A Higher Level System Communication Model for Object-Oriented Specification and Design of Embedded Systems

Kjetil Svarstad*

SINTEF Telecom and Informatics
Signal Processing and Systems Design group
N-7465 Trondheim, Norway
e-mail: Kjetil.Svarstad@informatics.sintef.no

Nezih Ben-Fredj, Gabriela Nicolescu, Ahmed A. Jerraya
TIMA Laboratory, SLS group
46, Avenue Félix Viallet,
38031 Grenoble CEDEX, France
e-mail: Gabriela.Nicolescu@imag.fr

Abstract— The design starting point for current embedded systems design is getting higher and higher on the abstraction level scale in order to meet the challenge of the increasing design gap. Up til now the state-of-the-art tools and methods have used as a highest abstraction of communication the send-receive over a channel, e.g. as in SDL and COSSAP. We introduce a novel higher level communication mechanism for system-level specification which has features supporting object-oriented descriptions and client-server type communication modelling as in CORBA. The communication primitives have been implemented as extensions to System-C, and simulation experiments have been performed.

I. Introduction

When moving higher up in abstraction for describing embedded systems, we do this for several reasons. In order to shorten development time, to minimize development and production cost, and to maximize product quality, one must: Be able to handle a higher complexity of systems without being drowned in the details. Be able to reuse parts of systems in other products and descriptions on a later stage. And facilitate a description of the system that is close to the actual specification of the system. In terms of the domains the systems are described with regard to, the predominant ones for lending themselves to abstraction is time, behaviour and communication. The *behaviour domain* can be broken down into several subdomains depending on typing possibilies such as type structures and data encapsulation. The *time domain* will move from the continuum of time through discrete time, higher level events and cycles, up all the way to where time is just local ordering, and a global order is at best partial. The conventional view on abstraction levels can be found in e.g. [1]. We will concentrate on the communication abstractions, though, and they are introduced in the following section.

*The author was supported by a grant from the Norwegian Research Council through the Codever progam

The novel approach to communication abstractions presented herein is called *named communication*. It realizes a most abstract way of thinking about the communication, the idea of the connectionless service level. Objects have service access points defined by names, and other objects may call on these services by these names. In terms of description, it closely resembles high level specifications.

In the area of communication and interface synthesis the communication models play an important role. Examples are found in [2], [3], [4], [5], [6], [7], [8], and [9].

II. Abstraction levels in communication

In order to handle complexity of both size and functionality, it is paramount to introduce higher levels of abstractions where a known set of details are abstracted away. Well known in the EDA community is the RTL abstraction being the current focus of state-of-the-art synthesis tools. Synthesis methodology is moving upwards in operational abstractions, though, but we will adresses specifically the communication, and we argue that this is the main problem and difference between the higher system-level abstractions.

In normal circumstances the level of abstraction is directly coupled to the mode of design description. In this paper, though, we are focusing on communication, and the levels of abstraction with regard to communication are, as we will show, slightly different. Also, we present the abstraction layers in a direction of increasing abstraction, a top-down design methodology would of course traverse the levels in the other direction. The reason for such a presentation is moving from the well known and basic facts, through higher abstractions of less succinctness, and finally to our novel approach to specification level abstraction for communication.

The basic properties of these abstraction levels for communication is shown in Table I. Do note that what is labeled *Driver level*, *Message level*, and *Service level* all constitute what is normally denoted the system level. The reason for differentiating them according to communication abstractions is a matter of the specification and de-

Abstraction Level	Communication			Encapsulation	Description	Typical primitive
	Media	Data type	Behaviour			
Service	Type-resolved dynamic net	Universal name spaces + concrete and algebraic datatypes	Routing	Classes (objects), Packages	Specification languages	request(print,device,file)
Message	Active channels with infinite FIFO or mailbox	Concrete generic datatypes	Protocol conversion	Dynamic process blocks	SDL, MSC	Send(data,disk)
Driver	Logical interconnections	Fixed enumerated datatypes	Driver-level protocol	Static process blocks, modules	Cossap, CSP, SpecCharts, System-C 1.1	Write(data,port) Wait until x=y
Register Transfer	Binary signals	Fixed binary data representation	Transmission	Modules, entities	VHDL, System-C 0.9–1.0, Verilog	Set(value,port) Wait(clock)

TABLE I
COMMUNICATION ABSTRACTION LEVELS

sign process. A specification[1] is normally described in terms of the services supplied to the environment, and a break-down of such a description would naturally be the services and their constituting requests in the composition of tasks or processes. After analysing the requirements and their fulfillment, the next step is designing an architecture[2] for the system. Such an architecture will consist of the tasks and processes from the specification, but in more detail. Yet the design and analysis of the communication will be the most untrivial task, since the properties of this communication scheme will severely impact the performance of the whole system. In order to handle this, we have concentrated on the communication mechanisms, primitives and abstractions. To allow reuse, selection and integration of different modules, the communication and the computation will be separated and encapsulated. Through the presented abstraction levels, a system will be modelled as an ensemble of communicating hierarchical modules. Each module is defined by its interface and its content where the interface is composed of a set of ports on which external or internal operations can be performed. The module content may be composed of other module instances, or in the case of the module being a leaf node, it may be composed of tasks or processes. Each abstraction level is defined by specific concepts that encapsulates lower abstraction level concepts, and they are themselves also encapsulated in more abstract concepts.

[1]We are here referencing the *system* specification, not a particular specification on some lower level as the term is normally used in HW-SW codesign. It is the product of the first phase of the over-all *specification-design-implementation* of an embedded system, and forms the basis for all subsequent design specifications and descriptions. In some literature it is denoted as a requirement specification.

[2]This is the over-all system architecture and not to be confused with the modular architectures on the subsequent design abstraction levels.

A. Register transfer level

The RT level deals with the loading of values into registers in modules. Combinatory logic will control the registers, and any address decoding or interrupt management will be explicitly defined and described. Values are represented on compound signals between the modules, and the register to be loaded may be chosen with another compound structure carrying the address for the register.

The main focus for operational abstractions on this level is the handling of the buses, i.e. the compound signal structures, and how the different modules can share the logical and operational space defined by these structures (e.g. shared bus, point-to-point communication, etc.). These abstractions are not real communication abstractions, though, since the RT level demands a fixed representation between data type and bit-vector representation meaning that the size of the data is well known, and the transmission of data is explicit. In the communication domain the abstraction is the same as for the physical level—setting values on physical/logical wires with an immediate[3] reaction with respect to values and time. The RT level communication abstractions are shown in the lowest row in Table I.

Most commercial synthesis tools are still at this level of abstraction. They offer a more unbounded way of describing the inner content of processes—not limited to a specific finite-state-machine description, the communication, however, is still bounded by the same restrictions and lack of abstractions.

[3]In the respect that there is no functional delay caused by any underlying protocol, the only delay is due to the physical character of the wires.

B. Driver level

Typical operational abstractions are master-slave bus-handling defining how the modules' access and privileges regarding the buses will take place. This entails buffered transmission of data, and specific protocols are needed since data have *size*. The system is modeled as interconnected modules communicating through logical connections exchanging fixed, enumerated data types (e.g. integers, reals, etc.) conforming to the driver level protocol. The communication time is non-zero but predictable in the sense that the size and structure of data is well-known, and the choice of data transmission protocol is deterministic.

The implementation of a description in this level of abstraction will typically entail choosing a bus topology and a transaction protocol. Driver level communication is refined in the RT level communication by a synthesis step. Interfaces between the bus and the modules for transforming the protocol transactions into well-known interface commands for the individual modules is the interface synthesis step that can be performed for both simulation and implementation.

System-level synthesis research and a few tools use this level of abstraction for design entry.

C. Message level

At the application modeling level it is useful to be able to describe how processes communicate in the respect of concurrency, synchronization, and channel behaviour. The send/receive model attempts this, and it is built on the semantics of the remote procedure call. The communication will be modeled through active channels capable of interconnecting modules independent of underlying communication protocols. Data are terms, and do not necessarily have a predetermined size, and the communication works solely on the level of such terms. Communication time is thus non-zero and in addition not predictable. It is a simple model, yet it can, by changing the underlying semantics and channel behaviour, describe diverse communication schemes, alas all at the approximately same level of abstraction. Refining active channels into logical interconnections will normally entail some module describing the channel behaviour acting as communication controller.

SDL, the Specification and Description Language standard of the CCITT [10], uses a process and channel modeling basis with the basic send/receive semantics in addition to infinite queueing for channels. It readily models systems of concurrently running and communicating processes without regard to whether the actual process implementation will be in software or hardware or a combination thereof.

Some projects have researched using this level for functional specification and synthesis, e.g [11], but they impose restrictions on the descriptions and communications in particular.

Embedded software methodologies and descriptions such as ROOM [12] also uses a message level abstraction for communication. In [13] it is argued for the elevation of communication abstractions, and in [14] they present several primitives for building communication descriptions upon. These are limited to the message level, though.

D. Service level

The ultimate abstraction level is reached when the communication is seen as the combination of requests and services. A process can request a service from another process, and the underlying protocols, connection structures, and essential timing issues are completely abstracted away. This communication abstraction can support several time models based on the concurrency structure and local time capabilities of the processes themselves. Details of this level are presented in the rest of the paper.

CORBA, the Common Object Request Broker Architecture [15], is a good example of the request-service model in the software domain. Programs or libraries register their services through descriptions in an interface definition language, IDL, and one or more ORBs (Object Request Broker) perform the actual communication routing between a mutual request-service pair.

The four communication abstraction layers are shown conceptually by their abstract architectures in Fig. 1. The service level uses a dynamic routing network accessed by the outgoing request ports, and connects those requests to resolved service ports. All processes and tasks in a module can access the request ports of its encapsulating module. The message level consists of point-to-point channels between modules where the processes can send messages representing data. The channel will perform any conversion according to protocols and connections to other modules. On the driver level, the communication goes through logical buses, and interfaces realises the protocols and stores the data while the protocol instructions are executed. Hence a communication controller is needed to control access to the bus resource and alleviate the protocols. The last one shows a typical RT level architecture where the tasks have been assigned to specific modules. The modules are connected to the bus structures (logical) signals through interfaces where necessary.

III. Specification level communication

When referencing client-server level descriptions, we are using a term from the SW community with certain aspects from large-scale business- and database-application programming. In the context of HW-SW embedded systems,

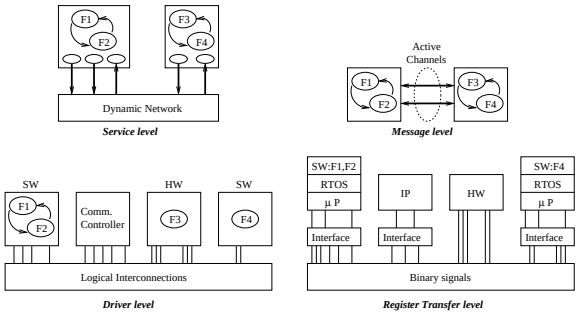

Fig. 1. Abstract architectures of communication

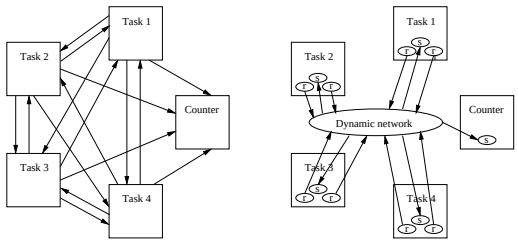

Fig. 2. Explicit versus named communication

we mean something qualitatively different, although it inherits some of the characteristics of the client-server programming techniques. However, for the sake of recognition and comparison, we have used the same term.

A. Object-oriented specification and description

Object-oriented methods have gained a high level of trust in the SW community when it comes to producing SW with high quality, fewer errors, and higher reusability. In Tab. I there is a column for the different encapsulation abstractions. On the RT, and even on the driver level, the main encapsulation is the module. This is a fixed structure which promotes the description of hierarchy, but it lacks the flexibility of the closure-oriented abstraction of classes encapsulating data and methods. On the message level, the blocks of dynamically created processes offer higher abstraction, and SDL in addition offers additional object orientation. Still, the class encapsulation, inheritance, and polymorphism of object-oriented system specifications are only met at the service level.

Classes and their instantiations—objects—are highly abstract compared to classic embedded system descriptions as they feature associations between them that are hard to synthesize both manually and automatically into HW-SW descriptions. The communication model herein described is meant to alleviate this gap through manifesting a unique interpretation of associations between objects as a set of fundamental communication primitives. These primitives are also shown to lend themselves to embedding into simulation semantics such as those for System-C.

B. Named communication

The main motivation for the named communication was the abstraction and thus simplification of describing highly complex communication in embedded systems. Take the example in Fig. 2. The lefthand side shows the topology of a system of four tasks passing tokens between each other, and also to some common counter module. Even with just four tasks, this means 7 explicit connections for sending or receiving tokens to or from the three other tasks plus sending a count signal to the counter. If the system was extended to 10 tasks, that would mean 19 connections for each individual task. For a high level system specification this is not acceptable. The communication in this example is much more likely to be specified and more easily understood using the topology of the righthand side system in Fig 2. All tasks communicate with each other implicitly by naming the services (labeled 's' in the figure) required using request ports (labeled 'r'). The dynamic network will send (route) the requests according to its knowledge of the namespace for

all the defined services. This simplifies the number of connections to three service access points per task, and this number will not change by the number of the tasks in the network.

In order to use such high-level communication abstraction in the specification and design descriptions of embedded systems, there were several requirements to uphold for consistency and ease of use within current methods.

- Ease of matching semantics of specification techniques and descriptions.
- Powerful enough to include client-server like communication.
- Ease of description and embedding in existing execution semantics.
- Simple semantics that can be extended to cover several abstraction levels and communication models.

Since middle-ware like CORBA offers procedure-call like methods of communicating with other objects through a behind-the-scenes object request broker, the choice was to use a request-service like communication primitive that offers a congruent abstraction for system-level specifications and descriptions of embedded systems. We assume a description of concurrent processes where a process may execute a request denoted by a three-tuple $\mathcal{R} = \langle NP, N, P \rangle$. NP is the *named port*, N is the specific *named request*, and P is the parameters of the request. The reciprocate service $\mathcal{S} = \langle NP, N, P \rangle$ will take over the control of the execution, executing some local procedure in the local process, and afterwards returning the control to the requesting process. An additional result of the request may be the service's change of the parameters P' that can be evaluated by the requesting process.

Formally we will define the named communication as such:

The communication space $CS_{n,m}$ is made out of n service groups SG_n and m named ports NP_m. Each service group SG_i is composed of k_i services,

$$SG_i = \left\{ N_1^i, \ldots, N_{k_i}^i \right\}$$

while the ports are just associated with one possible (incoming) service each:

$$NP_j = \left\langle SG_x, N_y^x, \mathcal{T} \right\rangle$$

where $\mathcal{T}$ denotes the type of the port. This means that when the service group, SG_x, has been chosen for the port, a corresponding service from that group, N_y^x, must be fixed for the function of the port. A port with an empty service will be called a request port, while a port associated with a service is a service port. Now we can define a *request* as:

$$\mathcal{R} = \left\langle NP_x, N_y^x, P_{\mathcal{T}} \right\rangle$$

NP_x is a request port, and the request will be for the named service N_y^x of the service group associated with the port NP_x. $P_{\mathcal{T}}$ is the parameter P of type $\mathcal{T}$ that will be furthered onto the requested service.

Conversely, a *service* is defined as:

$$\mathcal{S} = \langle NP_z, \mathcal{F}(P) \rangle$$

$\mathcal{F}(P)$ is the function of the service based on the parameter P of type $\mathcal{T}$. A *request-service resolution* is the pairing:

$$\mathcal{R} : \mathcal{S} = \left\langle SG_{NP_x} = SG_{NP_z}, N_y^x = N_{NP_z} \right\rangle$$

i.e. the signal group of the request and service ports are equal, and the requested service is equal to the type signature of the service port. The resolution ":" signifies a port or process shift, in this case from port NP_x to NP_z, which may be ports of different processes. The *response* is the function performed on the parameters, $\mathcal{F}(P)$. The type resolution behind the ":" forms the semantics of named communication. If resolution is possible between $\mathcal{R}$ and $\mathcal{S}$ in $\mathcal{R} : \mathcal{S}$, then control is passed to the process $\mathcal{S}$, the service procedure executed, and the control passed back to the process $\mathcal{R}$.

As an example, consider the class `Arit` which can perform some arithmetic services for other classes. First we define the service-type:

```
type Arit = add | sub | mult | div
```

And the service class can then be defined thus (the quasi-functional notation used should be self-explanatory):

```
class multiplier (mult) where
    service mult (x,y,_) =
        return (x,y,z) where z = x*y
```

The class is defined with a specific service port since it is of the *mult* value from the *Arit* service type. The service procedure itself, *mult* of type (::) *Arit*, evaluates the two first sub-parameters in the parameter tuple, and then computes the multiplication. It returns the whole tuple since the service is only supposed to change the values in the parameter, not to change the structure of the parameter-set itself.

In some process executing some object of known class, a request for the *mult* service of the *Arit* type would then be:

```
class comp (Arit) where
    ...
    process doArit (a,b) =
        let request (Arit, mult, (a,b,c)) in
            ...
```

This class has an non-specified port of type *Arit*, and that means it is a request port. The *doArit* process evaluates a request for the *mult* service to the *Arit* port upon executing. The behind-the-scenes object request broker then looks up services for the service type *Arit*, finds the

specific instance for the service *mult*, execute the named service (in class *arit*), and then passes back the execution control to the *doArit* process with the altered parameter-set (a, b, c). The new value of c can now be used in the scope of its binding within the process.

It can be argued that the named communication scheme resembles the π-calculus ([16] and [17]). The use of passing names are basic in both, but the π-calculus works on a much more fundamental level of communication, while the named communication are primitives on a more practical level. The π-calculus can only pass names, and all data must therefore be modelled as structures of known names.

IV. Embedding named communication in System-C

System-C [18] is a library-based addition to the programming language C++ where the HW-oriented concepts of signals, processes—both threaded and non-threaded, and modules are defined. In addition, System-C comes with a simulator and is available as free source-code, which makes it very easy to experiment with. This is the reason we chose System-C as the platform for experimenting with named communication.

The concept of named communication is embedded in System-C by defining two template classes, *sc_service* and *sc_request*, which will handle the named communication primitives for service and request. This means that instantiation of these classes will form a service access point of either the service or the request type as illustrated by the labelled ellipsis in Fig 2. The skeleton class definition for the service looks as follows:

```
template <class S, class P>
class sc_service : public sc_named_base {
public:
  sc_service (char * name, char * type,
              S& service);
  virtual void service (P& parameter)=0;
};
```

The template class *sc_service* takes two classes for instantiation where S is the service group type, and P is the corresponding parameter type of the service port. The constructor *sc_service* takes a *name* and a *type* argument of type string to identify the service port in simulations or debuggings. A parameter *service* of type reference to an actual value of type S is needed to establish the specific value of the service port in the service group type. In addition, a virtual method *service* must be defined for the instantiated class with the parameter of type P. This *service* method will be the method called upon to actually service any request.

The *sc_request* template class is very similar. It takes the same two classes S and P as template parameters for instantiating the request port class with service group type S and parameter type P. The constructor does not take a parameter of type S as the *sc_service* constructor. This is due to the fact that a request is not exclusive for any specific value in the service group type S, but can take any value in S in different requests to the same request port. In addition, the *sc_request* class has a predefined method *request* with parameters of type S and P. This is the method that any instance or inherited class of *sc_request* will use to execute a specific request.

```
template <class S, class P>
class sc_request : public sc_named_base {
public:
  sc_request (char * name, char * type);
  bool request (S service, P parameter) {}
};
```

When an *sc_request* or *sc_service* class or any inherited class thereof is instatiated, the constructor will connect the new instance to a global behind-the-scenes object request broker of type *sc_orb*. The ORB will then create a specific instance of class *sc_orb_checker* for every different service group type. This instance will hold a table of all available services of its inherent type. When a request is made to a request port, this calls upon the ORB which will match the type of the request port with one of its *sc_orb_checker*s. The *sc_orb_checker* finds the matching *sc_service* port according to the value of the S parameter, and calls the *service* method in that object instance with the P parameter.

Both the *sc_service* and *sc_request* template classes inherits the *sc_named_base* class which is responsible for the construction-time linking with the ORB, and also connects into the proper System-C classes for being simulated.

The mode of executing the request-service resolution is illustrated in Fig. 3. The dashed arrows are the resolution results.

After the *service* method is completed, the control is returned to the method in the first instance which executed the request, and the execution can succeed with the possibly changed parameters of type P.

V. Modeling and simulation example

In order to test the capabilities and descriptional power of the named communication, we used a small example with simple behaviour in the processes, yet more elaborate communication. The system is comprised of a number of processes sending tokens to each other according to a scheme decided by the type of token and the numerical id of the process. Tokens come in two flavours, direct and indirect. At last there is a common counter that all processes should increment upon receiving a token. This system is the one shown in Fig. 2 for 4 tasks.

We define new datatypes for the two types of services we require: The counter service for incrementing, *Count*, and the Task service for receiving a token, *Task*. The service group type *Count* holds only one possible value, *counter*,

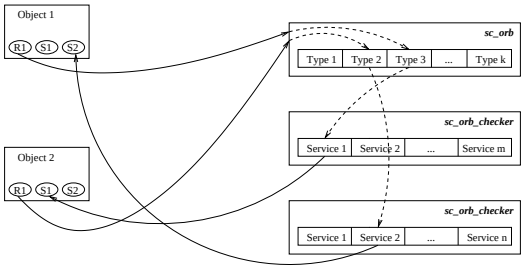

Fig. 3. The request-service processing in System-C

since the counter process is unique for this service, and all the processes will access the same one. The *Task* type is modeled as an integer since the tasks will typically be created in an iterative way where a simple counter can configure the tasks with a unique service id for naming the token receiver service.

The *Token* datatype is used for the type of the tokens, and will be used specifically as a request and service parameter type.

```
typedef enum {counter} Count;
typedef int Task;
typedef enum {direct, indirect} Token;
```

Each process will have a service port to receive tokens. This is the *RxToken* class defined nested within the *Task* below, and it inherits the basic functionality of the *sc_service* class. The service group type is *Task*, i.e. the uniquely identifying integer id, and the parameter type is the *Token* type. The constructor takes a specific value t of type *Task* that will identify this port uniquely.

```
class Task : public sc_sync {

  class RxToken
    : public sc_service<Task, Token> {
  public:
    RxToken (Task & t);
    : sc_service<Task, Token>
    ("RxToken", "RxToken", t) { ... }
    virtual void service (Token & t);
  } * rxToken;

  sc_request<Task, Token> * txToken;
  sc_request<Count, bool> * txInc;

private:
  bool itoken, dtoken;
  int id;
public:
  Task (...)
    : sc_sync ... {
    id = tasknr;
    ... }
  void entry (); };

void Task::RxToken::service (Token & t) {
  if (t == direct)  dtoken = true;
  else if (t == indirect) itoken = true; }
```

The *service* method of parameter type *Token* will be the method called for the defined service, and the function is to check the type of the received tokan t, and to set the local token state accordingly.

Processes having tokens also need to send them out to other processes, or possibly itself. This we facilitate with an *sc_request* port class with the service group type *Task* and the parameter type *Token*. The same procedure is used for the increment request to the common counter resource, the service type, however, is now *Count* and the parameter type bool.

The service port *rxToken* of type *RxToken*, and the request ports *txToken* and *txInc* of types *TxToken* and *TxInc*, respectively, are all instantiated in the class *Task*. This class inherits the System-C specific *sc_sync* class which defines the basic underlying semantics of a synchronous threaded process for the System-C simulation engine. The constructor is passed some unique identifying integer number *tasknr*, which is stored in the local *id* in each *Task* object. This value is used to supply the service name to the *RxToken* service port instantiation.

```
void Task::entry() {
  while(true) {
    wait();
    if (itoken || dtoken) {
      txInc.request (counter, true);
      if (...)
        txToken.request (idest, indirect);
      else if (...)
        txToken.request (ddest, direct);
      rxToken.reset (); } } }
```

When simulating System-C *sync* processes, the simulator will execute their respective *entry* methods. Hence this method must contain all the required behaviour of the process. For the *Task* class, the *entry* method will *wait* until a clock event, then check if there are tokens in the *rxToken* service port, and if so, will send a *txInc* request for the specific *counter* service name. Incidentally, this is the only name in this service group type, and it names the service port of the central counter process. Depending on the value of its *id* and the flavour of its *token*, the process will send a request either to the service port named by

the id *idest* with token value *indirect*, or to the service port named by *ddest* with token value *direct*.

Incrementing the common counter is realized as a service port class *RxInc* of service group type *Count* and parameter type *bool*. The constructor takes as demanded by the *sc_service* class a value *cs* of type *Count* that determines the port's service name. *RxInc* locally stores the increment request in the *inc* signal in the surrounding class. This signal is set to false in the constructor.

The *service* method of *RxInc* sets the *inc* signal to true so the parent *Counter* class can detect the increment request. An additional method *download* is available for the parent process to both check the increment value, and the local value in the service port will then be set to false again assuming the value is directly used or stored in the parent process.

```
class Counter : public sc_sync {

  class RxInc
    : public sc_service<Count,bool> {
  private:
  public:
    RxInc (Count & cs)
      : sc_service ("RxInc", "RxInc", cs) {
      inc_sig false; }
    virtual void service (bool &) {
      inc = true; }
    bool download () {
      bool r = inc;
      inc = false;
      return r; } } * rxInc;

private:
  int val_counter;
  sc_signal<bool> inc;
public:
  Counter (...)
    : sc_sync ... {
    val_counter = 0; }
  void entry ();
};
```

The class *Counter* is another synchronous thread process. It instatiates an object *rxInc* of type *RxInc* for the increment service port with the defined name value of *counter*. It also stores the local counter value in the variable *val_counter*.

The *entry* method of *Counter* realizes its functionality, and that is simply waiting for clock events, checking if there is an increment request in the increment service port *rxInc*, and if such is the case, to increment the value of the counter.

```
void Counter::entry() {
  while (true) {
    wait();
    if(rxInc.download())
      val_counter ++; } }
```

The main procedure of the token passing system is not shown here. It is quite simple in just instantiating the counter, four different tasks, and a system clock. Fig 4 shows the result of one simulation. It is easily seen how the tokens are passed from task to task. After initialisation, the upper left dotted pair of tokens are passed from Task 1 to Task 2 and 4 as shown in the leftmost lower dotted circle. Then one token is passed to Task 3, while Task 4 keeps its token (next circle to the right). Subsequently, after storing the tokens for one cycle, the indirect token is passed from Task 3 to Task 4 while the direct token is stilled kept by Task 4. The token passing continues this way, and it can easily be determined from the simulation results whether the token passing is functionally correct or not.

A. Comparative results

Compared to a behavioural level description of the same system, the named communication based model is 85 source code lines, while the behavioural description is 210 source code lines. The simulation time for 50,000 cycles is halved for the named communication model from 0.22s to 0.11s.

Another model has shown comparative results. This model is a distributed access control system of which the behavioural model is around 500 lines and the named communication based one is 260 lines. Do note that both the named communication based models and the pure behaviour models are all using System-C, the decrease in codesize by the factor 2 3 is due to the named communication abstractions alone.

VI. Conclusion and further work

The named communication abstraction and primitives of request and service realizes a client-server like communication pattern which is independent of any explicitly defined interconnections. It lends itself to powerful abstractions in system-level descriptions which are close to the assumptions and requirements in a system-level specification. The implementation of request-service communication upon the System-C platform shows that it is a viable and useful communication abstraction, and the simulated example shows that the description of a communication intensive system can easily and compactly be described using the named communication primitives.

Further work on named communication will focus on request-service primitives with additional capabilities. An indexing scheme will be added to facilitate the named communication between aggregate objects where a construction time index will be used for name resolution in addition to the service name. Also, a guarded variant of named communication will be researched in order to realize non-deterministic request-service communication using commited guard function resolution. And at last a hierarchical service name type system will be researched for specifically describing dispatch functionality between requests and services. This will increase the flexibility with regard to typing and polymorhism.

References

[1] D. D. Gajski, F. Vahid, S. Narayan, and J. Gong, *Specification and Design of Embedded Systems*. Prentice Hall, 1994.

[2] A. Takura and T. Ohta, "Stepwise refinement of communications service specifications for conforming to a functional-model," *IEICE Transactions on Communications*, vol. E77B, pp. 1322–1331, Nov 1994.

[3] B. Lin and S. Vercauteren, "Synthesis of concurrent system interface modules with automatic protocol conversion generation," in *Proceedings of the International Conference on Computer Aided Design*, IEEE, Nov 1994.

[4] M. Nakamura, Y. Kakuda, and T. Kikuno, "On constructing communication protocols from component-based service specifications," *Computer Communications*, vol. 19, pp. 1200–1215, Dec 1996.

[5] S. Vercauteren and B. Lin, "Hardware/software communication and system integration for embedded architectures," *Design Automation for Embedded Systems*, vol. 2, pp. 359–382, May 1997.

[6] A. Takura, T. Sera, and T. Ohta, "Protocol synthesis from rule-based communications service specifications," *Electronics and Communications in Japan, Part I—Communications*, vol. 81, pp. 22–35, Mar 1998.

[7] J. D. Kleinsmith and D. D. Gajski, "Communication synthesis for reuse," Tech. Rep. ICS 98–06, Department of Information and Computer Science, University of California, Irvine, Feb 1998.

[8] P. Coste, F. Hessel, P. Le Marrec, Z. Sugar, M. Romdhani, R. Suescun, N. Zergainoh, and A. A. Jerraya, "Multilanguage design of heterogeneous systems," in *Proceedings of Seventh International Workshop on Hardware/Software Codesign*, ACM Press, May 1999.

[9] P. Knudsen and J. Madsen, "Integrating communication protocol selectiom with hardware/software codesign," *IEEE Transactions on Computer-Aided Design of Integrated Circuits*, vol. 18, pp. 1077–1095, Aug 1999.

[10] International Telecommunication Union, *CCITT - Specification and Description Language (SDL)*, Mar 1993. Recommendation Z.100.

[11] W. Glunz, T. Kruse, T. Rossel, and D. Monjau, "Integrating SDL and VHDL for system level specification," in *Proceedings of the Conference on Hardware Description Languages*, Apr 1993.

[12] B. Selic, G. Gullekson, and P. T. Ward, *Real-Time Object-Oriented Modeling*. Wiley Professional Computing, Wiley, 1994.

[13] E. A. Lee, "Embedded software—an agenda for research," Tech. Rep. UCB ERL Memorandum M99/63, University of California at Berkeley, Dec 1999.

[14] E. A. Lee and Y. Xiong, "System-level types for component-based design," Tech. Rep. UCB/ERL M00/8, University of California at Berkeley, Feb 2000.

[15] Object Management Group, *CORBAservices: Common Object Services Specification*, Dec 1998. Available at http://www.omg.org/.

[16] R. Milner, J. Parrow, and D. Walker, "A calculus of mobile processes, part I," Tech. Rep. ECS-LCFS-89-85, Computer Science Department, University of Edinburgh, Jun 1989.

[17] R. Milner, J. Parrow, and D. Walker, "A calculus of mobile processes, part II," Tech. Rep. ECS-LCFS-89-86, Computer Science Department, University of Edinburgh, Jun 1989.

[18] Synopsys, CoWare, Frontier Design, *System-C Version 1.0 User Guide*, 2000. Available at http://www.systemc.org/.

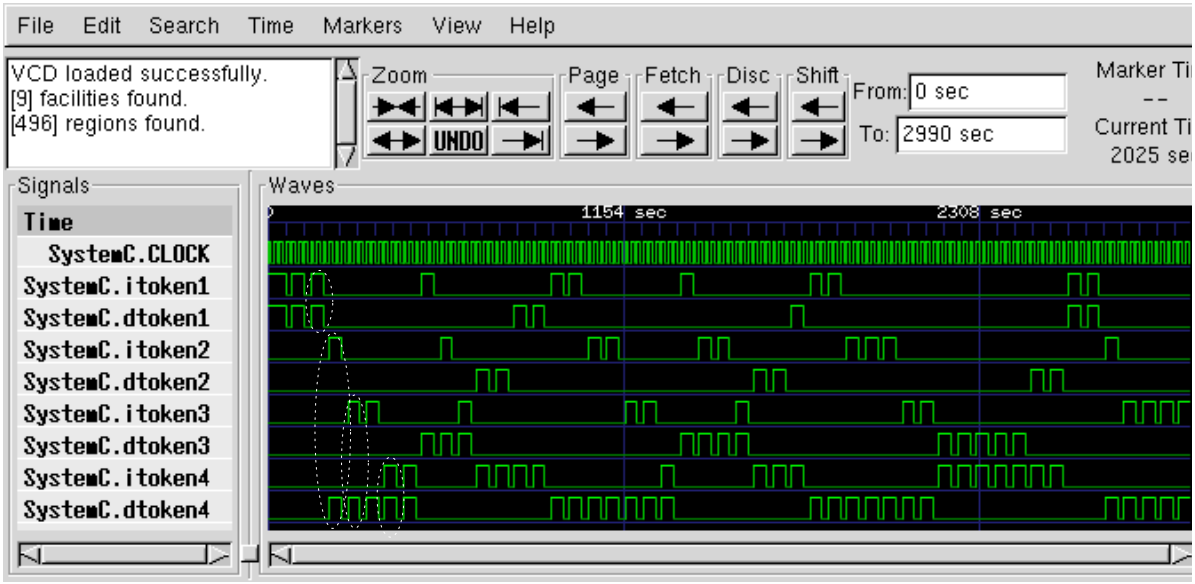

Fig. 4. Simulation results for token passing example