INTERNATIONAL COMPUTER SCIENCE INSTITUTE

1947 Center St. • Suite 600 • Berkeley, California 94704-1198 • (510) 643-9153 • FAX (510) 643-7684

Enabling Compiler Transformations for pSather 1.1

Michael Philippsen*
phlipp@icsi.berkeley.edu

TR-95-040

August 1995

Abstract

pSather 1.1 [2] is a parallel extension of the object-oriented sequential programming language Sather 1.1 [1]. A compiler for sequential Sather is available which is written in Sather. This document describes the basic ideas of the extensions of the sequential Sather compiler to handle pSather programs and is thus a high-level documentation of parts of the pSather compiler. Most of the transformations are presented in form of a transformation from pSather to Sather.

*On leave from Department of Computer Science, University of Karlsruhe, Germany

Contents

1 Introduction

Throughout this document we assume that the reader is familiar with the language definitions of sequential Sather 1.1 [1] and the parallel extension pSather 1.1 [2].

Conceptually, the transformation of pSather to Sather is organized in phases, each of which deals with a separate problem of the translation. The document's organization reflects these phases. Note however, that a phase does not correspond with a pass of compilation. Instead the implementation achieves all transformations in a single pass.

Phase I. The first phase of the transformation focuses on pSather's memory consistency model. The memory consistency model of pSather offers a shared address space to the programmer. But it is not a sequentially consistent shared memory model because changes to object attributes are in general not immediately visible to all threads. The language specification associates *import* and *export* operations with various language constructs and operations.

The goal of the first phase of the transformation of pSather's memory consistency model is to make these operations explicit, i.e., instead of associating import and export operations with language constructs and operations, functions of the SYS class are called explicitly.

The transformations of the first phase are described in section 2.

Phase II. In the second phase threads are transformed into routines or functions. This is relevant for all pSather implementations since most available thread packages require a correspondence between threads and functions. Local variables which are visible at the point of thread creation in pSather must be made accessible in the resulting routine. We decided to pass these local variables into the new routine by use of newly created objects, so called helper objects.

The transformations of the second phase are described in sections 3 to 5.

In this part of the document, we assume that the Sather compiler targets a run-time system that offers threads and mechanisms for blocking and synchronization. Hence, we rely on a mechanism for starting routines concurrently that are implemented in sequential Sather.

Phase III. In section 6 we focus on the declaration of helper objects and discuss the access to attributes of these helper objects. The transformation is context sensitive and behaves differently for different nestings of **par**, **fork**, and **attach** statements. The declaration of helper objects, their packing and unpacking, and the update of attributes is described in detail in section 6.

Finally, section 7 discusses a pSather implementation of the producer-consumer problem and presents the result of all transformation phases.

2 Transformation of pSather's Memory Consistency Model

The first phase of the transformation focuses on pSather's memory consistency model. The memory consistency model of pSather offers a shared address space to the programmer. But it is not a sequentially consistent shared memory model because changes to object attributes are in general not immediately visible to all threads. The language specification associates *import* and *export* operations with various language constructs and operations.

The goal of the memory model transformation is to make these operations explicit, i.e., instead of associating import and export operations with language constructs and operations, functions of the SYS class are called explicitly. In the remainder of this section we describe transformation templates that implement pSather's memory consistency model.

Let us first look at the import and export rules as defined in the pSather language specification. An **import** occurs:

Rule	Condition
Imp1	in a newly created thread,
Imp2	on exiting a **par** statement (children have terminated),
Imp3	on entering one of the branches of a **lock** statement,
Imp4	on exiting exclusive GATE and GATE{T} operations, and
Imp5	on completion of a **sync** statement.

An **export** occurs:

Rule	Condition
Exp1	in a parent thread when a child thread is forked,
Exp2	by a thread on termination,
Exp3	on exiting a **lock** statement,
Exp4	on entering an **unlock** statement,
Exp5	on entering exclusive GATE and GATE{T} operations, and
Exp6	on initiation of a **sync** statement.

In the remainder of this section we present transformation templates for all relevant constructs of pSather.

The transformation templates map pSather programs into pSather programs, since SYS::import and SYS::export are legitimate pSather statements. Moreover, because repetitions of one of these calls are semantically equivalent to a single call, the semantics of a given pSather program is not changed by adding these calls explicitly where implicit import or export operations are defined by the language.

For notational purposes, we define a new language pSather$_{ie}$, which is pSather without the association of import and export to constructs and operations. Therefore, if in pSather$_{ie}$ a thread changes an attribute of an object, another thread is only guaranteed to observe this change, if (1) the first thread has executed an explicit export operation, if (2) the second thread has executed an explicit import operation, and if (3) export and import have occurred in this temporal order.

In the code fragments shown below the left hand side shows the given pSather code. The right hand side shows pSather$_{ie}$ code which is the result of the transformation. The explanations of the transformation use the rule identifiers given in the above tables.

2.1 Constructs Dealing with Threads

2.1.1 Memory Model Transformation of *par* Statements

pSather	$\longrightarrow$ *pSather$_{ie}$*
par	1 SYS::export;
stmts_1	2 **par**--*ie*
end	3 SYS::import;
	4 stmts_1;
	5 SYS::export;
	6 **end**;
	7 SYS::import;

The language specification defines that the body of a **par** statement is (conceptually) executed by a new thread. Therefore, rule **Exp1** requires an export immediately before the **par** statement (line 1). Analogously, rule **Imp1** results in an import at the beginning of the body (line 3). Due to rule **Exp2**, the new thread must export changes before termination (line 5). These changes must be imported after the **par** in line 7 because of rule **Imp2**.

Note, that the new thread cannot legally terminate inside of stmts_1 since the use of **return**, **yield**, and **quit** is not allowed inside the body of a **par** statement. Moreover, iters can only be called inside the body of the **par** statement if the enclosing **loop** statement is as well inside the body of the **par** statement. Exceptions are not an issue with respect to thread termination. If the programmer does not catch exceptions with explicit **protect** statements in the body of the **par** (and hence before the termination of the thread) these exceptions are considered to be fatal errors. Since the program is terminated in presence of such exceptions, no export operation must be called.

Other than the mentioned rules **Imp1**, **Imp2**, **Exp1**, and **Exp2**, no memory consistency model rules must be applied when transforming a **par** statement.

2.1.2 Memory Model Transformation of *fork* Statements

pSather	$\longrightarrow$ *pSather$_{ie}$*
fork	SYS::export;
stmts_1	**fork**--*ie*
end	SYS::import;
	stmts_1;
	SYS::export;
	end;

The memory model transformation of the **fork** statement results from the same rules **Imp1**, **Exp1**, and **Exp2** as the memory model transformation of the **par** statement shown above.

The difference is that after the **fork** statement no import is required. Hence, unless caused by other import or export rules that might be hidden in stmts_1, there is no specific point in the program at which the thread that executes the **fork** statement will become aware of changes made by the forked thread. However, since **fork** statements may only occur inside the body of a **par** statement, the closing import caused by the **par** statement makes sure that these changes are seen by the originating thread.

Note again, that the new thread cannot legally terminate inside of stmts_1 since the use of **return**, **yield**, and **quit** is not allowed inside the body of a **fork** statement. Again, iters can only be called inside the body of the **fork** statement if the enclosing **loop** statement is as well inside the body of the **fork** statement. Because of the same reasons given for the **par** statement, exceptions are not an issue with respect to thread termination inside of the body of the **fork**.

2.1.3 Memory Model Transformation of *attach* Statements

Although the right hand side of an **attach** statement can be a complex expression, let us assume for now and without loss of generality that the right hand side is simply a function call.

The given transformation template is incomplete and only suggestive because such a definition of new routines induces additional problems due to local variables. Since this problem will be solved in section 3 we – for now – postpone the presentation of a complete transformation template.

$pSather \longrightarrow pSather_{ie}$, **incomplete**

expr1 :- operation;	SYS::export;	1
	expr1 :- operation'; --*ie*	2
	operation' =	101
	SYS::import;	102
	tmp ::= operation;	103
	SYS::export;	104
	return tmp;	105

The memory model transformation of **attach** statements is slightly more complicated. Although the same rules **Imp1**, **Exp1**, and **Exp2** apply to the **attach** statement as they do to the **fork** statement, the transformation requires more work since pSather allows only expressions on the right hand side of the **attach** statement. Hence, simply adding additional statements is not possible.

To circumvent this problem, we define a new routine operation' that has the necessary import and export calls. Instead of attaching operation to the gate determined by evaluation of expr1, the new routine operation' is attached.

Note, that there is no explicit import operation after the asynchronous call of operation' in line 2. The reason is, that the originating thread proceeds without waiting for the newly created thread. The two threads synchronize, when the originating thread uses an exclusive gate operation to access the result of operation' which is stored in the gate. See section 2.3 for the transformation templates used to implement rules **Exp4** and **Imp4** of pSather's memory consistency model.

2.2 Constructs Dealing with Locks

2.2.1 Memory Model Transformation of *lock* Statements

Rule **Imp3** requires an import when entering a branch of a **lock** statement. This is implemented in lines 3, 7, and 11 of the following (incomplete and only suggestive) transformation template . Rule **Exp3** results in export in lines 5, 9, and 13 at the end of the branches of the **lock** statement.

Other than the mentioned rules **Imp3** and **Exp3**, no memory consistency model rules must be applied when transforming the **lock** statement.

$pSather \longrightarrow pSather_{ie}$, **incomplete**

```
lock
guard <expr> when <lck_list_1> then
  stmts_1
when <lck_list_2> then
  stmts_2
else
  stmts_3
end
```

```
lock
guard <expr> when <lck_list_1> then
   SYS::import;
   stmts_1;
   SYS::export;
when <lck_list_2> then
   SYS::import;
   stmts_2;
   SYS::export;
else
   SYS::import;
   stmts_3;
   SYS::export;
end;
```

However, simply inserting an export call as the last statement of all branches of the **lock** statement is often incorrect. The export must occur whenever the body can be left by the control flow.

This is the case for **return**, **quit** and **raise** statements. (The **yield** statement is not allowed in the **lock** statement.) Moreover, the control flow can leave the **lock** statement if an iter is called inside the **lock** statement but the enclosing **loop** is outside the **lock** statement. To implement this requirement, the statement list stmts_1, stmts_2, and stmts_3 must be processed as shown in the following pseudo code:

Transformation Algorithm append_export*:*

```
append_export(stmt_list,mode) is
   -- process all statements in stmt_list sequentially
   s::=stmt_list.first_stmt;
   loop
      if is_in_loop_and_contains_iter_call(s) then
         insert_export_stmt_before(s);
      end;
      typecase s
      when RETURN_STMT, QUIT_STMT, RAISE_STMT then
         insert_export_stmt_before(s);
      when STMT_WITH_BODIES then
         loop append_export(s.bodies!,on_exit_only) end;
      end;
      until!(void(s.next)); s:=s.next;
   end;
   -- work on last statement of stmt_list
   if mode /= on_exit_only then
      typecase s
      when RETURN_STMT, QUIT_STMT, RAISE_STMT then -- done
      else
         insert_export_after(s);
      end
   end;
end;
```

Algorithm Description. The **loop** in lines 4 to 15 processes the statements in stmt_list. If the current statement (s) contains an iter call we add a (speculative) export in front of this statement. See lines 5 to 7. This export is only required by rule **Exp3** if the control flow really leaves the **lock** statement. However, since we cannot know this during the transformation, we add a potentially unnecessary export. The **typecase** statement in lines 8 to 13 handles all other statements that might result in leaving the **lock** statement. In front of **return**, **quit**, and **raise** statements an export is added (lines 9 and 10). For statements that have bodies of their own, for example the **if** statement, all their bodies are processed recursively. The difference is that their bodies are processed in on_exit_only mode. In this mode, only the transformation described so far is applied to the statement list.

For the top level statement list of the **lock** statement however, append_export is called in with_final_export mode. This mode makes sure, that at least one export is called at the end of the branch of the **lock** statement. See lines 17 to 23. The final export is only added if the last statement of the list does not cause an export itself. Moreover, the semantics of sequential Sather prohibits the existence of statements after **return** and **raise**.

Implementation Restriction.

> Although pSather requires that the export occurs immediately upon leaving the **lock** statement, the statement based transformation might cause some problems. Imagine a **return** statement that returns a value. However, let this value be provided by a function call that itself changes some global state. The transformation presented above will not export these changes because the export is called before the return statement. This could be corrected by evaluating the expression of the **return** statement into a newly introduced temporary variable first, i.e. before the export operation. The return statement then would return the new temporary variable. Similar effects might occur in statements that accomplish work before calling an iter. Again, since the export is already done the current implementation will not export the accomplished work. Similar to the return statement, the transformation could easily be extended to cover this case correctly.

Now we are equipped to present the complete transformation template for the **lock** statement. In line 4 the statement list stmts_1 is transformed by the algorithm we have just described. The recursive algorithm is called in with_final_export mode to make sure that there is at least one export after the statements in stmts_1 unless the type of last statement prohibits this. The same transformation algorithm is called for stmts_2 (in line 7) and stmts_3 (in line 10).

$$pSather \longrightarrow pSather_{ie}$$

lock	**lock**	1
guard <expr> **when** <lck_list_1> **then**	guide <expr> **when** <lck_list_1> **then**	2
stmts_1	SYS::import;	3
when <lck_list_2> **then**	append_export(stmts_1,with_final_export);	4
stmts_2	**when** <lck_list_2> **then**	5
else	SYS::import;	6
stmts_3	append_export(stmts_2,with_final_export);	7
end	**else**	8
	SYS::import;	9
	append_export(stmts_3,with_final_export);	10
	end;	11

2.2.2 Memory Model Transformation of *unlock* Statements

$$pSather \longrightarrow pSather_{ie}$$

unlock <lck>;	SYS::export;
	unlock <lck>;

Only rule **Exp4** is applicable to the **unlock** statement. Before executing an **unlock** statement, an export must occur.

2.2.3 Memory Model Transformation of *sync* Statements

$$pSather \longrightarrow pSather_{ie}$$

sync;	SYS::export;
	sync;
	SYS::import;

Rule **Imp5** causes an import in front of the **sync** statement. Similarly, rule **Exp6** is applicable and causes the export operation immediately after the **sync** operation in pSather$_{ie}$.

2.3 Exclusive Gate Operations

Rules **Imp4** and **Exp5** require that exclusive gate operations are surrounded by export and import operations. Exclusive gate operations are operations that work on the queue of a gate (set, get, enqueue, and dequeue).

To implement the requirement that exclusive gate operations are *immediately* surrounded by export and import, expressions must be broken up into sequences of individual statements with temporary variables, isolating the gate operation. After the isolation of exclusive gate operations in individual statements, the statement containing the gate operation can easily be immediately surrounded by export and import.

Implementation Restriction.

> Although pSather requires that these operations are immediately surrounded by export and import, the implemented transformation is slightly weaker. Instead of immediately surrounding exclusive gate operations, we surround the statement that contains the exclusive gate operation.
>
> The difference only becomes visible if a program relies on the order of execution of expressions. For example, in cases like g.get + x, where x is changed by a different thread and the import caused by get is essential for the correct behavior of the program.
>
> This implementation restriction requires a special treatment of the **if** statement which is presented in section 2.3.2.

2.3.1 Memory Model Transformation of Exclusive Gate Operations

$$pSather \longrightarrow pSather_{ie}$$

stmt_with_exclusive_gate_op;	SYS::export;
	stmt_with_exclusive_gate_op;
	SYS::import;

Rule **Exp5** requires that before the statement with the exclusive gate operation an export is added to the code. Rule **Imp4** results in an import after that statement.

Similar to the special treatment of the **return** and **raise** statement in the transformation algorithm append_export, the import is not necessary if the statement with the exclusive gate operation is a **return** or **raise** statement. Moreover, the semantics of sequential Sather in that case prohibit the added call of SYS::import. The specific transformation templates are given below:

$pSather$	$\longrightarrow pSather_{ie}$
return expr_with_exclusive_gate_op;	SYS::export; **return** expr_with_exclusive_gate_op;
raise expr_with_exclusive_gate_op;	SYS::export; **raise** expr_with_exclusive_gate_op;

2.3.2 Exclusive Gate Operations in *if* Statements

Without isolation of the exclusive gate operation in a separate statement, the **if** statement needs a special treatment[1]. Otherwise the effects of the evaluation of a condition with an exclusive gate operation might be invisible in the branches of the **if** statement.

$pSather \longrightarrow pSather_{ie}$

```
pSather                                  pSather_ie
if cond_with_exclusive_gate_op then      1  SYS::export;
   stmts_1;                              2  if cond_with_exclusive_gate_op then
end;                                     3     SYS::import;
                                         4     stmts_1;
                                         5  end;
                                         6  SYS::import;
```

Export rule **Exp5** results in the leading export in line 1. The import operation required by rule **Imp4** results in the two imports. An import occurs before the statements of the **then** part (line 3). If there is no **else** part, then an additional import must occur after the **if** statement (line 6).

$pSather \longrightarrow pSather_{ie}$

```
pSather                                  pSather_ie
if cond_with_exclusive_gate_op then      1  SYS::export;
   stmts_1;                              2  if cond_with_exclusive_gate_op then
else                                     3     SYS::import;
   stmts_2                               4     stmts_1;
end;                                     5  else
                                         6     SYS::import;
                                         7     stmts_2
                                         8  end;
```

If there is an **else** part then the second import occurs inside the **else** part (line 6) instead of after the **if** statement.

[1] Since the **case** statement is syntactic sugar based on the **if** statement, we do not consider the **case** statement in more detail.

Transformation of pSather's Threads

Threads can be created in pSather in three ways. One way is the **attach** statement, the second way is the **par** statement, and the third way is the **fork** statement. The basic idea of a transformation used in the pSather compiler is to replace these statements with sequential Sather statements: the **attach** statement, the **par** statement, and the **fork** statement are replaced with routines. This is necessary for most thread packages that can be used to implement a run-time system since only routines can be associated with run-time system threads in these packages.

The target language of this second step of transformation is called pSather$_{ie+thread}$. This language is pSather$_{ie}$ without **par**, **fork**, and **attach** statements and without **cohort**. Instead of these statements the language offers a macro THREAD with four arguments. The first argument is the name of a function to be called concurrently. The second argument is the helper object as defined below. The third argument denotes the gate to which the new thread is attached. Finally, the fourth argument of the macro is the cluster number which should be used to execute the new thread.

3 Transformation of *attach* Statements

3.1 Basic Transformation Principle

When presenting the pSather-to-pSather$_{ie}$ transformation for **attach** statements, we have already mentioned that a new routine is created. Here we complete the transformation template.

The following fragment shows pSather code. When routine r is executed, the running thread spawns a new thread in line 7 that executes the expression operation concurrently. On a parallel machine, operation is supposed to be evaluated on cluster p. The result of this evaluation is enqueued into the gate object resulting from the evaluation of expr1. The expression operation can be quite complex since it can contain routine calls and the use of local variables declared inside the routine r. The local variable local which is declared in line 3 is an example.

Original pSather Code:

```
1  class X is
2     r is
3        local : TYPE_OF_LOCAL := val;
4        --
5        -- some code
6        --
7        expr1 :- operation @ p;
8        --
9        -- some code (2)
10       --
11    end; -- of r
12    -- more class elements
13 end; -- of class X
```

Moving the evaluation of operation into a new routine as required by the transformation to pSather$_{ie}$ requires that all the objects and variables which are visible at the point of the **attach** statement are passed into the new routine.

The transformation presented here achieves the visibility of locals in the new routine by use of *helper objects*. The **attach** statement is replaced by several statements, see lines 28 to 33.

First, the left hand side of the **attach** statement is transformed. In line 28, the expression expr1 is evaluated into a new gate object g of type GATE{T} whereby T is the resulting type of operation. In general, this is not necessary, if expr1 already is a gate. However, since expr1 could be a complex expression which might block during its evaluation, we choose to evaluate expr1 first, before we continue to transform the **attach** statement.

After creation of the gate, a new helper object is created in line 29, then all visible local variables declared inside the routine r (and all parameters of r if there were any) are copied into this helper object (lines 30 to 31). The HELPER class is specific to the transformed **attach** statement. Transformation of other **attach** statements in general result in additional (and often different) helper classes and objects. The copy operations are called *packing* of the helper object. Due to the transformation to pSather$_{ie}$, an explicit export operation is added in line 32.

Finally, THREAD is a macro that starts a new thread. The new thread concurrently executes the function fct(helper,g) at cluster p. In pSather semantics, the new thread is considered to be attached to gate g. If the @-expression and the cluster number are missing, the thread is supposed to run on the same cluster as the calling thread.

pSather$_{ie+thread}$ Code after Transformation:

```
21 class X is
22    r is
23       local : TYPE_OF_LOCAL := val;
24       --
25       -- some code
26       --
27       --
28       g : GATE{T} := expr1;
29       helper ::= #HELPER;
30       helper.local := local;
31       -- ...
32       SYS::export;
33       THREAD(fct,helper,g,p); -- concurrently executable
34       --
35       -- some code (2)
36       --
37    end; -- of r
38    -- more class elements
39    private fct(helper:HELPER, g:GATE{T}) is
40       local : TYPE_OF_LOCAL;
41       SYS::import;
42       local ::= helper.local;
43       tmp ::= operation;
44       g.enqueue(tmp);
45       -- maybe: helper.local := local;
46       SYS::export;
47    end; -- of fct
48 end; -- of class X
```

In the routine fct (lines 39ff) the helper object is *unpacked* after an initial import operation, i.e., first local variables are declared that mirror the local variables which have been visible at the point of the original **attach** statement (line 40). Afterwards the helper object is unpacked, i.e., the newly declared variables are filled (line 42). Then the expression operation is evaluated (line 43), the resulting value is enqueued into the gate g in line 44. Before the final export operation in line 46 which is required by the transformation to pSather$_{ie}$, the affected local variables are copied back into the helper object. This is required in pSather 1.1 because of inout parameters. If operation would have had local as an inout argument, then the value of local could have changed. Rules for packing and unpacking of helper objects are discussed in more detail in section 3.2.

Note that the enqueue operation itself requires to be enclosed between export and import operations (see 2.3.1). Both operations however, can be omitted due to redundancy.

For the helper object a class must be defined that has an attribute for each local variable to be passed into the newly declared function.

```
class HELPER is
   attr local:TYPE_OF_LOCAL;
   -- ...
   create : SAME is
      return new;
   end; -- of create
end; -- of HELPER
```

The transformation of the right hand side expression of the **attach** relies on the fact that the pSather specification does not allow iters to be called in the right hand side expression. Otherwise the transformation would result in an iter which is called in a routine without being textually enclosed in a **loop** statement.

Rationale for helper objects: The use of helper objects and the packing and unpacking of values seems to introduce more complexity than necessary. An alternative transformation could pass the locals as arguments into the new routine. However, there are several reasons for the introduction of helper objects. The first reason is the intended simplicity of the run-time system. Passing all locals via routine arguments would require that the thread creation mechanism of the run-time system could deal with an arbitrary number and worse, arbitrary type of arguments. By always passing a fixed number of arguments, i.e. routine name, helper object, gate, and cluster number, the interface is much simpler. A reason for unpacking is the intended simplicity of the transformation process: Instead of processing the original expression operation and replacing all accesses to local variables with accesses of attributes of the helper object, the expression operation can be copied textually into the body of the new routine. Another reason is efficiency: moving the whole helper object to the cluster that hosts the new thread and then working with local variables is in general much faster than always going through an additional level of indirection. The fourth reason, however, is orthogonality. Similar helper objects will be used in the transformation of both **par** and **fork** statements, where argument passing is insufficient as will be shown in sections 4 and 5.

Optimization: By data flow analysis the export of local variables could easily be optimized: Instead of passing all visible local variables and parameters through the helper object into the thread that implements the right hand side of the **attach** statement, only those must be copied that are *used* in the right hand side expression. However, for simplicity of both the presentation and the transformation we pass all local variables here.

3.2 Helper Objects in pSather's Memory Consistency Model

As discussed in section 2 the memory consistency model of pSather requires that threads which use variables that are shared between threads import and export changes to these variables at certain points of the code. Every transformation must ensure a correct implementation of this memory consistency model.

By introducing explicit import and export operations into the code, the implementation of these routines must guarantee that changes to any objects will be observed correctly. The SYS routines however cannot guarantee correct import and export behavior between local variables and helper objects.

Therefore, the transformation from pSather$_{ie}$ to pSather$_{ie+thread}$ works in two steps, both of which have been applied above. In the first step, helper objects are created and threads are replaced by routines. However, the helper objects are neither packed nor unpacked.

In the second step, import and export operations which are either present in the original pSather program or might have resulted from the transformation to pSather$_{ie}$ are expanded:

Rule	Transformation
Hlp1	Whenever an explicit import operation is encountered, immediately *after* this import operation the helper object is unpacked, i.e., the local variables are set according to the values of the helper object.
Hlp2	Whenever a routine is called which has an import operation in its transitive closure of calls, immediately *after* this routine call the helper object is unpacked.
Hlp3	Whenever a local variable is changed (either by an assignment to it or by using it as an inout argument) and this variable is also available in a helper object, immediately *after* this change the corresponding attribute of the helper object is updated.
Hlp4	If both **Hlp2** and **Hlp3** must be applied after a routine call, rule **Hlp3** must be obeyed first.

In section 6 we will present in more detail what it means to pack/unpack/update helper objects in the general case, i.e., for arbitrary nestings of **par**, **fork**, and **attach** statements.

Implementation Restrictions:

> The same restrictions as for the requirement of immediately surrounding of exclusive gate operations by export and import apply here. The expansion is implemented on a per statement basis. For **if** (and hence **case**) statements the transformation is similar to the one shown in 2.3.2.

Optimization: By data flow analysis the update of local variables due to **Hlp3** could easily be optimized: Instead of updating a helper attribute immediately after the corresponding local variable has been changed, only the last of these changes preceding an export operation must be made visible in the helper object. Standard optimizations, e.g., loop invariant code motion could be used to improve run-time performance for locals that are written inside of a loop. However, for simplicity of both the presentation and the transformation we export all variables here.

4 Transformation of *par* Statements

The basic idea of the transformation of pSather$_{ie}$'s **par** statement is its reduction to the **attach** statement. When a **par** statement is encountered, the semantics of pSather enforce that a new gate is created which is subsequently referred to as **cohort**. Then conceptually a new thread is started and attached to this gate which executes the body of the **par** statement. The original thread blocks and continues after the new thread has terminated.

pSather$_{ie}$ Code:

```
class X is
   r is
      local : TYPE_OF_LOCAL := val;
     --
      -- some code
      --
      SYS::export;
      par--ie
         SYS::import;
         --
         some code (2)
         --
         SYS::export;
      end;
      SYS::import;
      --
      -- some code (3)
      --
   end; -- of r
   -- more class elements
end; -- of class X
```

The following code shows the (still incomplete) result of the transformation which is very similar to the one applied to the **attach** statement. In line 35 a new gate is created. All accesses to **cohort** inside the body of the **par** statement are replaced by accesses to this new gate. Similar to the right hand side of the **attach** statement, the body of the **par** statement is moved into a newly created routine fct (lines 47 to 55). Again, parameters (if any) and visible variables which are declared locally inside the surrounding routine r are passed into the new routine by means of the helper object. Packing and unpacking of the helper object is not shown in detail.

pSather$_{ie+thread}$ Code after Transformation, incomplete:

```
22 class HELPER is
23    attr local:TYPE_OF_LOCAL;
24    -- ...
25    create : SAME is
26       return new;
27    end; -- of create
28 end; -- of HELPER
29 class X is
30    r is
31       local : TYPE_OF_LOCAL := val;
32       --
```

```
33     -- some code
34     --
35     new_cohort ::= #GATE;
36     helper ::= #HELPER;
37     -- pack helper (see section 6)
38     SYS::export;
39     THREAD(fct,helper,new_cohort,any); -- concurrently executable
40     lock when new_cohort.no_threads then end;
41     SYS::import;
42     -- unpack helper (see section 6)
43     --
44     -- some code (3)
45     --
46   end; -- of r
47   private fct(helper:HELPER, new_cohort:GATE) is
48     local : TYPE_OF_LOCAL;
49     SYS::import;
50     -- unpack helper (see section 6)
51     --
52     -- some code (2), update helper object attributes
53     --
54     SYS::export;
55   end; -- of fct
56   -- more class elements
57 end; -- of class X
```

The **lock** statement in line 40 ensures that the original thread can only proceed when no more threads are attached to the new gate, i.e., if the thread that executes the body of the **par** statement has terminated. Later we will see that the same gate is used to attach threads that implement the bodies of **fork** statements. Therefore, all these threads have terminated as well, when the original thread succeeds in acquiring the lock. Note, that no additional import and export need to be introduced into the empty body of this **lock** statement.

Return values of any kind are not an issue. The definition of pSather does not allow any of the following statements to appear inside the body of the **par** statement: **return**, **yield**, **quit**.[2]

[2] The transformation could easily be extended to correctly handle **return** statements as well. For this purpose, a **return** statement in the body of the **par** must transport the return value (if any) to the original thread through the helper object. After the original thread succeeds in acquiring the lock it must check whether a **return** statement has been encountered – the helper object must provide a flag for this purpose – and then return this value.

5 Transformation of *fork* Statements

The basic idea of the transformation of pSather$_{ie}$'s **fork** statement is very similar to the transformation applied to **par** statements. The semantics of pSather only allow **fork** statements to appear in the body of **par** statements, and hence in the bodies of routines implementing **par** statements according to the transformation presented in section 4. Therefore, we face the following situation during transformation.

pSather$_{ie+thread}$ Code after Transformation of **par** *(only):*

```
1  class X is
2    -- more class elements
3    private fct(helper:HELPER, new_cohort:GATE) is -- transformed par statement
4      local : TYPE_OF_LOCAL;
5      SYS::import;
6      -- unpack helper
7      --
8      my_local : TYPE_OF_LOCAL;
9      --
10     -- some code
11     --
12     SYS::export;
13     fork @ p; --ie
14       SYS::import;
15       --
16       -- some code (2)
17       --
18       SYS::export;
19     end;
20     --
21     -- some code (3), update helper object attributes
22     --
23     SYS::export;
24   end; -- of fct
25 end; -- of class X
```

Before we discuss the result of the transformation, let us briefly recall the semantics of pSather. The local variable my_local which is declared (line 8) inside the body of the original **par** statement is *not shared* among all threads. When a new thread is started in line 13 to execute the body of the **fork** statement, this new thread receives a unique copy of my_local. Any changes that this new thread makes to his instance of my_local are not exported and are thus never visible in other threads. Hence packing and unpacking of helper objects must be sensitive to the context in which they occur. The context sensitivity is even more complicated because the semantics allow nesting of **fork** statements. We will discuss the transformation of arbitrary nestings of **attach**, **par**, and **fork** statements and the proper generation of packing, unpacking, and update statements in section 6.

The following three code sections show the result of the transformation which is very similar to the one applied to the **par** statement. The first section (lines 26 to 33) shows the code for the new helper object, the second code fragment (lines 34 to 56) shows the result of the transformation of the **fork** statement. Finally, the third code fragment (lines 57 to 69) illustrates the result of the transformation of the body of the **fork** statement.

Resulting $pSather_{ie+thread}$ *Code after Transformation of* **par** *and* **fork***: (part 1)*

```
26 class HELPER_2 is
27   attr helper:HELPER;
28   attr my_local:TYPE_OF_LOCAL;
29   -- ...
30   create : SAME is
31     return new;
32   end; -- of create
33 end; -- of HELPER
```

As usual, we declare a new helper object upon thread creation. This time, however, the original helper object helper is an attribute of the newly created helper object helper_2, see line 27. When helper objects are packed and unpacked, this nesting must be taken into account. See section 6 for details of nesting. Note, that an access to helper_2.helper.local reaches the same storage position as helper.local does, since all helper objects are reference objects.

Resulting $pSather_{ie+thread}$ *Code after Transformation of* **par** *and* **fork***: (part 2)*

```
34 class X is
35   -- more class elements
36   private fct(helper:HELPER, new_cohort:GATE) is
37     local : TYPE_OF_LOCAL;
38     SYS::import;
39     -- unpack helper (see section 6)
40     --
41     my_local : TYPE_OF_LOCAL;
42     --
43     -- some code
44     --
45     helper_2 ::= #HELPER_2;
46     -- pack helper_2: (see section 6)
47     helper_2.helper := helper
48     helper_2.my_local := my_local;
49     -- ...
50     SYS::export;
51     THREAD(fct_2,helper_2,new_cohort,p); -- concurrently executable
52     --
53     -- some code (3), update helper object attributes
54     --
55     SYS::export;
56   end; -- of fct
```

The transformation moves the body of the **fork** statement to a new routine. Instead of the original **fork** statement, the THREAD macro is issued in line 51. Since the original thread that executes the **fork** statement is attached to the gate **new_cohort** which has been passed as parameter, the new thread that implements the body of the **fork** will be attached to the same gate.

Resulting $pSather_{ie+thread}$ *Code after Transformation of* **par** *and* **fork***: (part 3)*

```
57   private fct_2(helper_2:HELPER_2, new_cohort:GATE) is
58     local : TYPE_OF_LOCAL;
59     my_local : TYPE_OF_LOCAL;
```

```
60        SYS::import;
61        -- unpack helper_2: (see section 6)
62        local := helper_2.helper.local;
63        my_local := helper_2.my_local;
64        --
65        -- some code (2), update helper (!) object attributes
66        --
67        SYS::export;
68    end;
69 end; -- of class X
```

The routine fct_2 that implements the body of the **fork** statement is very similar to the one that implemented the **par** statement before. The only difference appears in the update statements in lines 64 to 66. Whereas in the **par** routine *all* changed local variables are updated in the helper object, in the **fork** routine only those changed local variables are updated, that are inherited from the helper object of the surrounding **par**. (If the transformation of the **par** and the **fork** statement were equivalent, then in line 65 the code should read ...*update helper_2 object attributes*. Note the difference.) In particular, the local variable my_local is not updated.

6 Nesting of *attach*, *par*, and *fork* Statements

Several times during the transformation of the **attach**, **par**, and **fork** statement, we encountered the necessity to pack or unpack helper objects and to update some of their attributes. This section explains how packing, unpacking, and update work for arbitrary nestings of these statements and therefore for arbitrary nesting of helper objects.

6.1 Declaration of Helper Objects

Whenever a new helper object is created, the transformation must determine which attributes the new helper object should have. After the attributes of the helper object are collected, the declaration of local variables at the beginning of the routines that result from the transformation of **par**, **fork**, and **attach** statements are the easy part: for each attribute in the helper object a local variable is created.

6.1.1 Motivating Examples

Nesting Level 1.
The following code fragment shows a top level **par** statement on the left hand side and the corresponding helper object on the right hand side. (The top level **attach** statement has a similar helper object.)

```
a:TYPE_OF_LOCAL;                  class HELPER_1 is
par                                  attr a:TYPE_OF_LOCAL;
end;                                 create is ... end;
                                  end;
```

Nesting Level 2.
The situation for a second level **par** or **fork** statement is slightly more complicated, since the helper object of the surrounding **par** statement must be used to access shared variables. (The second level **attach** statement has a similar helper object.)

```
a:TYPE_OF_LOCAL;                  class HELPER_2 is
par                                  attr helper_1:HELPER_1;
  b:TYPE_OF_LOCAL;                   attr b:TYPE_OF_LOCAL;
  par/fork                           create is ... end;
  end;                            end;
end;                              --
```

The reason for nested helper objects might need more explanation. Aside from the original thread the variable a in the above example is shared by two threads. One thread executes the body of the top level **par** statement. The other thread executes the body of the second level **par** or **fork**. The idea is to have only one single copy of a in the helper objects. Therefore, if the second thread changes helper_2.helper_1.a, this change is immediately visible in the first thread. There are two reasons for the desire to have only one copy. The first reason is a performance reason: without nesting of helper objects the second thread would need to update two copies of a: one copy in helper_1 and another copy in helper_2. The second reason is even stronger: Assume that helper_2 is passed to a third level **par** or **fork**. If now the outermost thread changes the value of a the thread can update this in helper_1.a. However, the thread cannot update its change in any other helper object because these are not declared in his scope. Hence, when the innermost thread tries to import the current value of a, this thread cannot know which of the two versions of a is up-to-date. Hence, using nested helper objects is a prerequisite of easy import. The alternative would require a complex protocol for replication consistency handling.

Nesting Level 3.

Beginning at the third level of **par** or **fork** statements, the structure of the helper object becomes sensitive to the context. We first present the situation of a third level **par** or **fork** statement inside a second level **par** statement. (The third level **attach** statement inside a second level **par** statement has a similar helper object.) The next example will present the second level **fork** statement.

```
a:TYPE_OF_LOCAL;
par
  b:TYPE_OF_LOCAL;
  par
    c:TYPE_OF_LOCAL;
    par/fork
    end
  end;
end;
```

```
class HELPER_3 is
  attr helper_2:HELPER_2;
  attr c:TYPE_OF_LOCAL;
  create is ... end;
end;
--
--
--
--
```

Note that both variables a and b are shared by all threads. Therefore, they are accessible through helper_2 in the new helper object HELPER_3.

The third level **par** or **fork** statement inside a second level **fork** statement requires a different helper object. (The third level **attach** statement inside a second level **fork** statement has a similar helper object.)

```
a:TYPE_OF_LOCAL;
par
  b:TYPE_OF_LOCAL;
  fork
    c:TYPE_OF_LOCAL;
    par/fork
    end
  end;
end;
```

```
class HELPER_3b is
  attr helper_1:HELPER_1;
  attr b:TYPE_OF_LOCAL;
  attr c:TYPE_OF_LOCAL;
  create is ... end;
end;
--
--
--
```

Here only the variable a is shared by the threads. Therefore, an individual copy must be passed into the innermost **par** or **fork** statement. To understand this demand more clearly assume a situation where the forked thread on level 2 has changed its copy of b. Clearly this new value must be made visible inside of the third level. Hence, the copy of b which is available from HELPER_2 in general holds a wrong value, namely the unchanged version of b seen by the thread that executes the body of the top level **par** statement.

6.1.2 Pseudo Code

The structure of the helper objects, i.e., the attributes of a newly created helper object are determined by the following pseudo code:

Transformation Algorithm make_attributes_of_helper*:*

```
1  make_attributes_of_helper is
2     -- 1) link surrounding par helper
3     if we_are_in_a_par_or_fork then
4        surrounding_par_helper := par_helper_of(current_helper);
5        add_attribute_to_helper(surrounding_par_helper)
6     end
7     -- 2) work on visible local variables and parameters
8     loop local::=active_locals_plus_params!;
9        if user_defined(local) then
10          if we_are_in_a_par_or_fork
11          and reached_via_helpers(local,surrounding_par_helper)
12          then -- skip this variable
13          else add_attribute_to_helper(local)
14          end;
15       end;
16    end
17    -- 3) pass cohort into helper object?
18    if we_are_working_on_an_attach and we_are_in_a_par_or_fork then
19       add_attribute_to_helper(current_cohort)
20    end;
```

Algorithm Description. The first step (line 2 to 6) is skipped for top level **par** and **attach** statements. For **attach**, **fork**, and **par** statements inside of **par** and **fork** statements the helper object of that surrounding statement is considered. If the surrounding level is a **par** level then the corresponding helper object becomes an attribute of the new helper. If the surrounding level is a **fork** level, then there must be a **par** level surrounding this **fork**. Hence, the current helper object of the **fork** has an attribute caused by the surrounding **par** level. This attribute is linked into the new helper object. In the implementation, the routine call par_helper_of(current_helper) finds the appropriate helper object that must be linked in.

In the second step (lines 7 to 16) all parameters and visible locally declared variables are considered. If one of those variables is the result of an earlier transformation step, it is ignored. The condition user_defined(local) in line 9 implements this. For all remaining variables a new attribute is added to the helper object, except for those variables for which the condition in lines 10 and 11 holds. If the transformation happens to be inside of a **par** or a **fork** statement, then several variables are inherited (transitively) by the helper of the surrounding **par** statement. This helper object is found in step 1 (surrounding_par_helper, line 4). Note that the shortcut semantics for the evaluation of boolean expressions makes sure that the surrounding_par_helper is not used if the transformation is applied to a top level statement.

To understand the requirement of the transitive inheritance of locals consider the above example again which introduced HELPER_3. The helper object of the surrounding **par** statement (helper_2) contains only the local variable b. However, by working the levels up, the local variable a can be accessed as well: helper_2.helper_1.a. Therefore, the new helper class HELPER_3 only has the attribute c in its body.

The same condition in lines 10 and 11 makes sure that HELPER_3b has both b and c in its body, because b cannot be transitively reached via the surrounding helper object which is helper_1 here.

The third step of the pseudo code (lines 17 to 20) needs some discussion. In general helper objects transport only variables defined by the pSather programmer. Beyond that, local variables which are introduced during transformation are not used in the routine that receives the helper objet. However there is one exception. Inside of a **par** statement, the pSather programmer can freely use **cohort**. In particular, **cohort** can be used in the right hand side expression of an attach statement. Because the right hand side is moved into a newly created routine as has been presented in section 3, the name of the gate that implements **cohort** must be passed inside the **attach** routine. This is not necessary for the transformation of **par** and **fork** statements because the new threads are attached to **cohort** anyhow, i.e., the gate that implements **cohort** is accessible as third parameter of the THREAD macro.

6.2 Packing of Helper Objects, Export

After a new helper object is created during the transformation, the current values of the local variables and parameters are copied into the corresponding attributes of the helper objects. Since obviously all attributes must be filled initially, the pseudo code for pack_helper_object is very similar to the one of make_attributes_of_helper given in section 6.1.2.

Transformation Algorithm pack_helper_object*:*

```
1  pack_helper_object(helper) is
2     -- 1) link surrounding par helper
3     if we_are_in_a_par_or_fork then
4        surrounding_par_helper := par_helper_of(current_helper);
5        update_attribute_in_helper(local,helper);
6     end;
7     -- 2) work on local variables and parameters
8     loop local::=active_locals_plus_params!;
9        if user_defined(local) then
10          if we_are_in_a_par_or_fork
11          and reach_via_helpers(local,surrounding_par_helper)
12          then -- skip this variable
13          else update_attribute_in_helper(local,helper);
14       end;
15    end;
16    -- 3) pack cohort into helper object?
17    if we_are_working_on_an_attach and we_are_in_a_par_or_fork then
18       update_attribute_in_helper(local,helper)
19    end;
```

The only differences occur in lines 5, 13, and 18. Instead of making a new attribute in the class definition of the helper object, an assignment statement is created. The first step creates an assignment statement that looks like:

helper_2.helper_1 := helper_1;

This links the helper object of the surrounding **par** statement into the currently packed new helper object. The second step creates assignment statements for the local variables:

helper_2.local := local;

Finally, in the third step that is only executed when an **attach** statement inside of a **par** statement is transformed, a reference to the current **cohort** of that **par** statement is copied into the new helper object:

helper_2.**new_cohort** := **new_cohort**;

6.3 Update of Helper Object Attributes, Export

After a local variable is changed that is mirrored in a helper object the corresponding attribute of the helper object must be updated. See rule **Hlp3**: Whenever a local variable is changed (either by an assignment to it or by using it as an inout argument) and this variable is also available in a helper object, immediately *after* this change the corresponding attribute of the helper object is updated.

6.3.1 Motivating Examples

Nesting Level 1.
The following code fragment shows two top level **par** statements on the left hand side and the corresponding helper objects on the right hand side. (Top level **attach** statements have a similar helper objects.)

```
a:TYPE_OF_LOCAL;
par
end;
b:TYPE_OF_LOCAL;
par
end;
```

```
class HELPER_1a is
   attr a:TYPE_OF_LOCAL;
   create is ... end;
end;
class HELPER_1b is
   attr a:TYPE_OF_LOCAL;
   attr b:TYPE_OF_LOCAL;
   create is ... end;
end;
```

Inside of the first **par** statement, the helper object helper_1 must be used. In the second **par** the helper object helper_2 is used. After the end of the first **par** statement, the first helper object is no longer of any interest since all threads that might use this helper object have terminated. Note that at all points of the program the last declared helper object is the one that gets used. Here the last declared helper object always is a helper object caused by a **par** statement.

Nesting Level 2.
The situation for a second level **par** or **fork** statement is slightly more complicated, since the helper object of the surrounding **par** statement must be used for access to shared variables. (The second level **attach** statement has a similar helper object.) The code on the left hand side has a sequence of **par** and **fork** statements in its body.

```
a:TYPE_OF_LOCAL;
par
  b:TYPE_OF_LOCAL;
  par    -- uses helper_2a
  end;
  c:TYPE_OF_LOCAL;
  fork   -- uses helper_2b
  end;
end;
```

```
class HELPER_2a is
  attr helper_1:HELPER_1;
  attr b:TYPE_OF_LOCAL;
  create is ... end;
end;
class HELPER_2b is
  attr helper_1:HELPER_1;
  attr b:TYPE_OF_LOCAL;
  attr c:TYPE_OF_LOCAL;
  create is ... end;
end;
```

The interesting aspect here is that the thread that executes the top level **par** statement can use any of the helper objects when updating the value of a. Independent of the choice, the update will always reach helper_1.a. When updating the value of b however, this thread must be more careful.

Since b is shared with the thread that executes the second level **par** statement, the helper object helper_2a must be used. Moreover, helper_2b must not be used to update the value of b, because the thread that executes the **fork** statement initially gets a copy of b and is intended not to see any changes that are made to the original b. Hence, for a correct update of helper object elements, the last declared helper object must be used that is caused by a **par** statement.

Nesting Level 3.
The only interesting case here is the following:

```
a:TYPE_OF_LOCAL;
par
  b:TYPE_OF_LOCAL;
  fork
     c:TYPE_OF_LOCAL;
     fork
     end;
  end;
end;
```

```
class HELPER_3b is
  attr helper_1:HELPER_1;
  attr b:TYPE_OF_LOCAL;
  attr c:TYPE_OF_LOCAL;
  create is ... end;
end;
```

At the moment when the transformation reaches the innermost **fork** statement there is no helper object visible that is caused by a **par** statement. In this case, of course that last declared helper object must be chosen.

6.3.2 Pseudo Code

The following pseudo code creates a statement to update an attribute of the helper that mirrors a changed local variable. If a change of a local variable needs not to be updated, the pseudo code does not add a new statement.

Transformation Algorithm update_in_helper*:*

```
1  update_in_helper(local) is
2     -- 1) Find out the appropriate helper object for export
3     if no_helper_is_visible then return;
4     elsif no_par_helper_is_visible then helper_to_use := last_declared_helper;
5     else helper_to_use := last_declared_par_helper;
6     end;
7     -- 2) Copy into helper object if applicable
8     if reach_via_helpers(local,par_helper_of(helper_to_use)) then
9        update_attribute_in_helper(local,helper_to_use)
10    end;
```

Algorithm Description. Nothing is updated if no helper object is visible (line 3). Otherwise, the last declared helper object is considered which is caused by a **par** statement (line 5). If no such helper object can be found the last declared helper object is used instead (line 4). This has been motivated by the examples in the previous section.

The second step of update_in_helper makes sure that only those attributes are written which are accessible from helper objects that correspond to **par** statements. In the above example for nesting level 3, only changes to local variable a will make it into the helper object. The other two attributes of helper_3b cannot be reached in par_helper_of(helper_to_use). In the example the following update code will be created for a change of a:

helper_3b.helper_1.a := a;

6.4 Unpacking of Helper Objects, Import

Helper objects must frequently be unpacked, i.e., the local variables must be set to the values that are stored in the helper object. We must differentiate between two different cases. The first case is the initial unpacking that is needed at the beginning of routines which implement the bodies of **par** or **fork** statements or which implement the right hand side of **attach** statements. The second case is caused by rules **Hlp1** and **Hlp2**, i.e., whenever an import operation occurs that requires the shared local variables to be set to the up-to-date value.

The pseudo code for the unpacking is a mixture of the one for packing of helper objects (see 6.2, update_attribute_in_helper) and the pseudo code for the update of helper attributes (see 6.3.2, update_in_helper).

Transformation Algorithm update_from_helper*:*

```
1  update_from_helper(mode) is
2     -- 1) Find out the appropriate helper object for export
3     if no_helper_is_visible then return;
4     elsif no_par_helper_is_visible then helper_to_use := last_declared_helper;
5     else helper_to_use := last_declared_par_helper;
6     end;
7     -- 2) Consider different modes
8     if mode /= init then
9        helper_to_use := par_helper_of(helper_to_use);
10    end
11    -- 3) Copy into helper object if applicable
12    loop local::=active_locals_plus_params!;
13       if    (  user_defined(local)
14             or (mode = init and is_cohort(local)))
15          and reach_via_helpers(local,helper_to_use)
16       then
17          update_local_from_helper(local,helper_to_use)
18       end;
19    end;
```

Algorithm Description. First of all, the routine decides in the first step which helper object to use. This is done in the same way and for the same reasons as it has been done for the update of helper attributes. In the second step (lines 7 to 10) the mode is considered. If called in init-mode the last declared helper object is the one that is passed as a parameter to the routine implementing the new thread. Since nothing is changed in step 2 (lines 11 to 19), *all* variables that are (transitively) reachable from this helper object are set in the third step.

However, if called in import-mode, helper_to_use is changed to point to the helper object of the surrounding **par** statement, if the current routine does not itself implement a **par** statement. In this case only those variables are copied from the helper object that are mirrored in the helper of the surrounding **par** statement, which exactly implements the sharing of the corresponding variables.

7 Complete Example

In this section we first show a pSather implementation of the consumer-producer problem. Afterwards we present the complete result of the transformation described in this document. Actually, except for the comments which are added manually, the resulting program is produced by the pSather compiler.

7.1 Original pSather Program

```
1 class MAIN is
2    const pnum := 3; -- number of producers
3    const cnum := 2; -- number of consumers
4    const max_prod := 1000; -- number of 'items' a producer creates
5    attr comm_gate:GATE{INT}; -- used to queue 'items'
6    attr prod_gate:GATE; -- used to attach producers
```

The main program consists of a single **par** statement (lines 9 to 18). Inside of this **par** statement, two **loop** statements are used to create the producers and consumers by means of an **attach** and a **fork** statement.

In the first **loop** (lines 11 to 13) pnum producers are attached to the gate prod_gate, i.e., the producers are started to run concurrently. The producers must be attached to a named gate because the consumers must be able to check whether there are any producers left. If all producers have terminated and all produced items have been consumed, the consumers can terminate as well.

In the second **loop** (line 15 to 17) cnum consumers are started by the **fork** statement.

The **par** statement terminates when all consumers have terminated.

```
7    main is
8       comm_gate := #; prod_gate := #;
9       par
10         -- Create producers and attach to prod_gate
11         loop pnum.times!;
12            prod_gate :- producer;
13         end;
14         -- Create consumers
15         loop cnum.times!;
16            fork consumer end;
17         end;
18      end;
19   end;
```

The code of the producer is straightforward: there is a **loop** statement and inside of this **loop** items (which are consecutive INTs in the implementation) are enqueued into the communication buffer comm_gate. If max_prod items have been produced, the **loop** and thus the producer are terminated.

```
20   producer is
21      res:INT:=0;
22      loop -- some work
23         comm_gate.enqueue(res);
24         res := res+1; if res > max_prod then break! end;
```

```
25        end;
26      end; -- producer, this will remove from prod_gate
```

The code of the consumer is slightly more complicated because the consumer not only has to retrieve items from the communication buffer but in addition, the consumer must decide whether to terminate or to continue.

```
27      consumer is
28        loop
29          lock
30          when comm_gate.not_empty then
31            #OUT+comm_gate.dequeue;
32          when comm_gate.empty, prod_gate.no_threads then
33            break!;
34          end;
35        end;
36      end; -- consumer
37    end; -- MAIN
```

Whereas the producer will terminate after max_prod elements are enqueued into the buffer, the consumer uses the multi-branch **lock** statement to decide about termination. The **lock** statement (lines 29 to 34) has two branches. The first branch is entered if there is an element in the communication buffer. If this is the case, the buffer is locked, the element dequeued from the buffer, and the buffer is unlocked again. If there is no element in the communication buffer and there is no thread attached to prod_gate then the consumer can terminate. This is achieved by the **break!** in the second branch of the **lock** statement.

7.2 Result of the Transformation

The following code is basically generated by the implemented transformation of pSather. However, the code is beautified by hand to enhance readability. For example, helper objects and compiler declared temporary variables have been renamed to be more intuitive. Moreover, we re-introduced syntactic sugar that has been lost during the compilation, e.g., incrementing res in the producer is written as res+1 instead of res.plus(1).

To avoid confusion when referring to line numbers in the code, the resulting code starts at line number 100.

```
100   class MAIN is
101     const pnum := 3; -- number of producers
102     const cnum := 2; -- number of consumers
103     const max_prod := 1000; -- number of 'items' a producer creates
104     attr comm_gate : GATE{INT}; -- used to queue 'items'
```

The **main** routine is shortened during the transformation. The whole body of the original **par** statement is moved into the new routine pS_par. Instead first the gate used as **cohort** is created in line 108. Then the helper object pS_par_hlp is created, which is of type PS_PAR_HLP. Since there are no locally declared variables, the helper object has no attributes which otherwise would have been packed afterwards. Before the new thread that executes the body of the original **par** is started, changes are exported. The new helper object is thus made visible to other threads that might use it. The new thread is started in line 111. The thread is attached to the pS_cohort and runs routine pS_par(pS_par_hlp,pS_cohort). The fourth argument of THREAD is void, since the new thread should run, where the original thread was executed. The initial thread continues after evaluating THREAD and is stopped in the **lock** statement of line 112. There the initial thread is blocked until all threads which are attached to the pS_cohort have terminated. Afterwards the initial thread imports any changes other threads might have made.

```
105 main is
106    comm_gate:=#; prod_gate:=#;
107    -- par:
108    pS_cohort::=#GATE;
109    pS_par_hlp::=#PS_PAR_HLP;
110    SYS::export;
111    THREAD(pS_par, pS_par_hlp, pS_cohort, void); -- conc.
112    lock when pS_cohort.no_threads then end;
113    SYS::import;
114 end; -- of main
```

Potential Optimization. In the basic transformation template there is a lot of room for optimization. In the above code fragment some of the problems an optimization phase could address are obvious:

- It is not necessary, to create and and pass empty helper objects.
- If this can be avoided, no export needs to occur, since no objects are changed.
- No import needs to occur if the thread does not use any of the potentially imported new versions of objects.

The code of the producer did not change significantly. However, since import and export operations must be explicit in pSather$_{ie+thread}$ the exclusive gate operation (lines 119) is enclosed in calls of export and import.

```
115 producer is
116    res:INT:=0;
117    loop
118       SYS::export;
119       comm_gate.enqueue(res);
120       SYS::import;
121       res := res+1; if res > max_prod then break! end;
122    end;
123 end; -- of producer
```

The code of the consumer has been changed a little more. In addition to the explicit export and import code that surround the exclusive gate operation in line 130, there are additional import and export operations enclosing each branch of the **lock** statement (lines 128+132 and lines 134+136).

```
124 consumer is
125    loop
126       lock
127       when comm_gate.not_empty then
```

```
128         SYS::import;
129         SYS::export;
130         #OUT+comm_gate.dequeue;
131         SYS::import;
132         SYS::export;
133       when comm_gate.empty, prod_gate.no_threads then
134         SYS::import;
135         break!;
136         SYS::export;
137       end;
138     end;
139   end; -- of consumer
```

The transformation moves the former body of the **par** statement into the new routine pS_par. At the very beginning of the body there is the explicit import operation (line 141) that makes the helper object visible. At the end of the body (line 158) there is an explicit export operation.

Although the two **loop** statements are still present, their bodies have been changed significantly.

Lines 143 to 149 show the transformation of the **attach** statement. Lines 152 to 156 show the transformation of the **fork** statement.

For the **attach** statement, at first a new gate pS_gate is created that is subsequently used instead of prod_gate. Then a helper object of type PS_ATTACH_HLP is created and used to make the local variables and arguments visible in the routine that will implement the right hand side of the **attach**. Here a reference to the surrounding helper object of the **par** statement and the current cohort are copied into this helper object. After an explicit export operation in line 148, the routine pS_attach is started concurrently (line 149).

```
140   private pS_par (pS_par_hlp:PS_PAR_HLP, pS_cohort:GATE, pS_at:INT) is
141     SYS::import;
142     loop pnum.times!;
143       -- attach:
144       pS_gate::=prod_gate;
145       pS_attach_hlp::=#PS_ATTACH_HLP;
146       pS_attach_hlp.pS_par_hlp:=pS_par_hlp;
147       pS_attach_hlp.pS_cohort:=pS_cohort;
148       SYS::export;
149       THREAD(pS_attach, pS_attach_hlp, pS_gate, void); -- conc.
150     end; -- of loop
151     loop cnum.times!;
152       -- fork
153       pS_fork_hlp::=#PS_FORK_HLP;
154       pS_fork_hlp.pS_par_hlp:=pS_par_hlp;
155       SYS::export;
156       THREAD(pS_fork, pS_fork_hlp, pS_cohort, void); -- conc.
157     end; -- of loop
158     SYS::export;
159   end; -- of pS_par
```

The **fork** statement is transformed slightly differently. In contrast to the helper object we have used for the **attach** statement, the helper object PS_FORK_HLP of the **fork** statement does not include the current cohort. The reason for this is that the routine that implements the body of the **fork** statement always has the cohort gate as its second parameter. Since the routine that implements the right hand side of an **attach** statement is in general not attached to the cohort, the cohort had to be passed through the helper object.

Potential optimization. Again several special cases can be noted that could be exploited by optimization:

- It does not make much sense to link empty helper objects into new helper objects, as it is done with the empty helper object ps_par_hlp in lines 146 and 154.
- Data flow analysis could reveal that the thread created for the **attach** statement does not need access to pS_cohort. Therefore, passing this reference through the helper object could be left out without any harm.

The following code fragment shows the routine that results from the transformation of the **attach** statement. In line 161 a local variable is declared that mirrors the one which is visible at the point of the original **attach** statement. After the initial import operation (line 162), the helper object is unpacked, i.e., the mirroring local variables are filled according to the values their original counterparts have at the point of the **attach** statement (line 163).

The producer is started in line 164. The enqueue operation is as uninteresting as the side effect of the initial **attach** statement to count the number of terminated producers. The implementation of THREAD makes sure that the invocation of pS_attach is properly attached to pS_gate, which is needed for the **lock** statement in the consumer (lines 126 to 137) to work.

```
160 private pS_attach (pS_attach_hlp:PS_ATTACH_HLP, pS_gate:GATE, pS_at:INT) is
161    pS_cohort: GATE;
162    SYS::import;
163    pS_cohort:=pS_attach_hlp.pS_cohort;
164    producer;
165    pS_gate.enqueue;
166    SYS::export;
167 end; -- of pS_attach
```

Potential Optimization. See above.

The routine pS_fork does not need much discussion. Since there are no attributes in the helper object, no mirroring local variables need to be declared or initialized. Hence, the body of the routine simply consists of explicit import and export operations that enclose the call of consumer.

```
private pS_fork (pS_fork_hlp:PS_FORK_HLP, pS_cohort:GATE, pS_at:INT) is
   SYS::import;
   consumer;
   SYS::export;
end; -- of pS_fork
```

Potential Optimization.

- Closer analysis can avoid many of the import and export operations that are required by the language specification.

Finally, for the sake of completeness, we present the class definitions of the helper objects:

```
class PS_PAR_HLP is
   create:SAME is
      return new;
   end;
end;    -- of PS_PAR_HLP
class PS_ATTACH_HLP is
   attr pS_cohort : GATE;
   attr pS_par_hlp : PS_PAR_HLP;
   create:SAME is
      return new;
   end;
end;    -- of PS_ATTACH_HLP
class PS_FORK_HLP is
   attr pS_par_hlp : PS_PAR_HLP;
   create:SAME is
      return new;
   end;
end;    -- of PS_FORK_HLP
```

References

[1] Stephen M. Omohundro and David Stoutamire. The Sather 1.1 specification. Technical Report TR-in preparation, International Computer Science Institute, Berkeley, 1995. Available from http://www.icsi.berkeley.edu/Sather.

[2] David Stoutamire. The pSather 1.1 manual and specification. Technical Report TR-in preparation, International Computer Science Institute, Berkeley, 1995. Available from http://www.icsi.berkeley.edu/Sather.