UC Irvine
ICS Technical Reports

Title
Design visualization and entry using structural and functional entities

Permalink
https://escholarship.org/uc/item/8bx0c1d3

Authors
Sinha, Vivek
Gupta, Rajesh K.

Publication Date
2000-02-16

Peer reviewed

eScholarship.org Powered by the California Digital Library
University of California

Notice: This Material
may be protected
by Copyright Law
(Title 17 U.S.C.)

ICS

TECHNICAL REPORT

Design Visualization and Entry using Structural and Functional Entities

Vivek Sinha, Rajesh K. Gupta

Technical Report Number: #00-05

Information and Computer Science
University of California
Irvine, CA 92697-3245
http://www.ics.uci.edu/~iesag/yaml

February 16, 2000

EXCLUSIVE PROPERTY OF THE
UNIVERSITY OF CALIFORNIA ICS LIBRARY
DO NOT REMOVE FROM PREMISES

Information and Computer Science

University of California, Irvine

Abstract

Visualization of design is an important part of the system design process. In practice, systems are often visualized using a combination of structural and functional entities. In this technical report, we describe YAML (Yet Another UML front end) that enables the system designer to enter designs "schematically" using predefined structural and functional objects conforming to UML notation. YAML provides support for modeling objects and a range of object relationships that are crucial to real-life embedded system designs. A YAML design entry can then be automatically translated into a C++ or synthesizable C++ code for simulation and hardware synthesis.

Contents

1 Introduction **4**

2 System Modeling using Scenic and ICSP **4**
- 2.1 Scenic 5
- 2.2 Modeling Structure: ICSP 6

3 Visual Modeling and Design Entry **6**
- 3.1 YAML 6
- 3.2 Objects and relationships 7
- 3.3 Code generation from YAML 9

4 Implementation and Availability **9**

5 Summary **10**

6 Appendix **12**
- 6.1 Code generated by the UML Front End for the 2 stage pipeline using the Scenic Class Library 12
- 6.2 Code generated by the UML Front End for the 2 stage pipeline using the ICSP class library 15

List of Figures

1 The partitioning of module S1_2 of type Stage1_2 horizontally into two sub-components S1 and S2. Port locations are also shown. 6
2 YAML Menu box 7
3 A Scenic Class Diagram showing the 2 stage pipeline(Figure 1) created using YAML 7
4 UML examples for relationships between classes: (a) generalization (b) composition (c) aggregation (d) association 8
5 Dialog box for the Scenic Module class (Figure 3), showing names and parameters of the subcomponents 10

1 Introduction

A system is usually modeled as a combination of functional and structural entities. Functional entities allow a higher level of abstraction, while structure allows breaking up a large and complex system into smaller parts which can be handled with more ease. It also allows a hierarchical implementation of the system.

Traditionally, the Unified Modeling Language(UML) [1] has been used to conceptualize and model a design, before implementing it in an object oriented language. The software industry has been using UML as the standard for object oriented programming for a long time now. However, extending UML concepts to model hardware[2, 3] is a relatively new area.

Object oriented hardware modeling using C++ is quite popular in the industry these days. Projects like Scenic[4], SystemC[1][5] and Cynlib provide class libraries, which support hardware modeling. However, writing hardware descriptions in C++ using class libraries such as Scenic, can be quite tedious. Design Visualization tools are needed to help a designer conceptualize, model and refine system design without being burdened by the implementation and syntactic details of these C++ class libraries. Towards this end we have built a design entry tool YAML, which uses UML notations to model hardware, and allows the user to input information about the classes and relationships into the UML Diagram itself, rather than first modeling the system in UML and then writing the C++ code for it, as is traditionally the case. YAML generates this C++ code from the information input by the user to the UML Diagram. We note that the embedded system modeling capabilities are not defined by UML, which has limited support for specifying concurrency, timing and placement. In fact, we extend the UML notations to provide support for these hardware specific features[6].

2 System Modeling using Scenic and ICSP

As system complexity increases and design time shrinks, it becomes extremely important that system specification be captured in a form that leads to unambiguous interpretation by the system implementers. Object oriented hardware modeling provides a good alternative[7, 8].

This has driven many system, hardware, and software designers to create executable specifications for their systems. For the most part, these are functional models written in a language like C or C++. These languages are chosen for three reasons: they provide the control and data abstractions necessary to develop compact and efficient system descriptions; most systems contain both hardware and software and for the software, one of these languages is the natural choice; designers are familiar with these languages and the large number of development tools associated with them. However, a programming language like C or C++ is intended for software and does not have the constructs necessary to model timing, concurrency, and reactive behavior, all of which are needed to create accurate models of systems containing both hardware and software[4].

To model concurrency, timing, and reactive behavior, new constructs need to be added to C++. An object oriented programming language like C++ provides the ability to extend the language through classes[9], without adding new syntactic constructs. A class-based approach to providing modeling constructs is superior to a proprietary new language because it allows designers to continue to use the language and tools they are familiar with. These classes add the capabilities needed to model hardware to the C++ language.

[1]SystemC is a publically released version of Scenic by Synopsys Inc. and is available at http://www.systemc.org

2.1 Scenic

To describe hardware in C++, one has to use the building blocks provided in the Scenic[4] class library. The fundamental building block in Scenic is a process. A process is like a C or C++ function that implements behavior. A complete system description consists of multiple concurrent processes. Processes communicate with one another through signals, and explicit clocks are used to order events and synchronize processes. All the building blocks are objects(classes) that are a part of Scenic.

Using the Scenic library, the user can model a system at various levels of abstraction. At the highest level, only the functionality of the system may be modeled. For hardware implementation, models can be written either in a functional style or in a RTL (register-transfer level) style. The software part of a system can be naturally described in C++. Interfaces between software and hardware and between hardware blocks can be described either at the transaction-accurate level or at the cycle-accurate level. Moreover, different parts of the system can be modeled at different levels of abstraction and these models can co-exist during system simulation. C++ and the Scenic classes can be used not only for the development of the system, but also for the test-bench. Scenic consists of a set of header files describing the classes and a link library that contains the simulation kernel. These header files can be used to compile the program. Any ANSI C++ compliant compiler can compile Scenic, together with the program. During linking, the Scenic library, which contains the simulation kernel is used. The resulting executable serves as a simulator for the system described. Scenic has the following features:

- *Processes*: Processes are used to describe functionality. Processes can be stand alone entities or can be contained inside modules. The interfaces for synchronous and asynchronous processes are sc_sync and sc_async respectively. A synchronous process is clocked through the interface, and is invoked on each desired edge. An asynchronous process is invoked according to events on the signal from which the process is sensitive. The behavior is similar to VHDL processes. The processes have an entry() function which is invoked when an event triggers the process.
- *Modules*: Scenic has a notion of a container class called a module. This is a hierarchical entity that can have other modules or processes contained in it. The class for this is sc_module.
- *Signals*: Signals in Scenic are implemented by the sc_signal class. To support modeling at different levels of abstraction, from the functional to RTL, as well as to support software, Scenic supports a rich set of signal types. A clock mechanism implements the time step calculation, and event generation.
- *Channels*: Channels are implemented using the sc_channel class. It enables communication between synchronous processes using handshaking.
- *Clocks*: Scenic has the notion of clocks as special signals. Clocks are the time-keepers of the system during simulation.
- *Cycle-based simulation*: Scenic includes a cycle based simulation kernel that allows high speed simulation.
- *Multiple abstraction levels*: Scenic supports modeling at different levels of abstraction, ranging from high level functional models to detailed RTL models. It supports iterative refinement of high level models into lower levels of abstraction.

2.2 Modeling Structure: ICSP

While Scenic provides the concept of hierarchical containment, missing from Scenic is the concept of relative placement of components, and the attachment or placement of ports with respect to their components. The ICSP[10] class library developed at UCI, is an extension to Scenic to express structural information like component and port placement and layout. The Scenic component interfaces have been specialized to contain the following information:

- *Component order & Placement*: Placement is specified using a slicing tree consisting of horizontal and vertical composition of rectangles. This is implemented in the icsp_module class, which can be used to specify the location of subcomponents, as well as the location of input and output ports. For instance in Figure 1 the module stage1_2 is split horizontally into two subcomponents S1 and S2.
- *Ports Locations*: The location of input and output ports can be specified on the sides of a component. The ports are contained in totally ordered sets, one set for each of the four sides. For instance in Figure 1, if we consider subcomponent S1, the position of input port *in1* is "left upper" while that of output port *diff* is "right lower".

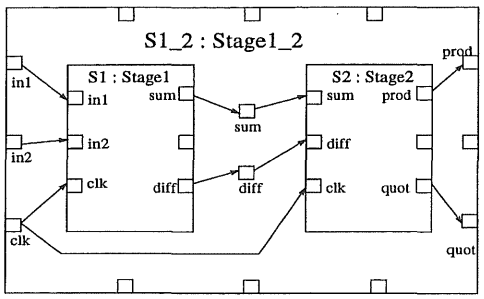

Figure 1: The partitioning of module S1_2 of type Stage1_2 horizontally into two sub-components S1 and S2. Port locations are also shown.

ICSP is useful in building a schematic editor for system level designs, where the user can specify the relative placement of components and ports.

3 Visual Modeling and Design Entry

3.1 YAML

The Unified Modeling Language(UML)[1, 3, 11, 12] is a language for specifying, visualizing, constructing, and documenting the artifacts of systems. It provides a convenient notation that allows the developer to capture structural and behavioral details which can cover both software and hardware characteristics.

As mentioned before, the motivation behind developing YAML is to allow the user to conceptualize and model the design in a graphical interface using UML notation, and add detailed information, which then gets automatically translated to C++ code. This is particularly helpful while using the Scenic and ICSP library of C++ classes, as it allows the user to model the system without being aware of the syntactic details of the underlying C++ code.

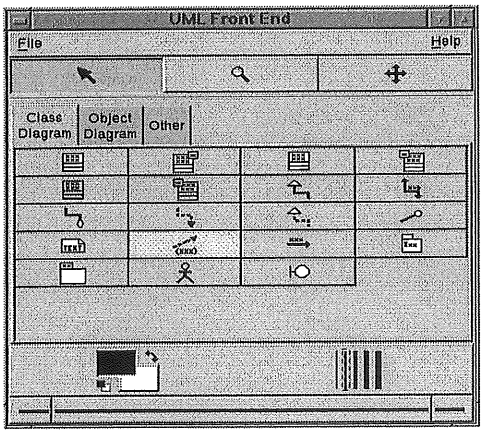

Figure 2: YAML Menu box

YAML allows the user to do a schematic like design entry of the system while conforming to UML notation. It provides support for Scenic and ICSP class libraries, so that the user may use classes from these libraries in the UML Class Diagram of the system. It allows the user to create Scenic processes and modules in the class diagram. The user can create a Scenic class and then add various properties of this class to the class dialog box, such as class name, input and output signals, internal signals and variables, entry function, subcomponents, etc. YAML also provides support for creating ICSP classes, and specifying detailed structure and layout information. The user can specify the type of component partition, either horizontal or vertical, subcomponent order, and the location of the input and output ports around the four sides of the component.

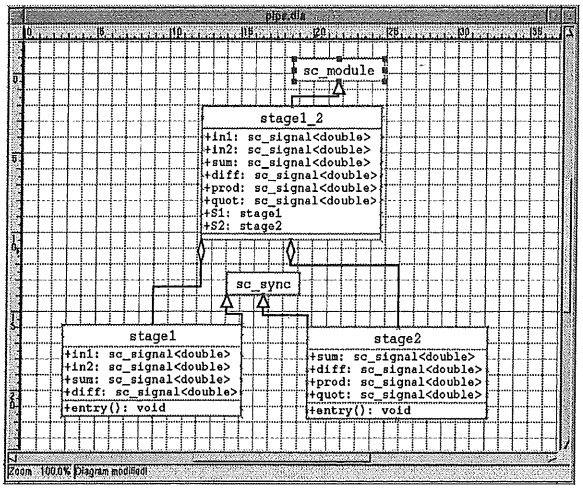

Figure 3: A Scenic Class Diagram showing the 2 stage pipeline(Figure 1) created using YAML

3.2 Objects and relationships

YAML supports system design using objects and relationships. While objects represent function or structure, relationships characterize the interface or interaction among these objects. We have extended the meaning and notation of these objects and relationships so as to support the design of embedded systems[6]. Relationships can be categorized as follows.

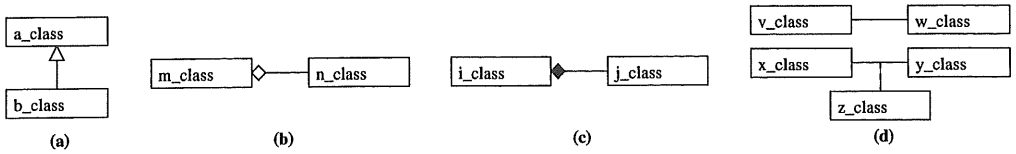

Figure 4: UML examples for relationships between classes: (a) generalization (b) composition (c) aggregation (d) association

1. *Generalization*: Generalization or inheritance is an important concept in object oriented programming. From a base class, we can derive specialized classes, through inheritance. In YAML, generalization is primarily used to derive specialized classes from the set of base classes provided in the Scenic and ICSP libraries. For example, in Figure 3 class stage1_2 is derived from the base class sc_module using the generalization relationship. It is represented by a unidirectional arrow with a triangular head.

2. *Aggregation*: An aggregation relationship applies when one object physically or conceptually contains another. It is represented by an arrow with a diamond shaped head which points towards the owner. The owned class may be shared by more than one owner classes. For example, in Figure 4(b), class m aggregates class n.

3. *Composition*: Composition is a strong form aggregation in which the owner is explicitly responsible for the creation and destruction of the owned objects. It can be represented by the actual inclusion of the owned object in the owner, or by a directed arrow with a diamond shaped filled head pointing towards the owner. Composition corresponds to component hierarchy in a design. In Figure 4(c), class j is a component of class i. This may represent, for instance a CPU datapath, that physically contains an ALU block.

4. *Association:* Association represents conceptual relationships between classes, and is used for the exchange of messages. It corresponds to communication, either through signals or channels. The association relationship can be extended to contain information like communication protocols between objects or even wire delays at a very low level of design. In Figure 4(d), class v has an association relationship with class w. Class z is an association class, which can be used to add details to the association relationship.

YAML provides support for interpreting the relationships between classes, and using this information to generate code. For instance, generalization can be used to create a derived class, and YAML uses this information from the UML Diagram to reflect this derivation in the code generated for the derived class.

We have provided handles in the data structure of a class, so as to access its neighboring classes and get any required information while processing it. The neighbors of a class are all the classes around it in the class diagram that are connected to it by some kind of relationship. This can be particularly helpful in interpreting the composition relationship that exists between a module class and the subcomponents declared in it, where we need to access the subcomponents declared in the Scenic or ICSP module. In short, handles provide a mechanism to traverse the graph of connected classes and access information from any of the nodes of this graph as required.

The association relationship can be extended to contain detailed information about the communication mechanism among objects in an object diagram. Communication protocols, wire delays, interface details are just some of the examples.

3.3 Code generation from YAML

A major drawback of using C++ class libraries for describing hardware is the complex syntactic details to which the classes have to conform. In other words writing code is not easy when these class libraries are used. YAML allows the user to perform graphical entry of the system using Scenic and ICSP class libraries, and processes this information to generate the underlying C++ code.

The option of generating the code is in the class dialog box itself. The user can generate the corresponding Scenic or ICSP code from the class diagram, after specifying the various properties for that class. Following is the code generated for the Scenic class stage1_2 from Figure 3. It is to be noted that for the scenic module class, input and output signals are declared only in the constructor.

```
#include "stage1"
#include "stage2"
struct stage1_2  : public sc_module {
//Internal Signals/Variables
sc_signal<double>  sum;
sc_signal<double>  diff;
//SubComponents
stage1 S1;
stage2 S2;
//Constructor
stage1_2(const char* NAME,
      sc_clock_edge& CLK,
      const sc_signal<double>&  in1,
      const sc_signal<double>&  in2,
      sc_signal<double>&  prod,
      sc_signal<double>&  quot)
 : sc_module(NAME),
S1("stage1", CLK, in1, in2, sum, diff),
S2("stage2", CLK, sum, diff, prod, quot){
   end_module();}};
```

4 Implementation and Availability

YAML has been built using an existing Diagram Editor, DIA[13]. We have used GTK[14] or the Gimp Toolkit to create the graphical user interface. YAML provides a pallet of design entry icons for the user to select from, while designing the system. These include Scenic & ICSP Classes and templates, and relationships like association, aggregation, generalization, composition etc. The UML diagram is created in a new window containing a GTK Canvas widget(Figure 3).

The user next, enters details specific to a class icon. This has been achieved by providing dialog boxes for each of the design entry icons mentioned above. The user can open a dialog box(Figure 5) for the icon he is working on, say a Scenic class, and specify the class name, input and output signals, entry function etc. C++ header and source files for that Scenic class can be generated next, which comply to the Scenic class syntactic details, and are directly compilable. YAML also allows the user to input the entry function for Scenic and ICSP classes into a text window which

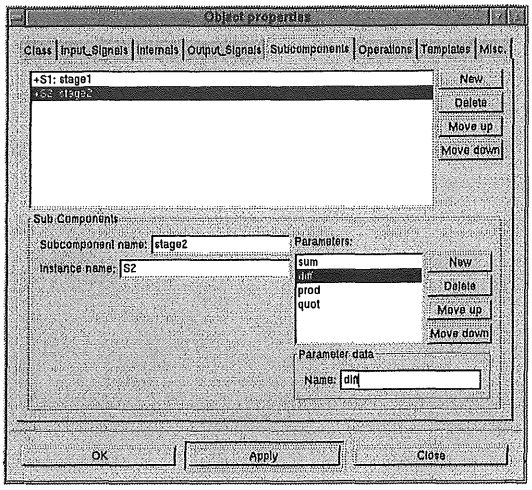

Figure 5: Dialog box for the Scenic Module class (Figure 3), showing names and parameters of the subcomponents

can be opened from their respective dialog boxes. Similarly, the user can also specify the body of the constructor for these classes to initialize the internal variables if required.

YAML is supported on Linux as well as Solaris operating systems and is currently under active development.

5 Summary

YAML provides a user friendly graphical interface to model systems under the guidelines of UML, using the Scenic and ICSP C++ class libraries. The user can specify the details of Scenic and ICSP classes into the UML front end. YAML frees the designer from the syntactic nuances and details corresponding to these class libraries, so that he may focus on the organization of the structural and functional components of his design. The code generated by YAML conforms to the syntax of ICSP and Scenic classes and can be directly compiled or simulated.

Acknowledgments

The authors would like to acknowledge Abhijit Ghosh and Synopsys Inc. for providing us with the Scenic source code and their support for this work.

References

[1] Website: http://www.rational.com/uml/

[2] Hallal H., Xiao-Hua K., Negulescu R.: *Experiments in Modeling Integrated Circuit Blocks by UML;* International Workshop on IP Based Synthesis and System Design'99.

[3] Bruce Powel Douglass: *Real-Time UML: Developing Efficient Objects for Embedded Systems;* Addison-Wesley 1998

[4] Liao S., Tjiang S., Gupta R. K.: *An Efficient Implementation of Reactivity for Modeling Hardware in the Scenic Design Environment;* 34th Design Automation Conference, 1997

[5] SystemC Reference Manual: Release 0.9, Synopsys Inc.

[6] Doucet F. et al: *System-on-chip Modeling using Objects and their Relationships - Technical Report #99-53;* CECS ICS, UC Irvinre October 1999.
http://www.ics.uci.edu/~cecs/

[7] Kumar, S., Aylor, J. H., Johnson, B. W., Wulf, W. A.: *Object-Oriented Techniques in Hardware Design;* Computer, June 1994.

[8] Vernalde, S., Schaumont, P., Bolsens, I.: *An Object Oriented Programming Approach for Hardware Design;* IEEE Computer Society Workshop on VLSI April 1999

[9] Weiler C., Kebschull U., Rosenstiel W.: *C++ Base Classes for Specification, Simulation and Partitioning of a Hardware/Software System;* VLSI 95, Japan.

[10] Siska, C.: *Incidence Structure-Composition Project - Technical Report;* CECS ICS, UC Irvine June 1999. http://www.ics.uci.edu/~cecs/

[11] UML Notation Guide: version 1.1 September 1997

[12] Fowler, M., Kendall, S.: *UML Distilled: Applying the Standard Object Modeling Language;* Addison-Wesley 1997.

[13] Website: http://www.lysator.liu.se/~alla/dia/

[14] Gale, Tony and Main, Ian: GTK v1.2 Tutorial, February 1999

6 Appendix

6.1 Code generated by the UML Front End for the 2 stage pipeline using the Scenic Class Library

stage1.h

```
struct stage1  : public sc_sync {

//Input Signals
const sc_signal<double>&  in1;
const sc_signal<double>&  in2;

//Output Signals
sc_signal<double>&  sum;
sc_signal<double>&  diff;

//Internal Signals/Variables

//Constructor
stage1(const char* NAME,
       sc_clock_edge& CLK,
       const sc_signal<double>&  IN1,
       const sc_signal<double>&  IN2,
       sc_signal<double>&  SUM,
       sc_signal<double>&  DIFF)
 : sc_sync(NAME, CLK),
   in1(IN1),
   in2(IN2),
   sum(SUM),
   diff(DIFF) {
}

void entry();

};
```

stage1.cc

```
#include scenic.h
#include "stage1.h"

void stage1::entry()
{
  double a, b;
```

```
  a = 20.0;
  b = 5.0;
  while (true) {
    sum.write(a+b);
    diff.write(a-b);
    wait();
    a = in1.read();
    b = in2.read();
  }
} // end of entry function
```

stage2.h

```
struct stage2  : public sc_sync {

//Input Signals
const sc_signal<double>&  sum;
const sc_signal<double>&  diff;

//Output Signals
sc_signal<double>&  prod;
sc_signal<double>&  quot;

//Internal Signals/Variables

//Constructor
stage2(const char* NAME,
      sc_clock_edge& CLK,
      const sc_signal<double>&  SUM,
      const sc_signal<double>&  DIFF,
      sc_signal<double>&  PROD,
      sc_signal<double>&  QUOT)
 : sc_sync(NAME, CLK),
   sum(SUM),
   diff(DIFF),
   prod(PROD),
   quot(QUOT) {
}

void entry();

};
```

stage2.cc

```
#include scenic.h
#include "stage2.h"

void stage2::entry()
{
  double a, b;

  a = 20.0;
  b = 5.0;
  while (true) {
    prod.write(a*b);
    quot.write(a/b);
    wait();
    a = sum.read();
    b = diff.read();
  }
} // end of entry function
```

stage1_2.h

```
#include "stage1"
#include "stage2"

struct stage1_2  : public sc_module {

//Internal Signals/Variables
sc_signal<double>  sum;
sc_signal<double>  diff;

//SubComponents
stage1 S1;
stage2 S2;

//Constructor
stage1_2(const char* NAME,
      sc_clock_edge& CLK,
      const sc_signal<double>&  in1,
      const sc_signal<double>&  in2,
      sc_signal<double>&  prod,
      sc_signal<double>&  quot)
 : sc_module(NAME),
S1("stage1", CLK, in1, in2, sum, diff),
```

```
 S2("stage2", CLK, sum, diff, prod, quot){
    end_module();
 }

 };
```

6.2 Code generated by the UML Front End for the 2 stage pipeline using the ICSP class library

Stage1.h

```
struct Stage1  : public icsp_sync {

//Input Signals
const sc_signal<double>&  in1;
const sc_signal<double>&  in2;

//Output Signals
sc_signal<double>&  sum;
sc_signal<double>&  diff;

//Internal Signals/Variables

//Constructor
Stage1(const char* NAME, sc_clock_edge& CLK,
      const sc_signal<double>&  IN1,
      const sc_signal<double>&  IN2,
      sc_signal<double>&  SUM,
      sc_signal<double>&  DIFF)
 : icsp_sync(NAME, CLK,
   icsp_porder(
      icsp_ppair(in1, icsp_e_uleft ),
      icsp_ppair(CLK, icsp_e_lleft ),
      icsp_ppair(in2, icsp_e_mleft ),
      icsp_ppair(sum, icsp_e_uright ),
      icsp_ppair(diff, icsp_e_lright ))),
   in1(IN1),
   in2(IN2),
   sum(SUM),
   diff(DIFF) {
}

void entry();
```

```
};
```

Stage1.cc

```
#include scenic.h
#include "Stage1.h"

void Stage1::entry()
{
  double a, b;

  a = 20.0;
  b = 5.0;
  while (true) {
    sum.write(a+b);
    diff.write(a-b);
    wait();
    a = in1.read();
    b = in2.read();
  }
} // end of entry function
```

Stage2.h

```
struct Stage2  : public icsp_sync {

//Input Signals
const sc_signal<double>&  sum;
const sc_signal<double>&  diff;

//Output Signals
sc_signal<double>&  prod;
sc_signal<double>&  quot;

//Internal Signals/Variables

//Constructor
Stage2(const char* NAME, sc_clock_edge& CLK,
      const sc_signal<double>&  SUM,
      const sc_signal<double>&  DIFF,
      sc_signal<double>&  PROD,
```

```
      sc_signal<double>&  QUOT)
  : icsp_sync(NAME, CLK,
    icsp_porder(
       icsp_ppair(sum, icsp_e_uleft ),
       icsp_ppair(diff, icsp_e_mleft ),
       icsp_ppair(CLK, icsp_e_lleft ),
       icsp_ppair(prod, icsp_e_uright ),
       icsp_ppair(quot, icsp_e_lright ))),
    sum(SUM),
    diff(DIFF),
    prod(PROD),
    quot(QUOT) {
}

void entry();

};
```

Stage2.cc

```
#include scenic.h
#include "Stage2.h"

void Stage2::entry()
{
  double a, b;

  a = 20.0;
  b = 5.0;
  while (true) {
    prod.write(a*b);
    quot.write(a/b);
    wait();
    a = sum.read();
    b = diff.read();
  }
} // end of entry function
```

Stage1_2.h

```
#include "Stage1"
#include "Stage2"
```

```
struct Stage1_2  : public icsp_module {

//Internal Signals/Variables
sc_signal<double>  sum;
sc_signal<double>  diff;


//SubComponents
Stage1 S1;
Stage2 S2;

//Constructor
Stage1_2(const char* NAME, sc_clock_edge& CLK,
      const sc_signal<double>&  in1,
      const sc_signal<double>&  in2,
      sc_signal<double>&  prod,
      sc_signal<double>&  quot)
 : icsp_module(NAME,
    ICSP_SPLIT_HORIZONTALLY,
    icsp_corder(
      icsp_comp( S1 ),
      icsp_comp( S2 )),
    icsp_porder(
      icsp_ppair(CLK, icsp_e_lleft ),
      icsp_ppair(in1, icsp_e_uleft ),
      icsp_ppair(in2, icsp_e_mleft ),
      icsp_ppair(prod, icsp_e_uright ),
      icsp_ppair(quot, icsp_e_lright ))),
   S1("Stage1", CLK, in1, in2, sum, diff),
   S2("Stage2", CLK, sum, diff, prod, quot, ){
      end_module();
   }

};
```