A Model for Describing Communication between Aggregate Objects in the 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

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 elevation of design description abstractions is a well accepted technique for handling the complexity and shortening the design time of modern embedded systems. It is shown that abstractions for communication are as important as for behaviour for specification and system level abstractions, and an extension on a novel higher level communication mechanism which has features for supporting the description of complex aggregate associations between objects in specifications such as UML *is investigated. The communication primitives have been implemented as extensions to SystemC, and a comprehensive example from a* UML *specification through functional specification down to an executable SystemC decription is included.*

1. Introduction

Higher abstraction levels in the description of embedded systems enable the handling of complexity, reuse of descriptions and modules, and also a design starting point which is closer to the initial system specification. These are all important solutions to closing the productivity gap, decreasing development time and cost, and increasing product quality while lowering production cost.

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. We will concentrate on the communication abstractions, though, and they are introduced in the following section.

In [10] we presented a communication model for a higher level system description, especially useful for object-oriented specifications and descriptions. The novel approach to communication abstractions was called *named communication* and 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. This theory and communication primitives has been extended to the *aggregate named communication* presented in this paper, and this is meant to offer the easy mapping and description of complex named associations between aggregate classes and objects such as relations in the standardized object-oriented specification notation UML [9].

In the area of communication and interface synthesis the communication models play an important role. Examples are found in [7], [12], [3], [4], and [2].

2. Abstraction levels in communication

The basic properties of the abstraction levels for communication are shown in Table 1. Do note that what is labeled *Driver level*, *Message level*, and *Service level* all constitute what is normally (e.g. as in [1]) denoted the system level. The reason for differentiating them according to communication abstractions is a matter of the specification and design process—in the specification phase a requirement specification and also a top-level functional specification will typically entail *services* rendered to the user and the system environment. An executable specification at this level should thus enable the modeling of such service and their requests.

The main characteristics of the abstraction levels can be summarized by:

- at the *register transfer level* (RTL), communication

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, SystemC 1.1	Write(data,port) Wait until x=y
Register Transfer	Binary signals	Fixed binary data representation	Transmission	Modules, entities	VHDL, SystemC 0.9–1.0, Verilog	Set(value,port) Wait(clock)

Table 1. Communication abstraction levels

is modeled by physical signals (e.g. shared buses or point-to-point communication) and communication primitives are consequently set/reset of signals and data are instantaneously[1] transmitted in binary representation. At this level interrupt management and address decoding will be explicitly defined. Communication time is based on the clock cycle, and processes correspond to finite state machines where each transition take a clock cycle.

- at the *driver level*, communication is modeled by logical interconnections encapsulating driver level protocols (e.g. hand shake or finite FIFOs). The primitive communication on the modules' ports are the reading and writing of fixed data types in conformity with a certain protocol (e.g. read_hand_shake or write_hand_shake). Communication time is non-zero and predictable due to data having determinate size. The behaviour of basic modules is described by processes realizing calculation steps and communication operations. At this level processes correspond to extended finite state machines (EFSM) where each transition may hide a complex calculation taking several clock cycles.

- at the *message level* the different modules of the system communicate through abstract communication channels (active channels). No assumption about communication implementation is made. Hence, the active channels ensure independent protocol communication of concrete generic data types by providing abstract level communication primitives (e.g. send, receive, put, or get). Such primitives encapsulate all the communication details, and the underlying semantics are based on the remote procedural call (RPC) mechanism. The basic modules' behaviour will be described by tasks communicating by sending and receiving messages. Communication time is non-zero and non-predictable since data are represented as structured terms and have no determinate size.

- at the ultimate abstraction level, the *service level*, the communication is seen as a combination of requests and services. A process can request a service from another (unknown) process, and the underlying protocol, connection structure, and essential timing issues are completely abstracted away. We focus on the modeling of this abstraction level and the necessary and sufficient communication primitives and their semantics.

[1] 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.

3. Specification level communication

In the SW community object-oriented methods have gained a lot of trust because of increased quality and fewer errors in the SW products. Also, the reuse of functionality through SW components are easier when using object-oriented descriptions in conjunction with interfaces to middleware like CORBA and COM. Specification descriptions (or notations) such as UML are now standardized, tools offer a short way from a reasonably abstract class specification down to executable programs. For HW-SW codesign, however, such a process is not that easy. Synthesising HW from the methods of a simple class may be easily solved through behavioural synthesis, or, as shown in [13], an object-oriented description with pre-determined inter-object function calls may be synthesised into a distributed embedded system. However, synthesising from a specification description with several classes and their complex and abstract relationships (or associations), have not been addressed yet.

3.1. Aggregate named communication

The *named communication* theory [10] is useful for describing an executable communication semantic for simple named associations in specifications such as UML. Since all associations must be uniquely instantiated in the communication types, the use will normally be restricted to:

- *one-to-one* class associations with a unique name.
- *many-to-one* class associations with no callback or client dependent functionality.

The last restriction is caused by the fact that a number of classes may use a specific named service, but the named service itself has no way of deciding which of these classes are the actual requesting class. This is too restrictive for most realistic cases such as the one in Fig. 1 which specifies the class relations in a rather simple access control system. Take for example the association between the unique $Central$ class and the unrestricted number of $AccessPoint$ classes. The $check$ relation models the checking of a requesting key or id at the $AccessPoint$, and the access will be granted or not by the $Central$. However, it will be necessary for the $Central$ to know which $AccessPoint$ is actually requesting the access since access rights may be different from point to point. This exact problem will also be the case between the unique common $Timer$ resource and the $Door$ and $Light$ classes. Hence, the aggregate named communication should also include straight forward description of:

- *many-to-one* class associations with possible callback and client dependent behaviour.
- *one-to-many* class associations.
- *many-to-many* class assocations with possible callback and client dependent behaviour.

This is made possible by including in the named requests and services a unique identifying index within each service-request namespace.

Formally we will define the aggregate named communication as such (the definition is an extension of the named communication in [10]):

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,

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

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

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

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 an *indexed request* as:

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

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

Conversely, an *indexed service* is defined as:

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

where $\mathcal{J} \leq \mathcal{K}$ define the index interval for which the service will respond. $\mathcal{F}(P)$ is the function of the service based on the parameter P of type $\mathcal{T}$. A *request-service resolution* is the pairing:

$$\begin{aligned}\mathcal{R}_{\mathcal{I}} : \mathcal{S}_{\mathcal{J},\mathcal{K}} \quad = \quad & \langle SG_{NP_x} = SG_{NP_z}, \\ & N_y^x = N_{NP_z}, \\ & \mathcal{J} \leq \mathcal{I} \leq \mathcal{K}\rangle\end{aligned}$$

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, and the requested index is in the index interval defined for the service. 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 $\mathcal{R}$.

Named communication resembles the π-calculus [8] in the basic passing of names. The π-calculus works on a much more fundamental level of communication, however, while the named communication constitutes primitives on a more practical and descriptive level. Lee et. al. argue in [5] and [6] for the use of higher communication abstractions in the description of embedded systems, but limit them to the message level.

3.2. Aggregate named communication in specification

In order to illustrate the principle of named communication, consider the example of an access control system like

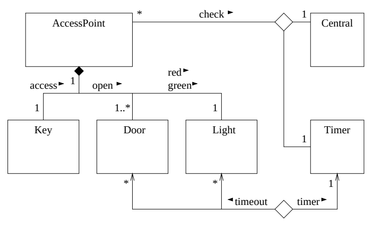


Figure 1. UML class specification of access control system

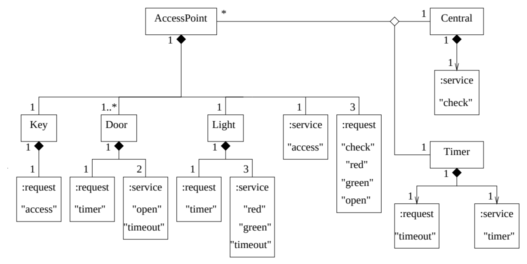


Figure 2. Service level UML specification of access control system

the one depicted in Fig. 1. As already described, this UML-description can be mapped into executable code in several different ways depending on the interpretation of the relations between classes. In order to generate for example a Java or C++ executable, the named associations must be mapped upon deterministic member function calls. The aggregate mode of *named communication*, on the other hand, allows the use of simple type definitions instead of deterministic function calls, and the functions to be called (services) are resolved at run time and not at compile time. For the access control example we use the following type definitions to capture the essential service level behaviour of the system:

```
type KeyOp = access
type AccessOp = check
type DoorOp = open | close
type LightOp = green | red | off
type TimerOp = timer | timeout
```

The type $KeyOp$ represents the service offered to some reader of the key cards, while the $AccessOp$ represents the service of actually checking the access according to the id of the key. $DoorOp$ and $LightOp$ directly models the possible states of the lock and the lightpanel, respectively. Two timer services are also included in the type $TimerOp$, namely $timer$ for the actual timer service and $timeout$ for the call-back services that the timer should acknowledge upon a specific timeout. Given these types the system modelled with services and requests will look like Fig. 2. The respective service and request ports of the classes are set according to a simple specification of the system.

For every named association in Fig. 1 there will be a request port at the source and a service port at the destination. For example, in Fig. 2, the "red" and "$green$" associations have the class $AccessPoint$ as source and $Light$ as destination. The first level of communication synthesis establishes a "red" and "$green$" request port for $AccessPoint$, and two individual "red" and "$green$" service ports for the $Light$ class. The "red" and "$green$" requests are both implemented by one single request port of type $LightOp$ since request ports span over all possible values of the corresponding type.

The following is a highly functional specification of the Door class. Although undefined and not yet executable, this functional description should be easy to understand.

```
class Door (open, TimerOp, timeout) where
```

```
    sc_int id, sc_signal<DoorOp> lock
    service open id _ =
        lock (open)
        request timer id (3000, open)
    service timeout id open = lock (close)
```

The *Door* class has local state variables for an index *id* and the *lock* state. The first declared service of type *open* responds to any *open* request with the appropriate *id*, and the result is opening the *lock* and requesting a *timer* event for its specific id. The latter service is for the callback of the *timeout* event, and the *lock* is then closed.

The *Light* class is similar to the *Door* class with its two services for any id-specific requests for green or red light. Both services also request timer events after setting the light. The *timeout* event service then turns off the lights.

```
class Light (green, red, TimerOp, timeout) where
    int id, LightOp light
    service green id True =
        light (green)
        request timer id (3000, green)
    service red id True =
        light (red)
        request timer id (3000, red)
    service timeout id _ = light (off)
```

In the *Timer* class there is a service for the *timer* requests which queues the requests with their preferred message parameter and the calling id. When the requested time arrives, a *timeout* request is made with the stored id and message parameter.

```
class Timer (TimerOp, timer) where
    TimeQ timeQ = makeQ
    service timer 0 63 id (time, para) =
        enterQ timeQ (time, id, para)
    process checkQ (askQ timeQ) =
        let (t, i, p) = getQ timeQ in
            if p then request timeout i p
```

When a key is inserted in the reader, a request is made for whether access will be granted or not.

```
class Key (KeyOp) where
    int id
    process checkKey (keyNr) =
        request access id keyNr
```

The *AccessPoint* class receives *access* requests, and check these by a request to the *AccessOp* request port. If the access is granted, a request for *open* the lock and turn the light *green* is made, while denied access requests a *red* light.

```
class AccessPoint (AccessOp, DoorOp,
                   LightOp, access) where
    int id
    service access id key =
        if request check id key then
           request open id ()
           request green id True
        else
           request red True
```

The common *Central* class services the *check* requests and checks whether the key id is valid.

```
class Central (check) where
    service check id =
        if valid id then return True
        else return False
```

This was the functional specification of the Access Control system. At the time being this is an informal description, we have no simulator for this kind of description. However, the specification is easily transformed into a description based on aggregate named communication primitives implemented in SystemC as we shall see in the next section.

4. Modeling and simulation example

In order to simulate systems described with aggregate named communication, we implemented some classes for the respective request and service ports under SystemC. These classes can be used to define specific classes for the service types in the application along with SystemC (see [11]) descriptions of the behaviour itself. The requests and services are connected to a behind-the-scenes *object request broker* (ORB) which takes care of the run-time resolution of requests to services. The classes will be shown in the following example of the Access Control system.

We define types for the request and service names like in the specification thus:

```
typedef enum {open, close} DoorOp;
typedef enum {red, green, off} LightOp;
typedef enum {timer, timeout} TimerOp;
typedef enum {card} KeyOp;
typedef enum {check} AccessOp;
```

This is exactly the same types as in the functional specification. The similarity between specification and the aggregate named communication model is remarkably high for the class declarations also. Take the *Door* class below. The *OpenDoor* local class definition in *Door* inherits the *sc_iservice* class which is predefined for the underlying semantics of a single index service. The template parameters *DoorOp* and *void* are the service type and the parameter type, respectively. The *sc_iservice* constructor which must be called for any inheriting classes needs parameters for defining the value of the service type (in this case *open*) and the value of the single index. Any service specific behaviour must be defined in the virtual member function *iservice* which receives as parameters the index of the requester and also the parameter itself (in this case *void*). Likewise, the *CloseDoor* class also inherits a predefined service class. However, the *TimeOut* class is predefined to be of service type *Timer* and of the specific value *timeout*, and the parameter of type void (see below for definition of *TimeOut* class). As can be seen, the typing system is more restricted for the C++-based SystemC description than for the functional-language based specification. We can not send along the preferred return-parameter to the *Timer* since template parameters must be static and known at compile time. This fact makes the SystemC based description a little bit more complex than the functional specification, and in addition the C++-based language is much more verbose than the specification.

```
class Door {

  class OpenDoor :
   public sc_iservice<DoorOp, void> {
    OpenDoor (char *, char *, int);
    virtual bool iservice (void);
  } * openDoor;

  class CloseDoor :
   public Timeout {
    CloseDoor (int, Door *);
    virtual bool iservice (int);
  } * closeDoor;

  TimerRequest * timer;
  bool Open ();
  bool Close ();
  ...};
```

The *TimerRequest* class that is instantiated in the *timer* pointer is defined below, and it inherits the *sc_irequest* class which defined the underlying semantics of an indexed request port. There is no need to define a local class since request classes have no virtual functions to be defined in order to work. They can readily be used as is.

Now we can define the local services as implemented in the local service classes' *iservice* member function. The *OpenDoor* service calls a defined *Open* member function in the encapsulating *Door* class, and the *CloseDoor* class the inversely equivalent *Close* function.

```
bool Door::OpenDoor::iservice (void) {
  return Open (); }

bool Door::CloseDoor::iservice (int code) {
  if ((DoorOp) code == close) {
    return Close (); }
  else return false; }

bool Door::Open () {
  if (door) return true;
  else {
    Timer::dint p = {3, (int) open};
    door = true;
    return timer->irequest (timer, id, d); } }

bool Door::Close () {
  door = false;
  return true; }
```

The use of a *Timer* request is seen in the *Open* member function of the *Door* class. It is exactly equivalent to the functional specification, the verbosity is due to the SystemC base C++ as mentioned.

```
class Light {

  class RedLight :
   public sc_iservice<LightOp, void> {
    RedLight (char *, char *, int);
    virtual bool iservice (void);
  } * redLight;

  class GreenLight :
   public sc_iservice<LightOp, void> {
    GreenLight (char *, char *, int);
    virtual bool iservice (void);
  } * greenLight;

  class OffLight : public Timeout {
    OffLight (int);
    virtual bool iservice (int);
  } * offLight;

  TimerRequest * timer;
  bool Set (LightOp);
 private:
  ...};
```

The *Light* class uses service and request classes for ports in exactly the same way as the *Door* class, only there are three service ports of type *LightOp* to capture the *green*, *red*, and *off* light services. Hence we show no details of the member functions, the principle is the same as for the *Door* class.

```
class TimerRequest :
 public sc_irequest<TimerOp, Timer::dint> {
  TimerRequest (int); }

class Timeout :
 public sc_iservice<TimerOp, int> {
  Timeout (int); }
```

TimerRequest and *TimeOut* as expected by the *Door* and *Light* classes are an indexed request and an indexed service class, respectively. They are both of the *TimerOp* type.

```
class Timer {

  typedef struct {int time, code} dint;

  class TimerS :
   public sc_iservices<TimerOp, dint> {
    TimerS (int, int);
    virtual bool iservices (int, dint);
  } * ts;
  TimerRequest * tr;
  ... };
```

Since the common *Timer* needs to service an aggregate number of Doors and Lights, it uses a local service class inherited from the *sc_iservices* class. This class uses an index interval in the definition, and the underlying semantics is that its *iservices* member class will be called for all requests to the proper type and value with an index in the service interval. The *iservices* member will at call time know the requester's index, and can use this for eventual callback functions if appropriate. For example, below is the *Timer* service port function. It adds the timer value to a queue along with the requester's index. When the timeout is reached as tested by the *Tick* function, a request is made to the assumed *TimeOut* port of the original requesting process.

```
Timer::TimerS::iservices (int ix, Timer::dint p) {
  addTQ (p.time, p.code, ix); }

void Timer::Tick () {
  while (t = chkTQ (t))
    if (tr->irequest (timeout, t->index, t->code))
      rmTQ (t); }
```

The *Key* class is simply a frontend class for calling the simulation model and test it out. The *Card* member function requests entry by the *KeyOp* type with the *access* value.

```
class Key {
  Key (int);
  ~Key ();
  bool Card (int);
 private:
  sc_irequest<KeyOp, int> * keyRequest;
  int id; };

bool Key::Card (int code) {
  return keyRequest->request (access, id, code); }
```

The *AccessPoint* class has one service *CardServer* for the *KeyOp* type and *access* value. On incoming requests on this service, an access request will be made on the *accessReq* port. If the result is an access grant, then an *open* door and a *green* light request is made, else a *red* light request is made. This is implemented in the *Check* member function below.

```
class AccessPoint {

  class CardServer :
   public sc_iservice<KeyOp, int> {
    CardServer (int);
    virtual bool iservice (int);
  } * cardServer;

  sc_irequest<DoorOp, void> * doorReq;
  sc_irequest<LightOp, void> * lightReq;
  sc_irequest<AccessOp, int> * accessReq;

  AccessPoint (int);
  ~AccessPoint ();
  bool Check (int);
 private:
  int id; }

bool AccessPoint::CardServer::iservice
 (int cid) {
  return Check (cid); }

bool AccessPoint::Check (int cid) {
  if (accessReq->request (check, id, cid))
    return doorReq->request (open, id, 0)
        && lightReq->request (green, id, 0);
  else
    return lightReq->request (red, id, 0); }
```

Model	Specification	Named communication model	Behavioural model
Token ring system	20	85	210
Access control system	50	260	500[2]

Table 2. Size of models using different abstractions

Access requests are services by the *CheckCard* service of the *Central* class. This is an interval service of the *AccessOp* type since all possible aggregated indexes must be taken into account. For simplicity, we grant all keys with id from 1 to 5 access, and all others are denied.

```
class Central {

  class CheckCard :
   public sc_iservices<AccessOp, int> {
    CheckCard (int, int);
    bool iservices (int, int);
  } * checkCard;

  Central (int, int);
  ~Central ();
  bool Access (int, int); };

bool Central::CheckCard::iservices
 (int ix, int card) {
  return Access (ix, card); }

Central::Access (int door, int card) {
  if (card > 0 and card < 6) return true;
  else return false; }
```

Fig. 3 shows the results of a simple simulation run with three different access points. At time 5 a key with id 3 is requesting access at point 2. This is granted, and door 2 is opened and light 2 set to green. After 3 seconds the door is closed and the light turned off as expected from the timer request of 3000 milliseconds. Overlapping this at time 7 a key with id 4 is requesting access at point 3. This is also granted, and door 3 opened and light 3 set to green. At time 12, however, a key with id 7 is requesting access at point 1. This is not granted, and the door stays closed while the red light is lighted for 3 seconds. The expected behaviour has been validated.

[2]Estimated from incomplete model

4.1. Comparative results

Table 2 sums up the results from the two models tested at the named communication level of description. The Token Ring system was described in [10]. As can be seen, the SystemC based description with named communication is 2–3 times larger than the functional description. This is a consequence of the restrictions on template typing in C++, and also the fact that C++ offers lower abstractions on functionality than the functional-language-based specification. Also, C++ is much more verbose with lots of redundant redeclarations of class members in and out of class declaration scope because of the need to include class declarations with each other.

From the named communication based description and to a straight forward behavioural description there is typically a 3–4 times increase in the size of the models. We think this a prominent side-effect of the higher level of communication abstractions. In addition, the simulations have uniformly shown a doubling in execution time when moving from named communication to the behavioural level—even though the request-service resolution takes place at run-time and not at compile time which normally penalizes execution time. Yet the compactness of the communication more than evens this out.

5. 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 SystemC platform shows that it is a viable and useful communication abstraction, and the simulated examples show that the description of communication intensive systems 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. A guarded variant of named communication will be researched in order to realize non-deterministic request-service communication using commited guard function resolution. And a hierarchical service name type system will be investigated for specifically describing dispatch functionality between requests and services. This will increase the flexibility with regard to typing and polymorhism. Also, it will be important to show how communication synthesis can take place from named communication abstractions.

Acknowledgments— The research was partly financed by a grant from the Norwegian Research Council through the Codever project.

References

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

[2] M. Gasteier and M. Glesner. Bus-based communication synthesis on system level. *ACM Transactions on Design Automation of Electronic Systems*, 4(1):1–11, Jan 1999.

[3] J. D. Kleinsmith and D. D. Gajski. Communication synthesis for reuse. Technical Report ICS 98–06, Department of Information and Computer Science, University of California, Irvine, Feb 1998.

[4] P. Knudsen and J. Madsen. Integrating communication protocol selection with hardware/software codesign. *IEEE Transactions on Computer-Aided Design of Integrated Circuits*, 18(8):1077–1095, Aug 1999.

[5] E. A. Lee. Embedded software—an agenda for research. Technical Report UCB ERL Memorandum M99/63, University of California at Berkeley, Dec 1999.

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

[7] 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.

[8] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, part I and II. Technical Report ECS-LCFS-89-85 and -86, Computer Science Department, University of Edinburgh, Jun 1989.

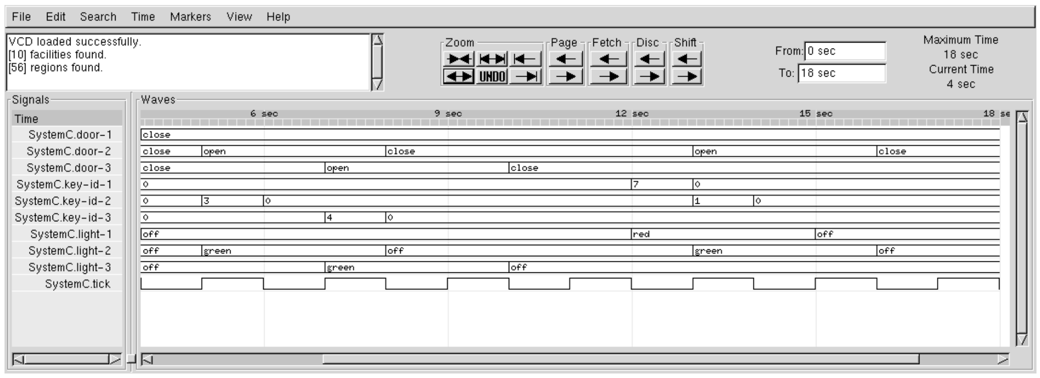
[9] Object Management Group. *OMG Unified Modeling Language Specification*, Jun 1999. Available at http://www.omg.org/.

[10] K. Svarstad, N. Ben-Fredj, G. Nicolescu, and A. A. Jerraya. A higher level system communication model for object-oriented specification and design of embedded systems. In *Proceedings of ASP-DAC 2001*, Jan 2001.

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

[12] S. Vercauteren and B. Lin. Hardware/software communication and system integration for embedded architectures. *Design Automation for Embedded Systems*, 2(3–4):359–382, May 1997.

[13] W. Wolf. Object-oriented cosynthesis of distributed embedded systems. *ACM Transactions on Design Automation of Electronic Systems*, 1(3):301–314, Jul 1996.



```
VCD loaded successfully.
[10] facilities found.
[56] regions found.
```

From: 0 sec
To: 18 sec
Maximum Time 18 sec
Current Time 4 sec

Figure 3. Simulation of access control system