Sequential and Parallel Exception Handling in Modula-3*: A Unifying Semantics Specification

Ernst A. Heinz

Institute for Program Structures and Data Organization, University of Karlsruhe

P.O. Box 6980, D-76128 Karlsruhe, F.R. Germany

(email: heinze @ ira.uka.de)

Abstract

The orthogonal integration of explicit parallelism and structured exception handling involves several non-trivial design decisions and questions as for language semantics. In the case of Modula-3*, we discuss and solve the issues related thereto by presenting a unifying specification of operational semantics for sequential and parallel exception handling.

We call the specification *unifying* because it covers the complete semantics of both kinds of exception handling by the same definitions. At the heart thereof lies a concept of *process groups* serving as the general units of execution. This concept allows for astonishingly simple but nevertheless exact and consistent descriptions of both sequential and parallel computations. Thus, process groups prove to be the unifying "glue" of the whole approach.

Keyword Codes: D.3.3; F.1.2; F.3.2

Keywords: Language Constructs; Modes of Computation: Parallelism; Semantics of Programming Languages

1 Introduction

Many language designers and programmers regard mechanisms for exception handling as inherently complex programming techniques that are hard to use, hard to understand, and hard to implement. Fortunately, the designers of Modula-3 did not give in to such prejudice and integrated structured exception mechanisms into their new programming language [3]. Today, exceptions enjoy wide-spread use and apprehension within the Modula-3 community. Their efficient sequential implementation within Digital Equipment's SRC Modula-3 programming environment [10] has proven all critics and scepticism wrong.

The current situation of explicitly parallel programming shares some similarities with that of exception handling mechanisms as described above. Most programmers do not know much about parallelism at all — nevertheless they presume that it is hard to use, hard to understand, and hard to implement. We do not share these opinions and try to improve on the situation with combined efforts in programming language design [8, 11, 14], compiler construction [6, 11], and hardware architecture [11].

Considering these situations, our proposal of combining exception handling and parallel programming in a single language framework will surely be confronted with suspicion if not outright defiance. However, the integration of both language features logically follows from our previous works that, right from the beginning, took issues of orthogonal design seriously. Moreover, we felt especially challenged by the possible benefits of a successful integration.

The remainder of this article is organized as follows. After discussing related work in Section 2, we introduce the explicitly parallel programming language Modula-3* in Section 3. Then we present a unifying, exception-based semantics specification for sequential and parallel execution of all Modula-3* statements in Section 4 and Section 5, before finally concluding.

2 Related Work

Since the mid-1960s, numerous authors and publications have continously elaborated on the design and semantics of effective mechanisms for sequential exception handling. In 1985, for example, Yemini and Berry introduced a modular, verifiable exception handling technique [15]. Beside other results, this large volume of research lead to the integration of quite structured exception handling schemes into the abstract data type framework and into several popular, modern programming languages like Ada, C++, Eiffel, ISO Modula-2, and Modula-3.

Similarly, the semantics of most concurrent and parallel programming constructs are equally well understood [1, 2, 4, 9, 13]. As for parallel language design, the work of Hagerup et al. on FORK [7] and the recent work of Prakash et al. [12] on UC closely resemble our approach that originated with Modula-2* back in early 1990. UC is a data-parallel extension of C featuring a shared address space, built-in data-parallel standard operations, and two new, explicitly parallel statements. Interestingly enough, these new `par` and `arb` statements of UC are effectively equivalent to the synchronous and asynchronous `FORALL`s of Modula-2*.

So far, however, we have not found any other work aiming at the formulation of a unifying semantics specification for both sequential and parallel exception handling.

3 Modula-3*

Modula-3* [8] is an extension of Modula-3 that incorporates the general PRAM programming model [1, 5, 13] into a modern, high-level programming language. It supports all Modula-3 abstraction mechanisms for parallel programming. Arbitrary nestings of synchronous, asynchronous, and mixed-mode parallel computations operate on data in the same single address space. Procedures may be called in any context, whether sequential or parallel, at any nesting depth, allowing for recursive parallelism without restriction.

Modula-3* defines only two new language constructs that are remarkably compact and as easy to use as to learn. The additional grammar rules for Modula-3* with respect to the original Modula-3 grammar from [3] are listed below.

```
ForallSt  ::=  FORALL RangeDecl {RangeDecl} [NOSYNC] DO
                 S
               END.
```

```
NoSyncSt  ::=  NOSYNC
                 S
               END.
RangeDecl ::=  IdList ":" Type.
Stmt      ::=  ForallSt | NoSyncSt.
```

3.1 Process Groups

In Modula-3* the general units of execution are not just single processes but whole groups thereof. Such *process groups* always contain a finite number of processes. All processes of a process group *PG* are tightly coupled, executing their respective statement sequence in parallel synchrony. Except for procedure calls, Modula-3* adopts the MSIMD semantics [1, 11, 13] of Modula-2* [6, 13, 14] for synchronous expression evaluation and branch execution. For the sake of completeness and transparency, Section 4 re-formulates the semantics of the latter based on process groups. Closely following the original Modula-3 language definition, we specify the exception-based, synchronous semantics for all remaining statements in Section 5.

The process model underlying synchronous execution is quite simple; it requires only two different states: *active* and *intercepted*. Active processes show "normal" behavior and really perform the actions prescribed by the current statement. Intercepted processes, however, merely accompany the active ones without performing any actions but idling instead.

Initially, every process is active and such it remains until being intercepted by the execution of a RAISE statement. The only way for an intercepted process to become active again is by reaching an appropriate handler that catches the raised exception (see Section 5.4). The cycle from activity to interception and vice versa may then start over again. Consequently, process groups are permanently divided into disjoint subsets of active and intercepted processes.

Discussion. In principle, process groups are "virtual" PRAMs consisting of dynamically changing numbers of processors. Sequential execution fits into the process group framework by modelling through so-called *sequential process groups* that contain only a single process. In order to be really unifying, our specifications must therefore ensure the equivalence of traditional, sequential semantics and its corresponding sequential process group semantics.

Notation. Because of its frequent use we like to introduce the convenient notation $X^{|p}$ (read as "*X* under *p*") for the concrete value resulting from evaluation of expression *X* by process *p*.

3.2 FORALL Statements

In Modula-3* parallel activity is invoked by means of the FORALL statement.

```
FORALL J: T DO
  S
END
```


On encountering the above FORALL, a process group *PG* first evaluates the possibly non-static scalar range type *T*. Secondly, each process $p \in PG$ creates a set $PG'_{[p]}$ of NUMBER($T^{|p}$) new processes and replaces itself by them. The identifier *J* is local to the FORALL and serves as a runtime constant; it is initialized to a unique value $J^{|q} \in T^{|p}$ for each new process $q \in PG'_{[p]}$.

Thereafter, the enlarged process group $PG' = \bigcup_{p \in PG} PG'_{[p]}$ executes statement sequence *S*. Finally, a process group reduction of PG' takes place at the END of the FORALL. This step restores *PG* by replacing all processes of $PG'_{[p]}$ in PG' with their original creator *p* while reducing exception information according to the rules from Section 5.2.

Discussion. Obviously, Modula-3* FORALL statements create as many new processes as there are elements in the range type *T*. The created processes execute statement sequence *S* in parallel synchrony. The END of a FORALL imposes an explicit synchronization barrier on the created processes. Termination of the whole FORALL statement is delayed until all processes have finished their respective execution of statement sequence *S*.

3.3 NOSYNC Statements

In addition to synchronous parallelism, Modula-3* supports asynchronous and mixed-mode parallel computations by providing the NOSYNC statement.

```
NOSYNC
  S
END
```

A process group *PG* executes the above NOSYNC by first partitioning itself into $n = |PG|$ new groups $PG_p = \{p\}$, namely one for every $p \in PG$. Secondly, all new process groups PG_p execute statement sequence *S* completely independent of one another (preferably in parallel, of course). Finally, each group waits at the END for all others to also finish their respective executions of statement sequence *S*. Thus, all process groups PG_p synchronize before they merge and restore *PG* upon leaving the whole NOSYNC statement.

Discussion. In sequential or asynchronous contexts, $|PG| = 1$ and $PG_p = PG$ holds. Hence, NOSYNCs have no semantic meaning therein at all. In other contexts, however, NOSYNCs temporarily disable the synchrony of process groups for the scope of statement sequence *S*; termination of the whole NOSYNC is delayed until all entering processes have finished their respective executions of *S*. This explicit synchronization barrier at the END re-establishes the synchrony of the process groups before they finally leave the whole NOSYNC statement.

3.4 Shorthand Notations

Modula-3* supports convenient shorthand notations for multi-range, directly nested, and completely asynchronous FORALLs. While greatly improving the readability and compactness

of parallel computations, they only require the additional non-terminal *RangeDecl* in the grammar to define their syntax.

Multi-Range FORALLs

```
FORALL RangeDecl1 ... RangeDeclk [NOSYNC] DO
  S
END
```

⇓ ⇑

```
FORALL RangeDecl1 [NOSYNC] DO
  ...
    FORALL RangeDeclk [NOSYNC] DO
      S
    END;
  ...
END
```

Directly Nested FORALLs

```
FORALL Id1 "," ... Idk ":" Type [NOSYNC] DO
  S
END
```

⇓ ⇑

```
FORALL Id1 ":" Type [NOSYNC] DO
  ...
    FORALL Idk ":" Type [NOSYNC] DO
      S
    END;
  ...
END
```


Completely Asynchronous FORALLs

```
FORALL Id ":" Type NOSYNC DO
  S
END
```

⇓ ⇑

```
FORALL Id ":" Type DO
  NOSYNC
    S
  END;
END
```

Remark. The above equivalences show that all Modula-3* shorthand notations are indeed "syntactic sugar". They do not introduce any new semantic concepts because purely syntactic rewrite rules suffice to define their meaning.

3.5 Programming Examples

The first example illustrates the basic form of `FORALL` statements. It implements a parallel permutation of vector `X` according to permutation function `p()`.

```
FORALL i: [1..n] DO
  X[i] := X[p(i)];
END
```

The synchronous semantics ensure that all RHS elements `X[p(i)]` are read and temporarily stored before any LHS variable `X[i]` is written. Thus, parallel modifications of overlapping data structures are easily formulated in Modula-3*.

In contrast thereto, a vector addition $Z = X + Y$ needs no inherent synchronization. It can be formulated as an asynchronously parallel computation for non-overlapping arrays `X`, `Y`, `Z`.

```
FORALL i: [1..n] NOSYNC DO
  Z[i] := X[i] + Y[i];
END
```

The following example employs the shorthand notation for directly nested FORALLs in order to easily implement a parallel matrix transpose.

```
FORALL i, j: [1..n] DO
  A[i,j] := A[j,i];
END
```

If placed inside FORALLs, declarations in block statements introduce local variables for all created processes. In the next example, such local variables Xi serve as temporaries to store the initial value of X[*i*] for each process *i*. The whole example performs an element-wise swap of the two vectors X and Y in parallel.

```
FORALL i: [1..n] NOSYNC DO
  VAR Xi := X[i];
  BEGIN
    X[i] := Y[i];
    Y[i] := Xi;
  END;
END
```

Local declarations greatly simplify the composition and analysis of parallel Modula-3* programs. Furthermore, they may improve performance; especially local variables of processes are often crucial for the efficiency of parallel algorithms because of their guaranteed collocation with computation.

Last but not least, we present a non-trivial image processing problem and its parallel solution in Modula-3*. The tasks are to smooth a symmetric, two-dimensional image by application of a spatial filter, to store the smoothed image for later use, to distort the smoothed image by combination with a distortion matrix, to mirror the distorted image at its diagonal, to additively combine the mirrored image with the un-smoothed original — yielding a new image — and, finally, to transpose the distortion matrix.

Let O denote the pixel matrix of the original image, D the original distortion matrix, R the pixel matrix of the resulting image, and f the function for spatial filtering. Then, the above tasks can be solved by calculating O', D', and R as follows.

$$O' = f(O)$$
$$D' = D^T$$
$$R = O + \left(f(O) \cdot D\right)^T$$

In Modula-3* we use two-dimensional REAL arrays for the original image matrix O, the distortion matrix D, and the resulting image matrix R. The spatial filter is implemented by a

function procedure f(). The proposed solution incorporates mixed-mode parallel programming and local variables of processes.

```
FORALL i, j: [1..n] DO
  VAR Oij := O[i,j];              (* save initial original *)
      Pij := 0.0;
  BEGIN
    O[i,j] := f(O[i-1,j], O[i,j-1],        (* apply filter *)
                O[i+1,j], O[i,j+1], O[i,j]);
    NOSYNC
      FOR k := 1 TO n DO              (* calculate product *)
        INC(Pij, O[i,k]*D[k,j]);
      END;
      R[j,i] := Pij;            (* mirror by transposition *)
    END;
    R[i,j] := Oij + R[i,j];         (* add initial original *)
    D[i,j] := D[j,i];       (* transpose distortion matrix *)
  END;
END
```

4 MSIMD Execution of Conditionals

The general idea of "multiple SIMD" = MSIMD execution is to let different branches of conditionals be taken simultaneously by many processes. The processes inside each branch constitute new process groups that execute their respective statement sequences in local synchrony before they finally re-merge with the processes from all other branches at the end of the conditional.

4.1 IF Statements

```
  IF    C1 THEN  S1
[ ELSIF C2 THEN  S2
  ...
  ELSIF Ck THEN  Sk  ]
[ ELSE           S0  ]
  END
```

On encountering the above IF statement, a process group PG first partitions itself into $k+1$ subgroups $PG_0, \ldots PG_k$ using the algorithm given below.

1. Initialize $j := 1$ and $PG_0 := PG$.
2. Let process group PG_0 evaluate condition C_j.
3. $PG_j := \left\{ p \in PG_0 : C_j^{|p} = \text{TRUE} \right\}$
4. If $j < k$ and $PG_0 \neq \emptyset$ then restart at step 2 with $j := j+1$, otherwise terminate.

Thereafter, each subgroup PG_j executes its statement sequence S_j where S_0 equals the empty statement if no ELSE clause is present. All subgroups PG_j operate in a mutually independent manner (preferably in parallel, of course) until they finish execution of their respective statement sequences S_j. Finally, each subgroup waits at the END for all others to finish, too. Thus, they synchronize before they merge and restore PG on leaving the whole IF statement.

Discussion. For $|PG| = 1$ the equivalence of this scheme and standard sequential execution is fairly obvious. For $|PG| > 1$ the scheme demands a step-wise evaluation of all necessary conditions before any statement sequence S_j may be executed. The rationale behind this strict ordering is to exclude any possibilities of counter-intuitive impacts on condition evaluation that could be caused by side effects generated by some statement sequence S_j. Please consider the following simple IF conditional as an illustrative example.

```
IF     X[i]=0  THEN  X[i-1] := 1;
ELSIF  X[i]=1  THEN  q();          END;
```

If the assignment X[i-1]:=1 of the first THEN clause is executed before the remaining processes evaluate the second condition X[i]=1, it could greatly influence the set of processes entering the second THEN clause. Depending on the speed of the different process groups such style of execution might even lead to indeterministic behavior of the whole conditional statement.

In order to argue for the semantic "correctness" of the partitioning algorithm, we now formally discuss some of its features. Beside completeness, i.e. $PG = \bigcup_{j=0}^{k} PG_j$, the algorithm ensures pair-wise disjointness of all resulting subgroups. Furthermore, it puts each $p \in PG$ into PG_{j_p} where $j_p = \min\{ i \in \{1..k\} : C_i^{|p} = \text{TRUE} \}$ if existent and $j_p = 0$ otherwise. Therefore, the following properties that imply semantic "correctness" hold after its termination.

- $\left(i \neq j \ \wedge \ i, j \in \{0..k\} \right) \ \rightarrow \ PG_i \cap PG_j = \emptyset$
- $p \in PG_0 \ \rightarrow \ \forall \ j \in \{1..k\} : C_j^{|p} = \text{TRUE}$
- $\left(p \in PG_j \ \wedge \ j \in \{1..k\} \right) \ \rightarrow \ \left(C_j^{|p} = \text{TRUE} \ \wedge \ \forall \, i \in \{1..j-1\} : C_i^{|p} = \text{FALSE} \right)$

Although we do not consider it a disadvantage, MSIMD execution of conditionals results in an intricate characteristic of IF statement semantics that needs to be mentioned. In contrast to strict, traditional SIMD execution, IF–ELSIF cascades are no longer equivalent to their corresponding IF–ELSE–IF cascades in the general MSIMD case. Only for process groups with at most one active process, e.g. during sequential execution, the equivalence still holds.

4.2 CASE Statements

```
  CASE X OF
    | L1  =>  S1
[   ...
    | Lk  =>  Sk ]
[   ELSE      S0 ]
  END
```

The above CASE statement is semantically equivalent to the following WITH/IF–ELSIF cascade where the new identifier V does not appear in the original statement.

```
  WITH V = X DO
    IF    V = L1 THEN  S1
[   ELSIF V = L2 THEN  S2
    ...
    ELSIF V = Lk THEN  Sk  ]
[   ELSE               S0  ]
    END;
  END
```

5 General Exception Handling by Process Groups

This section presents our notion of unified exception handling in Modula-3*. Firstly, we discuss the fundamental problems and some alternative solutions of the topic. Secondly, we formulate the general framework of our approach. Relying thereon and closely following the original language definition of Modula-3, we then specify exception-based, synchronous semantics for all remaining statements.

5.1 Basic Considerations

When analyzing the interplay between exception handling and process groups in Modula-3*, three fundamental questions arise.

- What happens during the merger of two or more process groups?
- What happens if several distinct processes raise different exceptions?
- What happens when sets of processes are replaced by their original creators?

The first two questions have easy, natural answers. Actually, nothing unusual happens in both two scenarios because each process carries its own specific state information that remains unchanged in case of process group mergings.

The third question, however, proves to be quite tricky. Major difficulties arise from the necessity to not only reduce the number of processes but also the overall exception information. For mixtures of active and differently intercepted processes such reductions are non-trivial — especially when taking parameterized, program-defined exceptions into account. The prototypical example below illustrates the characteristics of these situations.

```
FORALL J: T [NOSYNC] DO
  TRY
    IF C11 THEN RAISE E11(X11); END;
    ...
    IF C1k THEN RAISE E1k(X1k); END;
    IF C01 THEN RAISE E01(X01); END;
    ...
    IF C0m THEN RAISE E0m(X0m); END;
  EXCEPT
    | E01(Y1) =>  H1
    ...
    | E0m(Ym) =>  Hm
  END;
END
```

The exceptions $E_{01}, \ldots E_{0m}$ are handled inside the local handlers $H_{01}, \ldots H_{0m}$ of the `EXCEPT` clause by the same process that raised them. Hence, processes cannot be intercepted by any of these exceptions E_{0j} when reaching the `END` of the `FORALL`.

In contrast thereto, no local handlers exist for the exceptions $E_{11}, \ldots E_{1k}$ such that they remain unhandled inside the `FORALL`. Consequently, arbitrarily many of the created processes may be intercepted by different and/or distinctly parameterized exceptions $E_{11}, \ldots E_{1k}$ when reaching the closing `END` of the whole `FORALL` statement.

At first glance, resorting to vector-valued exceptions might look like an easy solution for the whole reduction problem. Unfortunately, this scheme does not help much because it only postpones the necessary reduction of exception information; the scheme essentially delegates this task to the handlers, i.e. the programmers implementing them.

Another obvious solution for the exception reduction problem consists of special reduction rules supplied by the programmer. Despite its indisputable yet hardly ever needed flexibility, this solution suffers from at least two severe disadvantages. Firstly, it requires further language extensions and, secondly, the number of reduction rules to be supplied grows exponentially with the number of unhandled exceptions raised in parallel contexts. Furthermore, the invisible but nonetheless implicitly predefined exceptions ⟨*exit-exception*⟩ and ⟨*return-exception*⟩ of Modula-3 do not fit well into this explicit reduction framework.

Our solution for the exception reduction problem aims at ease of use and a smooth integration (see Section 5.7) instead of unlimited flexibility. As stated by the rules in Section 5.2, we allow for process group reductions if and only if all intercepted processes of the reduction set carry exactly the same exception information; otherwise, a checked runtime error occurs. This approach does not sacrifice any functionality because programmers can still store and propagate distinct information for each intercepted process, if they absolutely have to. The following example shows how to do this by means of an exception *E* parameterized with a single reference *X* to a freshly allocated, dynamic array with `NUMBER(T)` elements of type *U*.

```
EXCEPTION  E (REF ARRAY OF U);

VAR X := NEW(REF ARRAY OF U, NUMBER(T));
BEGIN
  TRY
    FORALL J: T [NOSYNC] DO
      ...
      IF C THEN
        X[ORD(J)] := ... ;
        RAISE E(X);
      END;
      ...
    END;
  EXCEPT
    | E(Y) =>  ... ;
  END;
END
```

5.2 Exception Propagation

The following rules provide the base of our unifying scheme for sequential and parallel exception handling.

1. If all processes *p* of a process group *PG* are intercepted then *PG* immediately abandons the statement sequence it currently executes by simply skipping the remainder thereof.

2. Process group mergings, e.g. at the ENDs of IF or NOSYNC statements, do not influence the interception states of the participating processes.
3. Process group reductions at the ENDs of FORALL statements invoke reductions of exception information as well. For each process set *PG* being reduced, the runtime system must verify that all its intercepted processes bear the same exception with identical parameter values; verification success signals the existence of meaningful exception reductions for all sets *PG*. In this case, the representative exceptions and their associated parameter values are propagated to the resulting processes of the process group reduction. Otherwise, exception reduction does not make much sense for at least one process set *PG*; we define this situation to generate a checked runtime error.

Remark. We intentionally include the somewhat redundant rule 1 in order to clarify the behavior during execution of conceptually infinite statement sequences (see Section 5.7).

5.3 RAISE Statements

```
RAISE E [(X)]
```

A process group *PG* executes a RAISE statement by first evaluating argument *X*, if present, and then raising exception *E* for each active $p \in PG$, parameterized with $X^{|p}$ if necessary.

Discussion. All processes of *PG* are intercepted after executing the RAISE statement. Hence, *PG* abandons the statement sequence it currently executes immediately after the RAISE according to rule 1 of Section 5.2. Thus, the proposed scheme subsumes the sequential RAISE execution of Modula-3.

5.4 TRY–EXCEPT Statements

```
  TRY
    S
  EXCEPT
[   | E1 [(X1)] =>  H1
    ...
    | Ek [(Xk)] =>  Hk ]
[   ELSE            H0 ]
  END
```

On encountering the above TRY–EXCEPT statement, a process group *PG* first executes statement sequence *S*. Then it partitions itself into two disjoint subgroups, namely $PG_I = \{p \in PG : p \text{ is intercepted now but was active on entrance of } S\}$ and $PG_O = PG \setminus PG_I$. Thereafter, PG_I executes the EXCEPT–ELSE clauses in MSIMD manner like a CASE statement, with all its processes actively participating despite their intercepted states.

Furthermore, each process $p \in PG_I$ clears its exception state and continues normally as soon as it enters an appropriate handler H_j. Meanwhile, PG_O stalls at the END and waits for PG_I to finish. Thus, both subgroups synchronize before they merge and restore *PG* on finally leaving the whole TRY–EXCEPT statement.

Discussion. Exactly those processes execute the EXCEPT–ELSE clauses that enter statement sequence *S* in the active state and are intercepted afterwards. The subgroup PG_I collects all such "freshly" intercepted processes and synchronizes their exception handling activities. Meanwhile, the second subgroup PG_O gathers the remaining processes which must wait at the END for the termination of exception handling.

In the case of sequential process groups *PG* with $|PG| = 1$, either PG_I or PG_O, respectively, will be empty if statement seuqence *S* terminates normally or not. Hence, the above scheme subsumes the standard sequential execution of TRY–EXCEPT statements in Modula-3.

5.5 TRY–FINALLY Statements

```
TRY
  S1
FINALLY
  S2
END
```

The above TRY–FINALLY statement is semantically equivalent to the following nested TRY–EXCEPT cascade with a number of different RAISE clauses inside.

```
TRY
  S1 ;
  RAISE <finally-exception>;
EXCEPT
  | <finally-exception> => S2
  ELSE
    TRY
      S2 ;
      RAISE <finally-exception>;
    EXCEPT
      | <finally-exception> => RAISE <previous-exception>;
    END;
END
```

Here, <*finally-exception*> denotes a uniquely predefined exception raised solely after the execution of statement sequences S_1 and S_2. The conceptual <*previous-exception*> represents the exception potentially raised during execution of statement sequence S_1 for the corresponding process. Thus, the second handler for <*finally-exception*> re-raises the exceptions that processes were intercepted by during execution of statement sequence S_1.

Remark. The above equivalence subsumes and formalizes the traditional, sequential semantics of TRY–FINALLY statements in Modula-3 which are only described verbally in the original language definition.

5.6 Procedure Calls

```
[ EVAL ]  P([ F1 := A1 , ... Fk := Ak ])
```

The above (function) procedure call is semantically equivalent to the following NOSYNC/ TRY–EXCEPT nest wrapped into a block statement.

```
[ VAR
    F1 := A1 ;  ...  Fk := Ak ; ]
  BEGIN
    NOSYNC
      TRY
        <body-of P>;
        RAISE <return-exception>;
      EXCEPT
        | <return-exception>(R) => [ <result-of P> := R; ]
      END;
    END;
  END
```

Here, <*return-exception*> denotes the uniquely predefined exception solely raised by RETURN statements. If *P* is a function procedure the handler for <*return-exception*> assigns the exception argument *R* as the final result of the call. Furthermore, the conceptual <*body-of P*> represents the textual insertion of the statements implementing procedure *P*.

Discussion. This equivalence rule cannot be used as a valid source–to–source transformation if the called procedure employs direct or indirect recursion. The equivalence means that Modula-3* only requires the synchronization of parameter passing in case of synchronous procedure calls; the statements themselves inside called procedures need not be synchronized. This greatly simplifies Modula-3* compilation on real parallel machines because it frees compilers of tracking and resolving global side-effects of procedures called in parallel. Hence,

the outcome of synchronous Modula-3* procedure calls may sometimes differ from completely synchronized ones if they contain a mixture of read and write accesses to the same global data.

5.7 Other Statements

In the cases of all other statements (DEC, EXIT, EVAL, FOR, INC, LOCK, LOOP, REPEAT, RETURN, WHILE, WITH) no changes of the original Modula-3 language definition [3] are required when specifying their parallel semantics. Below we present the exception-based semantics of loops as an illustrative example.

```
LOOP
  S
END
```

The above LOOP statement is semantically equivalent to the TRY–EXCEPT given below where <*exit-exception*> denotes the uniquely predefined exception solely raised by EXIT statements. The three dots symbolize an infinite repetition of statement sequence *S*.

```
TRY
  S; S; ... S;
EXCEPT
  | <exit-exception> =>  (* skip *)
END
```

Remark. This conceptual TRY–EXCEPT construction exactly captures the well-known SIMD semantics of synchronous loops that are, for instance, described in [1, 2, 7, 13, 14]. The repeated execution of loop body *S* proceeds as long as any process of the entering group is still active, i.e. has not yet reached an appropriate EXIT.

5.8 Programming Example

Good examples of explicit exception handling cannot be really short because TRY–EXCEPT and TRY–FINALLY statements tend to be somewhat involved by definition. Despite its space consumption, we therefore choose to present a complete, realistic example that illustrates the capabilities of both sequential and parallel exception handling in Modula-3*.

```
EXCEPTION BadData;
PROCEDURE ComputeData(k: INTEGER) RAISES ANY   =  ... ;
PROCEDURE GlobalDataRepair(k: INTEGER)         =  ... ;
PROCEDURE InputData(VAR k: INTEGER) RAISES ANY =  ... ;
PROCEDURE LocalDataRepair(k: INTEGER)          =  ... ;
```

```
PROCEDURE ProcessData(k: INTEGER) RAISES {BadData} =
BEGIN
  FORALL i: [1..k] NOSYNC DO
    TRY
      ComputeData(i);                    (* 1st local trial *)
    EXCEPT
      | BadData => LocalDataRepair(i);      (* local repair *)
                   TRY
                     ComputeData(i);     (* 2nd local trial *)
                   EXCEPT ELSE
                     (* any exception => data is still bad *)
                     RAISE BadData;
                   END;
      ELSE
        (* prevent the propagation of all other exceptions *)
    END;
  END;
END ProcessData;
```

```
PROCEDURE HandleData() RAISES ANY =
VAR n: INTEGER;
BEGIN
  LOOP
    InputData(n);      (* get the next data to be processed *)
    TRY
      ProcessData(n);                   (* 1st global trial *)
    EXCEPT
      | BadData => GlobalDataRepair(n);    (* global repair *)
                   ProcessData(n);      (* 2nd global trial *)
    END;
  END;
END HandleData;
```


The first framed box of the example provides all global declarations, namely the parameterless exception `BadData` and the signatures of four auxiliary procedures `ComputeData`, `GlobalDataRepair`, `InputData` and `LocalDataRepair`. The two remaining boxes contain full implementations of the procedures `ProcessData` and `HandleData` which employ typical try–repair–retry strategies of fault-tolerant data processing. Both procedure listings show that the advanced exception handling mechanisms of Modula-3* allow for concise formulations of these strategies in terms of code structure, readability, and safety.

The procedure `ProcessData` relies on parallel execution and parallel exception handling implemented by an asynchronous `FORALL` with two nested `TRY-EXCEPT`s inside. Therein, each created process tries to locally resolve the data inconsistencies signalled by the exception `BadData`. Moreover, the `FORALL` processes and the procedure as a whole must spend some effort on ensuring that no other exception but `BadData` gets raised after their termination.

The procedure `HandleData` repeatedly fetches and processes data until an input exception or a globally irresolvable data inconsistency as signalled by the exception `BadData` occurs. In every iteration, `HandleData` calls `ProcessData` by a sequential try–repair–retry scheme.

6 Conclusion

We discussed and presented a unifying specification of operational semantics for sequential and parallel exception handling in Modula-3*. Based on a concept of process groups, these semantics neatly fit into the original language framework of Modula-3 because the definitions of most statements remain unchanged. This clearly assesses the elegance of our approach that seems to have good potential for wide-spread use and adaption into other languages as well.

Furthermore, the concepts of process groups and their interception provide an excellent starting point for the implementation of Modula-3* on real parallel machines. There, explicit "active set" modelling is inevitably needed for the synchronous execution of loops. According to our unifying scheme, this effectively comes for free with parallel exception handling which should thus result in greatly enhanced functionality without much more implementation effort.

Acknowledgements

Paul Lukowicz and Lutz Prechelt reviewed early versions of this work. Niclas Andersson spent a good deal of time searching and collecting components for the PC that I finished this article on while staying at PELAB, Linköping University, Sweden. Last but not least, I owe a thousand thanks to Marianne Hardt who spent a long night helping me to convert the final version of this article from its LaTeX original to a professionally looking MS-Word document.

References

[1] G.S. Almasi and A. Gottlieb. *Highly Parallel Computing (2nd edition).* Benjamin Cummings Publishing Company, Redwood City, USA, 1994.

[2] L. Bougé. On the semantics of languages for massively parallel SIMD architectures. In *Proceedings of the 1991 Conference on Parallel Architectures and Languages Europe*

(Volume II), pages 166–183, Eindhoven, The Netherlands, June 13–16, 1991 (published by Springer Verlag as LNCS 506).

[3] L. Cardelli, J. Donahue, L. Glassman, M. Jordan, B. Kalsow, and G. Nelson. Modula-3 Language Definition. In *ACM SIGPLAN Notices*, 27(8):15–42, August 1992.

[4] N. Gehani and A.D. McGettrick (editors). *Concurrent Programming.* Addison-Wesley Publishing Company, Reading, USA, 1988.

[5] P.B. Gibbons. A more practical PRAM model. In *Proceedings of the 1989 ACM Symposium on Parallel Algorithms and Architectures*, Santa Fe, USA, June 18–21, 1989 (published by ACM Press).

[6] S.U. Hänßgen, E.A. Heinz, P. Lukowicz, M. Philippsen, and W.F. Tichy. The Modula-2* environment for parallel programming. In *Proceedings of the Working Conference on Programming Models for Massively Parallel Computers*, pages 43–52, Berlin, Germany, September 20–23, 1993 (published by IEEE Computer Society Press).

[7] T. Hagerup, A. Schmitt, and H. Seidl. FORK: A high-level language for PRAMs. In *Proceedings of the 1991 Conference on Parallel Architectures and Languages Europe (Volume I)*, pages 304–320, Eindhoven, The Netherlands, June 13–16, 1991 (published by Springer Verlag as LNCS 505).

[8] E.A. Heinz. Modula-3*: An efficiently compilable extension of Modula-3 for problem-oriented explicitly parallel programming. In *Proceedings of the Joint Symposium on Parallel Processing 1993*, pages 269–276, Tokyo, Japan, May 17–19, 1993 (available by ftp from “i41s10.ira.uka.de” [129.13.13.110] in directory “pub/m3s” as “jspp93.ps”).

[9] J. JáJá. *An Introduction to Parallel Algorithms.* Addison-Wesley Publishing Company, Reading, USA, 1992.

[10] B. Kalsow and E. Muller. *SRC Modula-3 Version 2.11.* Systems Research Center, Digital Equipment Corporation, Palo Alto, USA, January 1992.

[11] M. Philippsen, T.M. Warschko, W.F. Tichy, C.G. Herter, E.A. Heinz, and P. Lukowicz. Project Triton: Towards improved programmability of parallel computers. In *D.J. Lilja and P.L. Bird (editors). The Interaction of Compilation Technology and Computer Architecture*, pages 249–281, Kluwer Academic Publishers, Boston, USA, 1994.

[12] S. Prakash, M. Dhagat, and R. Bagrodia. Synchronization issues in data-parallel languages. In *Proceedings of the 6th International Workshop on Languages and Compilers for Parallel Computing*, pages 76–95, Portland, USA, August 1993 (published by Springer Verlag as LNCS 768).

[13] J.R. Smith. *The Design and Analysis of Parallel Algorithms.* Oxford University Press, New York, USA, 1993.

[14] W.F. Tichy and C.G. Herter. Modula-2*: An extension of Modula-2 for highly parallel, portable programs. Technical Report 4/90, Faculty of Informatics, University of Karlsruhe, January 1990 (available by ftp from “i41s10.ira.uka.de” [129.13.13.110] in directory “pub/m2s” as “m2s90.ps”).

[15] S. Yemini and D.M. Berry. A modular verifiable exception-handling mechanism. In *ACM Transactions on Programming Languages and Systems*, 7(2):214–243, April 1985.