcl b integer;
  ...
}
```

the parameter `a` and the variable `b` will be in the same scope, the procedure scope. For compound statements within the outermost procedure scope the general rules above apply.

Execution Performance in Applications

Cadvanced

All concepts in the algorithmic extensions have efficient C implementations, except variable declarations in local scopes (including in a loop variable indicator), as such compound statements will become SDL procedure calls.

Cmicro

All concepts in the algorithmic extensions have efficient C implementations. Compound statements containing variables are implemented using C block statements.