Component Based Simulation of ATM Switch Fabrics

Tanir, O.; Erdogmus, Hakan

COMPONENT BASED SIMULATION OF ATM SWITCH FABRICS

Oryal Tanir

Bell Canada, Quality Engineering & Research
2nd floor, 1050 Beaver Hall, Montreal, QC Canada
H2Z 1S4 – *Oryal.Tanir@Bell.ca*

Hakan Erdogmus

National Research Council of Canada, IIT-S/W Eng..
Montreal Road, Bldg. M-50, Ottawa, ON, K1A OR6, Canada

KEYWORDS

Reuse, Simulation Environment, component, structure, architecture.

ABSTRACT

As designs have become more complex, the need for tools to support reuse in the early stages of the design process has become increasingly important. Such tools have the potential to facilitate the simulation and evaluation of designs far before actual implementation. To support these applications, simulation environments must be able to support the reuse of models in many ways. This paper complements work regarding structural reuse and presents some insight in designing models that are amiable for component based simulation.

INTRODUCTION

In the design of computer-based systems, simulation tools employed during various stages of the design cycle can provide significant insight and knowledge into the behavior of the proposed design. Unfortunately, the knowledge gained through the course of a simulation exercise is typically lost and inaccessible to other designs. One promising solution to this problem is the utilization of development environments that can support libraries of models at high levels of abstraction — more suitable for reuse. Languages such as the *Design Specification Language (DSL) (Tanir 1997)* allow designers to model, experiment with, and reuse model components very early on in the design life cycle. There exists two explicit forms of reuse: structural and component based.

In structural reuse (Shaw and Garlan 1992), (Monroe and Garlan 1996) (sometimes called context reuse (Biddle and Tempero 1996)) the objects of reuse are not the individual artifacts that make up a system description, but rather the "contexts" in which these artifacts are embedded. Thus structural reuse is based on the identification, isolation and exploitation of organizational patterns recurring across system descriptions. An organizational pattern is expressed in terms of configurations of *abstract* components that serve as placeholders for real, or *concrete*, components. Such patterns are often parameterized, making them generic — and hence more amenable to reuse.

A framework for reuse supporting three levels of abstraction is proposed in (Tanir and Erdogmus.1997a). These levels are defined as the topological, architectural, and system layers. Structural reuse commences at the *topological level* with a clear separation of structure from functionality (behavior). At this high level of abstraction, the gross structure of a system is modeled. The next level of abstraction is the *architectural level*, where more detail can be added, yet components are still not bound to any particular functionality. Component interfaces and connectivity relations may be refined when moving from a topological specification to an architectural one.

Representation of systems at the topological and architectural levels can be accomplished using the *Extended Style Notation*. ESN is a strongly typed, interpreted functional language based on the graph algebra defined in (Tanir Erdogmus 1997b) and the architectural model of (Erdogmus.1995). It has special constructs for expressing topologies and classes of topologies (the *style* sublanguage), refinement rules (the *map* sublanguage), and architectures (the *template* sublanguage).

The third and the lowest level of abstraction in the framework is the *system level*, where each concrete component is assigned a functionality. At this level component reuse can be exploited through languages such as the Design Specification Language (DSL). Figure 1 depicts the overall framework. An overview of the two languages is given below.

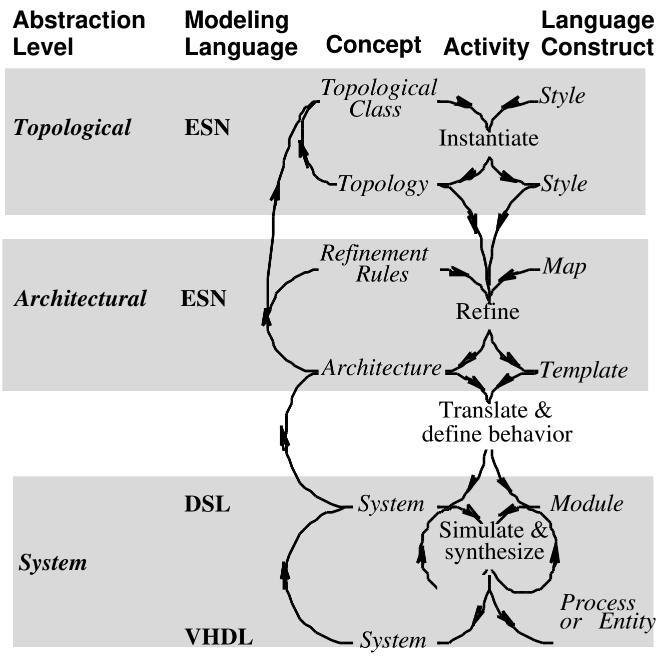

Figure 1: A methodology for structural reuse.

ESN

ESN is used for topological and architectural representation. An overview of some key features of ESN is presented below. Once the structural form is obtained, binding of components to behavioral entities (which can be described by DSL modules) is required before simulation of the complete system.

Expressions of type style in ESN specify topologies. Formally, a topology is a finite graph consisting of typed (and named) nodes and named binary edges. A parameterized style definition (a generic style) is often thought of as specifying a topological class, which when instantiated with actual parameters, is evaluated into an instance, yielding a particular topology. Expressions of type template specify architectures, and those of type map specify refinement rules. Constants of type name (which always begin by a backslash) stand for themselves, and can be indexed by an integer vector; e.g., \a, \a<1, 2>. A definition invocation must be prefixed with the type of the value returned, e.g., name a, int b[k], style S[~a, 1]. The default type is int (which is ordinarily omitted) and the symbol ~ is often used as a shorthand for the type keyword name. Strong typing allows the inference of the type of any ESN expression independent of the context.

The language construct *template* is used to specify architectures. The architecture definition consists of an *interface*, an *internal structure*, and a set of *external bindings*. An *interface* consists of a set of interaction points, called *ports*. The interface allows an instance of a system to be connected with instances of other or similar systems. *Internal structure* is defined in terms of a set of components and a set of internal bindings between these components. Each such component is an instance of some system. An interface port of a component is referred to as an *internal port*. Hence, a component and an interface port of the component's parent identify each internal port. A *binding* models a connection between two components. An *internal binding* involves a pair of internal ports. An *external binding* involves an internal port and an interface port. In a composite system, the set of external bindings relates the internal structure to the interface.

DSL

DSL is a specification language that provides the necessary representation mechanisms at the system level of design. It is employed within DASE (Tanir 1997 and described later in the paper), a rapid protoyping and synthesis tool that was developed at McGill University and Bell Canada. DSL is based on Prolog in which DASE is implemented. This section will summarize some of the basic concepts of the language.

The basic construct in DSL is the *module*. Modules are the primitive building blocks of a system. A module has a *name*, a set of possible *behaviors*, and a set of *resources*. A module is defined as follows:

```
module(module_name, [
  behavior1,
    ...
  behaviorn,
]).
```

Each behavior consists of a guard and a sequence of actions. If the guard is satisfied, the associated actions are executed in the given order. The guard is a predicate that holds true upon the reception of a message from another module. An action can

- initiate communication with other modules,
- modify or query the local resources of the module, or
- suspend the execution of the module for a specified time period.

Modules can possess their own unique

behavior or *inherit* the behavior of other modules. They can also store local data through a *resource* construct.

Inter-module communication is realized using the send construct (action) whose general form is:

```
send(destination, port, message)
```

where *message* is the message to be sent; *port* specifies an output port from which the message is to be sent; and *destination* is the destination module. When this field is blank, the message is sent to the first module connected to *port* and that is capable of interpreting the message. Unspecified parameters are synthesized during simulation – a powerful feature of DSL. A DSL simulator creates the data structures necessary for communication, and takes care of internal queuing of incoming and the scheduling of outgoing messages.

DSL permits hierarchical specifications through composition of modules into *higher order* (*ho*-) modules. Connections between modules within a ho-module are specified through path statements. A path statement binds a port of one module to a port of another module. For example,

```
path(modX, modY, [portX, portY])
```

connects portX of modX to portY of module modY.

Note that since ports are not typed, any message can be sent from or received at any port. A module's behavior only refers to output ports. Thus, there is no knowledge within a module as to which port incoming messages will be received. Path statements are optional - bindings between ports are synthesized by a simulator at run-time.

DSL also has a special library mechanism to support component reuse and design space exploration. But the discussion of this topic is beyond the scope of this paper.

EXAMPLE OF AN ATM FABRIC

This section illustrates the way components can be defined for reuse through a small example of a Asynchronous Transfer Mode (ATM) Knockout switch fabric (Awdeh and Mohftah 1995). The switch fabric constitutes the most complex layer of an ATM switch, however the behavioral details of the fabrics are omitted.

The example takes advantage of the ESN *package* facility. *Packages* are like containers used to group related definitions together within a common name space. In this example, two ESN *packages* are assumed: (1) package BASE which contains definitions that can be reused across many application domains and (2) package ATMS which contains definitions particular to the ATM application domain.

Reuse of generic styles defined in the package BASE is possible in many ways to generate different topologies. Two key styles in this package have signatures style Parallel[style, style, name 1, int 1, int 1] and style NStar[name, name 1, int 1, name, name 1, name 1]. The first accepts two styles as arguments. It displaces external names of the second style (specified by the third and fourth arguments) by a corresponding value (specified by the fifth argument). This intermediate result is then concatenated with the first style argument to produce a new "parallel" style.

The second generic style defines a class of "star" topologies. It returns an instance with a central node of type ~C and K types of satellite nodes. There are m[k] satellite nodes for each node type ~A[k], k ranging from 1 to K. The central node is named ~c, and the satellite nodes of type ~A[k] are named from ~a[k]<0> to ~a[k]<m[k]-1>. The edge named ~AC[k]<j> connects the satellite node ~A[k] having name ~a[k]<j> to the central node.

```
def style NStar[~C, ~c, m(K), ~A(L), ~a(M),
~AC(N)]
    is
      ~c node ~C
      rep k from 1 .. K in
        rep j from 0 .. m[k]-1 in
          ~a[k]<j> node ~A[j]
          edge ~AC[k]<j><~a[k]<j>, ~c>
        end rep
      end rep
    pre
      distinct <~C, ~A[1 .. K]>, distinct <~c, ~a>,
same <K, L, M, N>,
      rep k from 1 .. K in
        distinct <~a[1 .. m[k]]>, distinct <~AC[1 ..
m[k]]>,
      end rep
    post
      ...
end def;
```

The pre- and post-conditions specify the properties that must hold for the arguments and the instance returned.

Topological description

An M x M Knockout network consists of M parallel streams, where the kth stream consists of a central node of type \K representing a *concentrator* component. M nodes of type \I<k> represent the *address filter* components (entry nodes), and an \O-node represents a *shared buffer* component (exit node). Each of the \I<k>-nodes and the \O-node are connected to the central \K-node through the edges named IK<j> (where j ranges from 0 to M-1) and \KO, respectively. The \I<k>- and \O-nodes are external, named from \I<0> to \I<M-1> and \O<0> to \O<M-1>, respectively. It is not possible to generate the M parallel streams using the generic "parallel" style defined in the BASE package since the index k in the \I<k>-nodes is different for each stream. An instance of the Knockout style is shown in Figure 2.

```
     private def style Knockout[M]
       is
         let
           style KStream[M] be
             style NStar[\K, \k, <M, 1>, <\I<M-1>,
\O>, <\i, \o>, <\KI, \OK>]
             hide \k
             ren \OK<0> as \OK
         in
           branch
             case M = 1 then style KStream[1]
             otherwise
               style Knockout[M-1]
               (style KStream[M] ren ~o<0> as
~o<M-1>)
           end branch
```

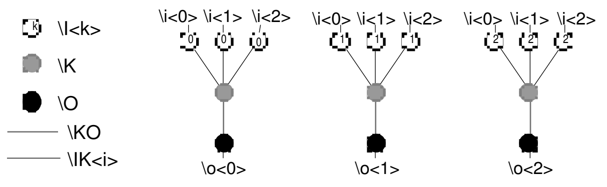

Figure 2: Graph of *style Knockout[3]*.

```
       pre
         M > 0
       post
         ...
     end def;
```

Architectural representation

The next step in the modeling process is to create a composite template that represents the architecture of the associated switch fabric. The application of refinement rules that are defined by the corresponding map (to a proper topology specified by an instance of the underlying style) accomplishes this task. The instantiation of a private style of the package ATMS into a fixed topology to produce an architectural template for the corresponding subsystem is an example structural reuse.

As with styles, where applicable, parameterization of each template is by integer values that specify the *size* of the desired instance. Other parameters may also be added if required. The switch fabric also requires primitive templates to represent internal components. These are the *address filter* (temp A[int]), *concentrator* (temp K[int, int]), and *shift buffer* (temp SB[int]) components. The concentrator component specification includes the integer parameters M and N for a Knockout concentrator with M input ports and N output ports. Finally, the shift buffer component is specified with an integer parameter N which represents the number of input ports.

The architectural model of the switch fabric is more detailed than its topological model. The concentrator node of the Knockout switch is refined into a corresponding component with M input ports and N output ports (where M > N). Similarly, the shift buffer node is refined into a component with N input ports and a single output port. (Note that a template does not actually distinguish between input and output ports; here the distinction is made to facilitate understanding.) The detailed map of a knockout fabric was presented in (Tanir and Erdogmus, 1997b) and will not be repeated here.

The architecture of a generic switch fabric can now be defined as a *variable* synthesized template which makes it possible to choose between a Knockout and any other fabric *on the fly*.

```
     def temp SF[M]
       is
         extend
           choose
             choice …

             choice KNOCKOUT[N] is
               apply                          map
Knockout_to_KNOCKOUT[M, N] to style Knockout[M]
           end choose
         with
           interface \r
         end extend
       pre
```

```
        M > 0
      post
        interface temp this seteq {\i<0 .. M-1>, \o<0
.. M-1>, \r}
      translations
        ...
    end def;
```

Variability is expressed by the choice construct, which can be used in any type of expression. Whether to express component variability at the architectural or topological level is a design decision. In this example, the variability of the switch fabric architecture could have been expressed at the topological level in terms of a variable “fabric” style.

The architecture of the fabric of a Knockout switch can be generated with a defined internal concentration factor. For example, a 4 x 4 Knockout switch fabric with internal concentration factor of 3 is illustrated in Figure 3.

Components of a Knockout Fabric

The Knockout fabric requires some synchronous control. Consequently, a clock message is necessary to harmonize the transfer of data internally within a fabric.

Address filters, shift buffers, and (Knockout) concentrators are the components used in a fabric of a Knockout switch. The implementation given here is a simplification over the original. The difference is in the way the interaction between the shift buffer and the concentrator is achieved. Typically the shift buffer would be realized through a more complicated shift-and-store scheme than the one presented here. The rational for such a scheme is to provide some degree of fairness when processing the incoming messages.

Address Filters

Each address filter component is connected to a specific input port of the fabric. An address filter only accepts messages (cells) destined to it, and ignores the others. A component of this kind is represented in the ATMS package by the primitive template *temp* A[*int*] with an integer parameter. The translation rules for this template as are follows:

```
    def temp A[k]
      is
        ...
      translations
        with DSL use "address_filter(%(k))" where
          \i is "cell_in",
          \o is "cell_out"
```

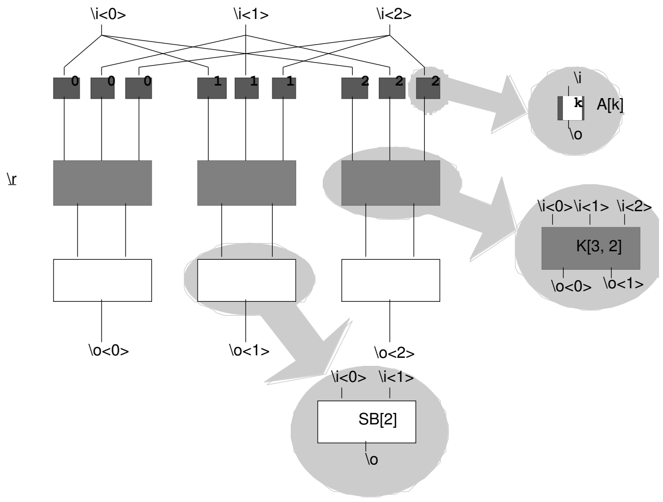

Figure 3: The template *temp SF[4]@KNOCKOUT[3].*

```
        end with
      end def;
```

To implement an instance *temp* A[k] of this template, the DSL module address_filter is instantiated with the value of parameter k. As in a Banyan fabric, a new cell arrives at an input port of a Knockout fabric in the form of an atm_cell message with an address parameter. An address_filter module accepts an atm_cell message only if the address parameter of the message matches the module's own Address parameter. Upon accepting this message, the module forwards it to a concentrator component.

```
      module(address_filter(Address), [
        (atm_cell(VPI, VCI, Data, Address) :-
          delay(Latch_delay),
          send(_, cell_out, atm_cell(VPI, VCI, Data))),
(atm_cell(NVPI, NVCI, DX, Any_other_address))
      ]).
```

Knockout Concentrators

A Knockout concentrator is represented by the primitive template *temp* K[*int*, *int*]. This template is to be implemented by a DSL module concentrator with a single parameter which is bound to the second parameter of *temp* K[*int*, *int*]:

```
      def temp K[M, N]
        is
          ...
        translations
          with DSL use "concentrator(%(N))" where
            rep k from 0 .. M-1 in
              \i<k> is "cell_in(%(k))"
            end rep,
            rep k from 0 .. N-1 in
              \o<k> is "cell_out(%(k))"
            end rep
          end with
        end def;
```

The concentrator module has a resource called register with a list parameter V1 which can store up to a fixed number of cells.

resource(concentrator(X), register(V1), []).

The resource stores up to L cells — which represents the capacity of the concentrator, and also the internal concentration factor. This is the only parameter of the module concentrator. When the resource is full, new cells received from an address filter component via an atm_cell message are discarded. When a clock message arrives, the contents of the resource are transmitted as soon as possible to a shift buffer component. This is accomplished through the *internal* message empty_out which the module sends to itself.

```
module(concentrator(L), [
  (atm_cell(VP, VC, D) :-
    check_res(register, Q),
    list_length(Q, Len),
    Len < L,
    set_res(register, [D | Q])),
  (atm_cell(VP, VC, D) :-
    print("Cell", VP, VC, " dropped from concentrator.")),
  (clock :-
    check_res(register, Q),
    delay(Latch_delay),
    send(empty_out(L, Q))),
  (empty_out(0, X) :-
    print("Concentrator latched outputs.")),
  (empty_out(N, [Top | Rest]) :-
    N is N-1,
    send(_, cell_out(N), Top),
    send(empty_out(N, Rest)))
]).
```

Shift Buffers

A shift buffer accepts messages from a concentrator component and forwards them off to an output interface unit of the switch. The primitive template representing a component of this type is $*temp* SB[*int*]$. An instance of this template is implemented by the DSL module shift_buffer.

```
def temp SB[N]
  is
    ...
  translations
    with DSL use "shift_buffer" where
      rep k from 0 .. N-1 in
        \i<k> is "cell_in(%(k))"
      end rep,
      \o is "cell_out"
    end with
  end def;
```

The behavior of the module shift_buffer is simple. If there are no incoming cells, it receives an empty message [] which is immediately discarded. Otherwise, an incoming cell is delayed for a fixed period of time, and then forwarded to the output port of the module.

```
module(shift_buffer, [
  ([] :-
     print(ëignored empty message at latchí)),
  (atm_cell(VP, VC, D) :-
    delay(Shifter_delay),
    send(_, cell_out, atm_cell(VP, VC, D)))
]).
```

The Fabric

Finally, we give the translation rules for the switch fabric component itself which is represented by the synthesized template *temp* SF[*int*].

```
def temp SF[M]
  is
    ...
  translations
    with DSL use "generic_fabric(%(M))" where
      rep k from 0 .. M-1 in
        \i<k> is "cell_in%(k))",
        \o<k> is "cell_out%(k))"
      end rep,
      \r is "synch_in"
    end with
  end def;
```

Thus generic_fabric is also a ho-module which is automatically generated by the DSL translator. Recall that $*temp* SF[*int*]$ is a *variable* template, and therefore, when it is evaluated, a *variant* must be specified. For example, when the DSL translator evaluates an exported expression such as \`*temp* SWITCH[4]' or \`*temp* TOP[4]', it will also recursively attempt to evaluate the expression \`*temp* SF[4]' since this is a component of *temp* SWITCH[4]. Interaction with the user can be avoided by specifying the variant *a priori*; for example., by exporting instead the expression

\`*temp* SWITCH[4]@KNOCKOUT[3]',

However, for a nested variable template, such *a-priori* specification requires the user to know the exact order of the recursive evaluation, which can be discovered easily through an *on-the-fly* trial evaluation.

DASE

To facilitate design exploration and all the other capabilities possible with DSL descriptions, a simulation environment called *Design Analysis and Synthesis Environment* (DASE) has been developed. The various components of DASE are shown in figure 4. A DSL interpreter processes ESN descriptions and translates them into a DSL model.

Object-oriented library support features allow for the creation, storage and retrieval of module libraries in an organized manner. Libraries maintain all the information related to a module as well as added information regarding any constraints to be imposed on the modules, any configuration rules to be applied to the components of the library, and the interface specification of the library module. The interface specification is created by the library system to define exactly what ports are available for communication with the library module. The language also allows system and module level constraints to be defined.

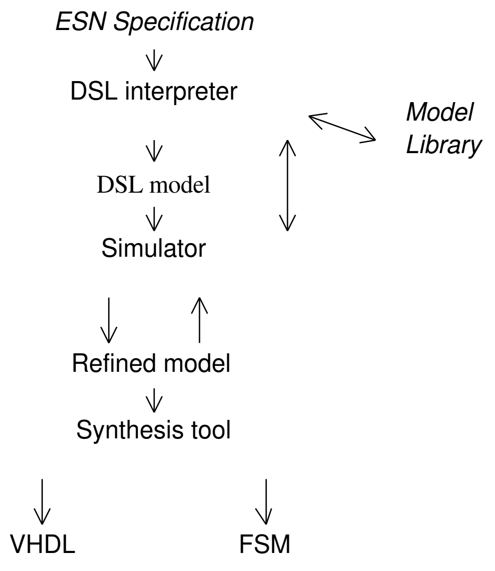

Figure 4. The DASE framework

Constraints define limits upon the structure and behavior of modules. They can be classified as local (dependent upon parameters from one module) and system (dependent upon more than one module) constraints. The former parameters are known beforehand whilst the latter are configuration dependent. For example, the maximum size of a memory module, the minimum delay period for message reception, or the maximum number of calls possible on a switch are represented as local constraints. The size of a cache module, determined through a calculation of the number of processor modules in a multi-processor model is an example of a system constraint and is calculated upon invocation of the whole DSL model.

The simulator uses a DSL model to configure and setup the relevant constraints and resources to support DSL simulation. During simulation, the simulator interacts with the user upon detecting a constraint or simulation violation. The simulator searches for alternate design modules from an existing model base which belong to the same module class and rerun the simulation. The user can also relax the problematic constraint and proceed with the simulation.

When the designer is satisfied with the simulation results, the synthesis stage may be initiated. An analysis methodology also exists whereby the DSL modules can be translated into predicate/transition petri-nets. A synthesis tool within the DASE environment translates the DSL constructs into concurrent entities in VHDL.

CONCLUSION

This paper presented examples of component design for an ATM knockout switch fabric using a reuse approach. The approach gives rise to a framework where structural reuse is employed in conjunction with component reuse. Such a framework permits simulation and modeling of complex systems at a high level of abstraction. An ESN interpreter has recently been implemented at NRC. Currently, work is underway at Bell Canada to interface the ESN interpreter with DSL using an ESN-to-DSL translator. Work is also underway at NRC to integrate the ESN interpreter with a visualization tool.

BIOGRAPHIES

Oryal Tanir is a Senior Consultant and Director of Research within the Quality Engineering and Research organization of Bell Canada in Montreal. He is also and Adjunct Professor at McGill university's Electrical/Computer Engineering department. He has been an employee of Bell Canada since 1986 and holds a Ph.D. and a Master's degree in Electrical Engineering from McGill University in Montreal. He is an active researcher within the discrete even simulation milieu and has published a book and many papers on the subject. He is chairman of the IEEE P1173 working group on simulation as well as member of the ACM, a senior member of SCS, IEEE computer and IEEE communications societies.

Hakan Erdogmus is a research officer with the Institute for Information Technology, National Research Council of Canada, Ottawa. He joined the Software Engineering Group of IIT in 1995. His research activities are centered around formal methods, reuse, software architecture, and design patterns. He holds a doctoral degree in Telecommunications from Institut national de la recherche scientifique of Université du Québec and a Master's in Computer Science from McGill University, Montréal.

REFERENCES

M. Shaw and D. Garlan. Characteristics of higher-level languages for software architecture. Report CMU-CS-94-210, Carnegie Melon University, School of Computer Science, December 1992.

R.T. Monroe and D. Garlan. Style-based reuse for software architectures. In Proc. 1996 International Conf. on Software Reuse, 1996.

R. Biddle and E. Tempero. Understanding the impact of language features on reusability. In Proc. Fourth International Conf. on Software Reuse, Orlando, FL, April 1996.

[O. Tanir and H. Erdogmus. A framework to support structural reuse in simulation environments. In Proc. Of 11th European Simulation Multiconference, SCS, Istanbul, Turkey . June 1-4, 1997a.

H. Erdogmus. A Formal Framework for Software Architectures. Technical Report ERB-1047, National Research Council of Canada, Institute for Information Technology, Ottawa, Canada. December 1995.

O.Tanir. Modeling Complex Computer and Communication Systems: A Domain-Oriented Design Framework. McGraw Hill. 1997.

R.Y. Awdeh and H.T. Mohftah. Survey of ATM switch architectures. Computer Networks and ISDN Systems, 27:1567–1613, 1995.

O. Tanir and H. Erdogmus, Structural reuse in the design of ATM switch fabrics, Proceedings of the world Congress on Systems Simulation, pp. 427-431, SCS, Singapore, September 1-3 1997b.