Derivation of Secure Parallel Applications by Means of Module Embedding[1]

Atanas Radenski

Computer Science Department, Winston-Salem State University, P. O. Box 19479
Winston-Salem, North Carolina 27110, USA
radenski@computer.org

Abstract. An enhancement to modular languages called *module embedding* facilitates the development and utilization of secure generic parallel algorithms.

1 Introduction

We have designed and implemented a strictly typed modular language framework that supports the specification of generic parallel algorithms and the derivation of specific parallel applications from such generic algorithms. The focus of our research is on *message-passing* parallelism and cluster computing applications.

A *generic parallel algorithm* encapsulates a common control structure, such as a master-server network, a pipeline, a cellular automaton, or a divide-and-conquer tree. Such a generic algorithm can be used to derive parallel solutions for a variety of problems. The key idea is that the generic algorithm must provide complete coordination and synchronization pattern in problem-independent manner, while its clients must provide only problem-specific *sequential* code in order to derive specific parallel applications.

In our language framework, generic parallel algorithms and their applications are specified as modules. A particular application can be derived from a generic algorithm by means of *module embedding,* a code reuse mechanism that enables the building of new modules from existing ones through inheritance, overriding of procedures, and overriding of types [11].

We have incorporated module embedding in the experimental language *Paradigm/SP*. Our language is an enhancement of SuperPascal, a high-level parallel programming language developed by Hansen [5]. In addition to embeddable modules, the language provides standard message-passing parallel features, such as *send, receive, for-all*, *parallel* statements, and *channel* types. We have developed a prototype compiler, which generates abstract code, and an interpreter for this abstract code.

---

[1] This work has been supported by NASA grant NAG3-2011.

We use Paradigm/SP to specify general parallel paradigms and to derive particular parallel applications from such general paradigms. We use the Paradigm/SP compiler and interpreter to test such paradigms and their derived applications. Once we have established the validity of a Paradigm/SP program, we convert it into efficient C code that runs on top of a cluster-computing library, such as PVM.

We agree with others [3] that "…for a parallel programming language the most important security measure is to check that processes access disjoint sets of variables only and do not interfere with each other in time-dependent manner". We have adopted in Paradigm/SP an interference control scheme that allows *secure* module embedding in above sense. The Paradigm/SP compiler guarantees that processes in derived parallel applications do not interfere by accessing the same variable in time-dependent manner.

In this paper, we introduce the concept of embeddable module and show how a generic parallel algorithm can be specified as an embeddable module. We demonstrate how module embedding can be employed to derive specific parallel applications from generic algorithms. We also explain how module embedding guarantees that processes in derived applications do not interfere by reading and updating the same variable.

2 Specification of Generic Parallel Algorithms as Embeddable Modules

An *embeddable module* encapsulates types, procedures (and functions), and global variables. *Module embedding* enables building of new modules from existing ones through inheritance, and through *overriding* of inherited types and procedures. An embedded module inherits entities that are exported by the embedded module and further re-exports them. A principal difference between module embedding and module import is that an embedded module is contained in the embedding module and is not shared with other modules, while an imported module is shared between its clients. Another difference is that a client module cannot override types or procedures that belong to an imported module, while an embedding module can override types and procedures that belong to an embedded module.

Type overriding allows a record type that is inherited from an embedded module to be redefined by the embedding module by adding new components to existing ones. Type overriding does not define a new type but effectively replaces an inherited type in the embedded module (i.e., in the inherited code) itself. In contrast, type extension, and similarly, sub-classing, define new types without modifying the inherited ones. Further details on module embedding and type overriding can be found in [11].

We demonstrate the applicability of module embedding to generic parallel programming with a case study of a simplified master-server generic parallel algorithm. The master-server generic algorithm (Fig. 1) finds a *solution* for a given *problem* by means of one *master* and *n server* processes that interact through two-way

communication *channels*. The master *generates* a version of the original problem that is easier to solve and sends it to each server. All servers *solve* their assigned problems in parallel and then send the solutions back to the master. Finally, the master *summarizes* the solutions provided by the servers in order to find a final solution to the original problem.

The generic parameters of the master-server algorithm (Fig. 2) include the type of the *problem* to be solved, the type of its *solution*, and three sequential procedures:

- a procedure to *generate* an instance of the problem that is to be solved by server *i*;
- a procedure to *solve* a particular instance of the problem;
- a procedure to *summarize* the *set* of solutions provided by the servers into a final solution.

The generic master-server algorithm provides its clients with a procedure to *compute* a solution of a specific problem. The *compute* procedure incorporates the master and server processes, but those are not visible to the clients of the generic algorithm.

In Figure 3, all components of the master-server generic algorithm are encapsulated in an embeddable module, *MS*. The export mark '*' [3] designates public entities that are visible to clients of module *MS*. Unmarked entities, such as *master* and *server*, are referred to as private. The types of the problem and the solutions are defined as empty record type (designated as double-dot, " .. "). Clients of module *MS* can (1) extend such inherited record types with problem-specific components, (2) provide domain-specific versions of procedures *generate, solve* and *summarize*, and (3) use procedure *compute* to find particular solutions.

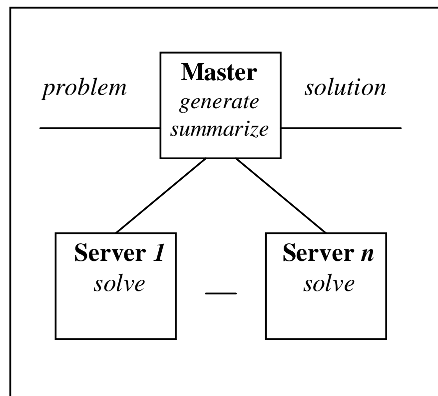
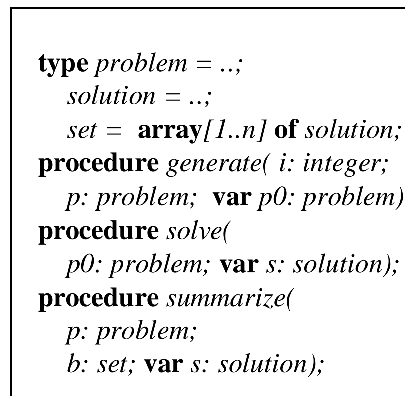


Fig. 1. Generic master-server algorithm



```
type problem = ..;
   solution = ..;
   set = array[1..n] of solution;
procedure generate( i: integer;
   p: problem; var p0: problem);
procedure solve(
   p0: problem; var s: solution);
procedure summarize(
   p: problem;
   b: set; var s: solution);
```

Fig. 2. Generic parameters

```
module MS;
const n = 10; {number of servers}
type
  problem* = ..; solution* = ..;
  set* =  array[1..n] of solution;
  channel =  *(problem, solution);
  net = array [1..n] of channel;

procedure solve*(
  p0: problem; var s: solution);
begin end;

procedure generate*( i: integer;
  p: problem; var p0: problem);
begin { default: } p0 := p; end;

procedure summarize*(
  p: problem;
  b: set; var s: solution);
begin end;

procedure server( c: channel);
var p0: problem; s0: solution;
begin receive(c, p0);
  solve(p0, s0);
  send(c, s0);
end;

procedure master(c: net;
  p: problem; var s: solution);
var  i: integer;
  p0: problem; b: set;
begin
  for i := 1 to n do begin
    generate(i, p, p0);
    send(c[i], p0);
  end;
  for i := 1 to n do
    receive(c[i], b[i]);
  summarize(p, b, s);
end;

procedure compute*(
  p: problem;  var s: solution);
var  c: net; i: integer;
begin
  for i := 1 to n do open(c[i]);
  parallel
    master(c, p, s) |
    forall i := 1 to n do
      server(c[i])
  end;
end;

begin end. {MS}
```

Fig. 3. Embeddable module *master-server*, *MS*. Public entities are marked by '*'.

3 Derivation of Specific Parallel Algorithms By Means Of Module Embedding

A parallel generic algorithm is a common parallel control structure (such as master-server) in which process communication and synchronization are specified in a problem-independent manner. Clients of the generic algorithm can derive particular applications from the generic algorithm by extending it with domain specific sequential algorithms. When the generic algorithm is specified as a module, the derivation of specific applications can be achieved by means of module embedding. An application module can embed the generic master-server module and override relevant entities that are inherited from the embedded module, giving them more specialized meaning. This is explained in details in the next section.

3.1 Derivation of Parallel Integration Application

Consider, for example, the problem of deriving a simple parallel integration algorithm based on the trapezoidal method. This can be achieved by extending module *MS* into a module *TI* (Fig. 4). The *embedding module, TI,* inherits the components of the *base module, MS* and *re-exports* all inherited public entities. Besides, module *TI* introduces a new generic parameter *f*, the function to be integrated that should be supplied by clients of *TI*.

The embedding module, *TI*, overrides the inherited type *problem*, so that the new *problem* definition incorporates the lower and upper limits *a, b* of the integral to be calculated. Similarly, *TI* overrides the inherited type *solution*, so that the new *solution* definition incorporates the integral value *v*. Note that *problem* and *solution* were originally defined in module *MS* as empty record types. Overriding of non-empty record types is also permitted, as illustrated in the next section.

The embedding module also overrides the inherited default version of procedure *generate* and the inherited 'null' versions of procedures *solve* and *summarize*. The newly declared version of *generate* divides the integration range into *n* equal parts, one for each server. Procedure *solve* is defined in *TI* to be trapezoidal integration. Procedure *summarize* sums-up the partial integrals provided by the *n* servers.

```
module TI(MS);
type problem* =
   record  a*, b*: real; end;
  solution* = record v*: real; end;

function f*(x: real): real;
begin end;

procedure solve*(p0: problem;
  var s: solution);
begin s.v := ((p0.b - p0.a) / 2) *
              (f(p0.a) + f(p0.b));
end;

…complete implementations of
procedures generate and
summarize…
…
end. {module T1}

module IA(TI);
var  p: problem;  s: solution;

function f*( x: real): real;
begin f := x * sin(sqrt(x)); end;

begin compute(p, s) end. {LA}
```

Fig. 4. Derived modules *trapezoidal integration, TI*, and *integration application, IA*.

Module *TI* can be embedded on its turn into a specific *integration application* module, *IA*, that defines a particular function *f* to be integrated. Module *IA* serves as a main program by invoking procedure *compute* that is provided by the generic *MS* module (Fig. 4).

3.2 Derivation of Parallel Simulated Annealing and Traveling Salesperson Algorithms

A variety of specific parallel algorithms can be derived from the same general parallel generic algorithm. For example, we have derived a generic algorithm for approximate optimization that is based on *simulated annealing*, organized as module *SA* (Fig. 5). Note that the definition of type *annealingPoint* contains a component, *dE*, that is needed for all possible application of simulated annealing..

The generic *simulated annealing* algorithm can be used to derive approximate algorithms for different intractable optimization problems. For instance, we have derived a parallel algorithm for a particular *traveling salesperson problem* (module *TSP* in Fig. 5). Note that the inherited definition of type *annealingPoint* is overridden in *TSP* by adding two new problem-specific components, *i, j,* to the inherited component *dE*.

```
module SA(MS);
type
  problem* = record
    ...annealing parameters...
  end;
  annealingPoint* = record
    dE*: real;
  end;

 ...procedures select and change
 declared  as generic parameters...

procedure solve*(
  p0: problem; var s: solution);
 ...complete implementation that
 performs simulated annealing
 using the generic parameters
 select and change...
end. {SA}
```

```
module TSP(SA);
...
type
  city = record x, y: real end;
  tour = array [1..m] of city;
  solution* = record t: tour; end;
  annealingPoint* = record
    ...field dE inherited from SA...
    i, j: integer;
  end;

var p: problem; s: solution;

 ...complete implementations of
 procedures select, change,
 summarize...

begin compute(p, s) end. {TSP}
```

Fig. 5. Derived modules *simulated annealing*, *SA*, and *traveling salesperson*, *TSP*.

4 Interference Control For Embeddable Modules

When a parallel application is executed *repeatedly with the same input*, the relative speeds of its constituent parallel processes may vary from one execution to another. If one parallel process *updates* a variable and another process *updates or reads* that

same variable, the order in which those processes access the variable may vary from one execution to another, even when the input for the parallel application do not change. Such parallel processes are said to *interfere* with each other in a time dependent manner due to a *variable conflict.* Interfering parallel processes may update and possibly read the same variable at unpredictable times. The output of an application that contains interfering parallel processes may vary in an unpredictable manner when the application is executed repeatedly with the same input. Such an application is said to be *insecure* due to a *time-dependent error.* A secure parallel programming language should allow detection and reporting of as many time-dependent errors as possible. The implementation may efficiently detect time-dependent errors through process *interference control* at compile time and, less efficiently, at run time.

Hansen [4] advocated the benefits from interference control and developed an interference control scheme for the parallel programming language SuperPascal. The SuperPascal language is subject to several restrictions that allow effective syntactic detection of variable conflicts, i.e., detection at compile time. These restrictions apply to a variety of language constructs and assure that a variable that is updated by a parallel process may be read only by that process. Note that parallel processes are allowed to read-only shared variables.

For each statement belonging to a SuperPascal program, the compiler determines the target variable set and the expression variable set of that statement. The *target variable set* consists of all variables that may be updated during the execution of the statement, while the *expression variable set* consists of all variables that may be read during that statement's execution. In SuperPascal, processes are created by *parallel* and *forall* statements. A parallel statement **parallel** $S_1 \mid S_2 \mid \dots S_n$ **end** incorporates *n process statements* S_1, S_2, … S_n such that the target variable set of S_i is disjoint with the target and expression variable sets of $S_1, \dots S_{i-1}, S_{i+1}, \dots S_n$, $i = 1, 2, \dots n$. A forall statement **forall** $i := m$ **to** n **do** S incorporates a single *element statement S* which generates $n\text{-}m\text{+}1$ processes and, for this reason, is required to have an empty target variable set.

It should be noted that the above restrictions on target and expression variable sets are very natural for parallel applications running in a cluster computing environment. Processes that are generated by a *forall* statement will run on separate cluster nodes. If such processes were to share a target variable, it could be quite hard and inefficient to synchronize that shared access over a network. At the same time, it is easy to make these processes efficiently share read-only variables by broadcasting those variables values just once to all processes. Similar considerations apply to processes that are generated by a *parallel* statement.

A SuperPascal parallel application consists of a single main program, exactly as in the standard Pascal language. The interference control scheme of SuperPascal [4] guarantees that single-module parallel applications do not contain time-dependent errors, i.e., they are secure in this sense. The Paradigm/SP language has been designed as an extension to SuperPascal that introduces separately compiled embeddable modules [11]. We have extended the single-module interference control scheme of SuperPascal to serve the specific requirements of Paradigm/SP.

In SuperPascal, procedures are never overridden. Therefore, the target and expression variable sets for procedure statements can be determined during the compilation of SuperPascal's single module parallel applications. This is not the case in a language with embeddable modules, such as Paradigm/SP: procedures that are defined in an embeddable module *M0* can be overridden in an embedding module *M1*. The overriding procedures that are defined in *M1* may have different target and expression variable sets from those in *M0*. Therefore, procedure statements in the embedded module *M0,* a module that has already been separately compiled, may have their target and expression variable sets changed by procedure overriding in *M1*. Thus, restrictions on target and expression variables sets that have been validated during the compilation of *M0* may be violated later, when *M0* is embedded in *M1*.

```
module M0;
  procedure p*(j: integer);
  begin end;
begin
   forall i := 1 to 10 do p(i);
end. {M0}
```

```
module M1(M0);
  var  k: integer;
  procedure p*(j: integer);
  begin k := j end;
begin k := 0 end. {M1}
```

Fig. 6. Modules *M0* and *M1*.

Consider, for example, module *M0* from Fig. 6 that defines and exports procedure *p*. Module *M0* contains a statement **forall** *i := 1* **to** *10* **do** *p(i)* that generates processes by executing the procedure statement *p(i)*. The procedure statement *p(i)* has an empty target variable set; therefore, its generated processes do not interfere due to variable conflicts.

Assume now that module *M0* is embedded in module *M1* and that *M1* overrides the inherited procedure *p*, as illustrated in Fig. 6. The overriding body of *p* may have access to a global variable, *k*. Therefore, the target variable set of the procedure statement *p(i)* in the separately compiled module *M0* will now actually contain the variable *k*, and will, therefore, be non-empty.

The main difficulty to interference control in a language framework with module embedding comes from the possibility to change, through procedure overriding, the target and expression variable sets in embedded modules that have been already separately compiled. We remedy this problem by introducing additional restrictions that make it impossible to modify variable sets during procedure overriding. More precisely, we exclude the so-called unrestricted procedures (and functions) from *parallel* and *forall* statements, as explained below.

A procedure that is declared in a module can be marked for export with either a *restricted mark* "*" or an *unrestricted mark* "-". A procedure exported by a module *M0* can be overridden in an embedding module *M1*, provided that the procedure heading in *M1* is the same as in *M0* (in particular, the export mark, "*" or "-", must be the same).

A *restricted procedure* is a procedure exported with a restricted mark, "*".

An *unrestricted procedure* is:

- a procedure that is exported with an unrestricted mark, "-", or
- a private procedure that invokes an unrestricted procedure.

A restricted procedure is not permitted to use global variables (directly or indirectly), and invoke unrestricted procedures. In contrast, an unrestricted procedure can use global variables and invoke unrestricted procedures.

Overriding an unrestricted procedure *p* in an embedding module *M1* may change target and expression variable sets in a separately compiled embedded module *M0*, because the overriding procedure is allowed to access global variables. This is why *parallel* statements and *forall* statements are *not* permitted to invoke unrestricted procedures. This requirement is in addition to the limitations on target and expression variable sets in *parallel* and *forall* statements, as defined earlier in this section. Restricted procedures are not excluded from *parallel* and *forall* statements because, in contrast to unrestricted procedures, they cannot modify target or expression variable sets in separately compiled embedded modules,

There are also procedures that are neither restricted nor unrestricted, such as, for example, private procedures that use global variables but do not invoke unrestricted procedures. This category of procedures may participate in *parallel* and *forall* statements as well, as far as they comply with the limitations on target and expression variable sets, as discussed earlier in this section.

Consider again the example modules in Fig. 6. Procedure *p* is declared with a restricted mark, "*". Therefore, accessing a global variable such as *k* in *M1* is a syntax error. Procedure *p* would be allowed to access a global variable if *p* was declared with an unrestricted mark, "-". In such a case, however, the use of *p* in a *forall* statement like as the one in *M0* would be a syntax error..

The exclusion of unrestricted procedures from *parallel* and *forall* statements permits syntactic detection of variable conflicts in separately compiled modular parallel applications. The Paradigm/SP compiler guarantees that a variable that is updated by a process cannot be used by another process, while sharing read-only variables is permitted. Paradigm/SP parallel applications may not be insecure due to variable conflicts.

Is the exclusion of unrestricted procedures form *parallel* and *forall* statements a serious practical limitation? Technically, it means that if an exported procedure is used to generate a process, and if it needs to access global variables, it must do so through explicit send/receive statements or through parameters, rather than directly. We are convinced that this restriction is quite natural in the domain of message passing cluster algorithms, because parallel access to global variables from different processes must be implemented through send/receive, anyway. Programmers who are forced to implement access to global variables through explicit send/receive statements are more likely to be aware of the underlying inefficiency of such access, in contrast to programmers for whom implicit message passing is generated by the implementation. Our experiments with four generic algorithms and several derivatives from each of them make us believe that the exclusion of unrestricted procedures form parallelism is not a serious practical limitation.

5. Conclusions

This paper outlines module embedding, a form of inheritance that applies to modules and that permits overriding of inherited types. Embeddable modules have been incorporated in a parallel programming language called *Paradigm/SP*. A prototype implementation of Paradigm/SP has been developed and documented [12]. Paradigm/SP has been used to specify generic parallel algorithms and to derive concrete parallel applications from them by means of module embedding. Paradigm/SP has been used as a higher-level prototyping language in order to conveniently test the validity of derived parallel applications before finally converting them into efficient C code that runs in a cluster-computing environment, such as PVM.

We have specified several generic parallel algorithms as embeddable modules, such as a probabilistic master-server [10], a cellular automaton [9], and an all-pairs pipeline [8]. Though module embedding, we have derived diverse parallel applications from such generic algorithms. Despite of the use of generic parallelism, most of the derived applications have demonstrated very good performance in cluster-computing environments, and a couple of derived applications have achieved super linear speed-up [8].

We have adopted interference control scheme for embeddable modules. This scheme guarantees that processes in derived applications do not interfere by reading and updating the same variable. That derived algorithms are *secure* in this sense is what makes module embedding unique in comparison to traditional object-oriented techniques supported by C++, Java, Corba, etc., where no static control helps programmers to avoid time-dependent errors in derived algorithms. For example, it has been recognized that Java multithreaded applications are inherently insecure because nothing prevents different threads from invoking unsynchronized methods [3]. A related insecure feature of Java is that data members are by default *protected* and that protected data members can be accessed from all classes that belong to the same package. For these reasons, it easy to gain access from different threads to protected data members by adding new classes to a package and to create applications that are insecure due to time-dependent errors.

Others have proposed dynamic load-time class overriding through byte-code editing [6]. This technique is justified by the so-called adaptation and evolution problems that appear when sub-classing is used to build software components. Our approach has the merit of integrating type overriding within the programming language and its compiler.

In traditional modular object-oriented languages, such as Oberon-2, Ada-95 and Modula-3, modules are not embeddable, while classes are represented by means of extensible record types [15]. What is different in our approach to classes is that record type extension overrides an existing type (both in the new embedding module and in the existing embedded module) and does not introduce a new type. A disadvantage of embeddable modules as compared to classes is that modules do not introduce types, and therefore cannot be used to create multiple instances. Furthermore, inherited type overriding imposes additional run-time overhead on the

implementation. It has been recognized [7], [13] that both modules and classes support necessary abstractions, which should be used as a complementary techniques.

A collection of object-oriented language features that support the development of parallel applications can be found in [1], [2]. Parallel programming enhancements of a mainstream language, C++, are presented in [14]. A survey of earlier object-parallel languages is contained in [16]. An example of template-based genericity is contained in [17]. We do not know of a traditional object-oriented language that performs static analysis in order to guarantee that parallel applications are free of time-dependent errors. The main benefit of module embedding is that it guarantees at compile time the lack of such errors and that its static interference analysis scheme eliminates the overhead of run-time synchronization.

Paradigm/SP is a specification and prototyping language and as such is simpler than production languages and environments. Algorithm developers may focus on what is essential in their developed parallel control structures and application methods without being burdened by the complex details that are required for efficient practical programming. Simplicity and ease of use are advantages of Paradigm/SP as an algorithm development and validation language in comparison to production languages and environments.

As a continuation of this project in the future, we envision that it would be possible and beneficial to develop an interference control scheme for multithreaded Java applications. A Java source code analyzer may be used to discover variable conflicts between threads and to help eliminated time-depending errors due to such conflicts.

If algorithms are to be published on the web, they can be shaped as multimedia web-pages. A separately compiled module can be shaped as a source html file that can be fed into a compiler in order to produce executable code. Module import and embedding can be designated by means of hyper-links. Source modules that comprise an application can reside on different servers. These same servers can host corresponding distributed executable objects. The design of adequate language and compiler support is another possible continuation of this project in the future.

References

1. G. Agha, P. Wegner, and A. Yonezawa, editors. *Research Directions in Concurrent Object-Oriented Programming*. MIT Press, 1993.
2. J.-P. Briot, J.-M.Geib, and A. Yonezawa. *Object-Based Parallel and Distributed Computation*. Lecture Notes in Computer Science 1107, Springer, 1996.
3. P. B. Hansen. Java's Insecure Parallelism. *ACM SIGPLAN Notices*, Vol. 34, No 4, April 1999, pp.38-45.
4. P. B. Hansen. *Studies in Computational Science: Parallel Programming Paradigms*. Prentice Hall, 1995.
5. P. B. Hansen. SuperPascal - A Publication Language For Parallel Scientific Computing. *Concurrency - Practice and Experience*, 6, No 5, 1994, 461-483.

6. R. Keller, U. Holzle. Binary Component Adaptation. *ECOOP'98 Conference Proceedings* (E. Jul, editor), Lecture Notes in Computer Science 1445, Springer, 1998, pp.307-329.
7. H. Moessenboeck. *Object-Oriented Programming in Oberon-2*. Springer, 1993.
8. A. Radenski, B. Norris, W. Chen. A Generic All-Pairs Cluster-Computing Pipeline and Its Applications. *Proceedings of ParCo99, International Conference on Parallel Computing, August 17-20, 1999, Delft, The Netherlands*, Imperial College Press (under print).
9. A. Radenski, A. Vann, B. Norris. Development and Utilization of Generic Algorithms for Scientific Cluster Computations. *Object Oriented Methods for Interoperable Scientific and Engineering Computing* (M. Henderson, C. Anderson, and S. Lyons, editors), SIAM, 1999, 97-105.
10. A. Radenski, A. Vann, B. Norris. Parallel Probabilistic Computations on a Cluster of Workstations. *Parallel Computing: Fundamentals, Applications and New Directions* (edited by E. D'Hollander et al.), Elsevier Science B.V., 1998, 105-112.
11. A. Radenski. Module Embedding. *Intl. Journal Software - Concepts and Tools*, Vol. 19, Issue 3, 1998, 122-129.
12. A. Radenski. Prototype Implementation of Paradigm/SP, http://www.rtpnet.org/~radenski/research/, 1998.
13. C. Szyperski. Why We Need Both: Modules And Classes. *OOPSLA'99 Conference Proceedings*, ACM, 1992, pp.19-32.
14. G. Wilson and P. Lu, editors. *Parallel Programming using C++*. MIT Press, 1996.
15. N. Wirth. Type Extensions. *ACM Transactions on Programming Languages and Systems*, 10, 1987, 204-214.
16. B. Wyatt, K. Kavi, and S. Hufnagel, 1992. Parallelism in Object-Oriented Languages: A Survey. *IEEE Software*, 9, No 6 (Nov.), 1992, 56-66.
17. L.-Q. Lee, J. Siek, and A. Lumsdaine, 1999. The Generic Graph Component Library. *OOPSLA'99 Conference Proceedings*, ACM, 1999, pp.399-414.