Available online at www.sciencedirect.com
SCIENCE DIRECT
Information and Software Technology 48 (2006) 221–234
INFORMATION AND SOFTWARE TECHNOLOGY
www.elsevier.com/locate/infsof

DEVSpecL: DEVS specification language for modeling, simulation and analysis of discrete event systems

Ki Jung Hong*, Tag Gon Kim

EECS Department, Korea Advanced Institute of Science and Technology, 373-1, Guseong-dong Yuseong-gu, Daejeon 305-701, South Korea

Received 22 July 2004; revised 9 April 2005; accepted 13 April 2005
Available online 19 July 2005

Abstract

Discrete EVent Systems Specification (DEVS) formalism supports specification of discrete event models in a hierarchical modular manner. This paper proposes a DEVS modeling language called DEVS Specification Language (DEVSpecL) based on which discrete event systems are modeled, simulated and analyzed within a DEVS-based framework for seamless systems design. Models specified in DEVSpecL can be translated in different forms of codes by code generators, which are executed with various tools for models verification, logical analysis, performance evaluation, and others.

Keywords: Discrete event system; DEVS formalism; DEVS specification language; Model testing; Seamless design framework

1. Introduction

Recently, discrete event systems, such as computer/communication networks and manufacturing systems, are getting more and more complex due to a variety of requirements which include desired functions/properties, required performance, real-time constraints and others. Accordingly, design of such systems has become much complicated, which is an iterative process of modeling, logical analysis and performance evaluation to check satisfaction of the requirements. Moreover, a complete development of such a system requires an additional process of implementation and run-time testing. Thus, an efficient development may need a systematic method and associated computerized tools, which support the overall development process. To do so requires a unified formal modeling framework based on which all activities in the overall process can be performed in a seamless manner. To the authors' best knowledge no such framework and supporting computer-aided tools are in place in the discrete event system area. However, some methods and tools were reported, which do not support the overall process but support sub-processes of the overall development process. The main problem to apply such methods is that a model used in one design phases may not be used in the other phase. For example, a model which is used for the logical analysis phase is not be applied to the performance evaluation phase.

Over the years, communication network research proposed discrete event system modeling languages for protocol engineering, such as SDL [14], ESTELLE [12], and LOTOS [13]. SDL, ESTELLE are LOTOS are all protocol description languages which are based on finite state machine (FSM), process algebra, and CCS [18], respectively. The main objective of the languages is to describe protocol based on which functional/logical analysis is performed without consideration of performance evaluation within the description. Since they are languages an implementation of a described protocol can be automatically obtained through a language translation process. However, testing of the implementation is another process, which may be done by a method of conformance testing [2]. For performance evaluation attempts have been made to extend the languages to timed SDL [3], timed LOTOS and timed ESTELLE in which semantics for time delay between state transitions was defined.

For discrete event systems modeling/simulation the DEVS (Discrete Event Systems Specification) formalism has been widely used for last twenty-five years. The DEVS

* Corresponding author. Tel.: +82 42 869 5454; fax: +82 42 869 8054.
E-mail address: kjhong@smslab.kaist.ac.kr (K.J. Hong).

0950-5849/$ - see front matter © 2005 Elsevier Ltd All rights reserved.
doi:10.1016/j.infsof.2005.04.008

formalism supports specification of discrete event models as a form of a timed state transition system in a hierarchical modular manner [20]. Originally, the main use of the formalism is to build DEVS models for performance evaluation. However, recent research on DEVS theory has been extended to such areas as logical analysis [7], real-time software development [8], and discrete event systems verification [9]. More recently, a unified DEVS framework for development of discrete event systems has been proposed in which modeling, logical analysis, performance evaluation and virtual prototyping of discrete event systems can be all done by the DEVS formalism [16]. However, the framework did not propose a method for testing of implementation. To implement the framework as a compute-aided environment, first of all, needs a DEVS specification language from which other tools are to be developed and/or linked to support the overall development process.

Some DEVS specification languages has actually been defined and implemented. In [7], a DEVS specification language has been defined and implemented for behavioral analysis of discrete event models without time information. Another specification language, called openDEVS, was defined in [19] which has three characteristics: preservation of the DEVS models information, object-oriented modeling, and model type-check. However, the language was not implemented. However, openDEVS has not been implemented. The purpose of this paper is to define and implement a specification language called DEVS Specification Language (DEVSpecL), which is an extension of the two specification languages. More importantly, DEVSpecL would be a basis for development of a set of automated tools for the seamless development methodology proposed in [16]. Thus, DEVSpecL would be applied for all phases of the development process ranging from modeling, analysis and implementation to verification of implemented code.

This paper is organized as follows. Section 2 presents a proposed design framework with the DEVSpecL language and introduces the DEVS formalism. Section 3 describes the DEVSpecL language with an example of alternate bit protocol. Section 4 presents applications of DEVSpecL. Finally, conclusion is made in Section 5.

2. DEVS-based system design process

2.1. Overview of design process

A unified framework for discrete event systems design should seamlessly embrace various activities as shown in Fig. 1. Logical analysis is to verify functional properties and constraints of a system to be designed. The verification with the framework checks a lower-level operational specification against a higher level assertional specification. Specifications for the two levels may employ either one formalism in a single language approach [16] or different formalisms in a dual language approach [7]. Performance evaluation within the framework employs object oriented simulation of hierarchical DEVS models for which simulation environments running on different languages are developed. Such environments include DEVSim++ [17] of a C++-based environment and DEVSimjava of a java based environment. Implementation within the framework may be done by two ways depending on an existence of real-time constraints. One way is to implement with real-time constraints, i.e. a DEVS model can be directly executed, without any modification, into a real-time simulation engine. Such executable DEVS model is specified by the real-time DEVS formalism [1,8]. Other way is to implement without real-time constraints, i.e. each model is scheduled by its time information. Scheduled time is synchronized to a wall clock.

Verification of an implementation is a final step of the design process, which we view as a software testing phase. Software test is mainly classified in three approaches based on degree of knowledge about internal information of the implementation under test: black-box test, white-box test and gray-box test, [5]. In the black-box approach, testing is carried out with unknown information about internal structure and state of an implementation [4,10]. In white-box testing, the whole internal information of an implementation, i.e. state and structure information, is known and the test suite is generated from the known structure. Finally, the gray-box approach is mixed with the white-box and the black-box approaches which may be more efficient than application of one of the two separately. Testing complexity of the above three depends on both test coverage and the number of states in the test target. To reduce such complexity, the gray-box approach is adopted in the proposed design framework with an assumption that an upper bound of the number of state in the implementations is known.

2.2. DEVS formalism

The DEVS formalism specifies a model in a hierarchical, modular form. A discrete event system consists of entities whose dynamics are described as a set of procedure rules. Such rules control the interactions among the communicating entities. The communicating entities and the procedure rules can be decomposed into the smaller ones with the modeling semantics. These decomposed components are directly mapped to the atomic models, from which larger ones are built. A basic model, called an atomic model, is not further decomposed into smaller ones. Formally, an atomic model AM is specified as [20]:

$$\mathrm{AM} = \langle X, Y, S, \delta_{\mathrm{int}}, \delta_{\mathrm{ext}}, \lambda, \mathrm{ta} \rangle$$

X, input events set
Y, output events set
S, states set
δ_{int}: $S \rightarrow S$, internal state transition function

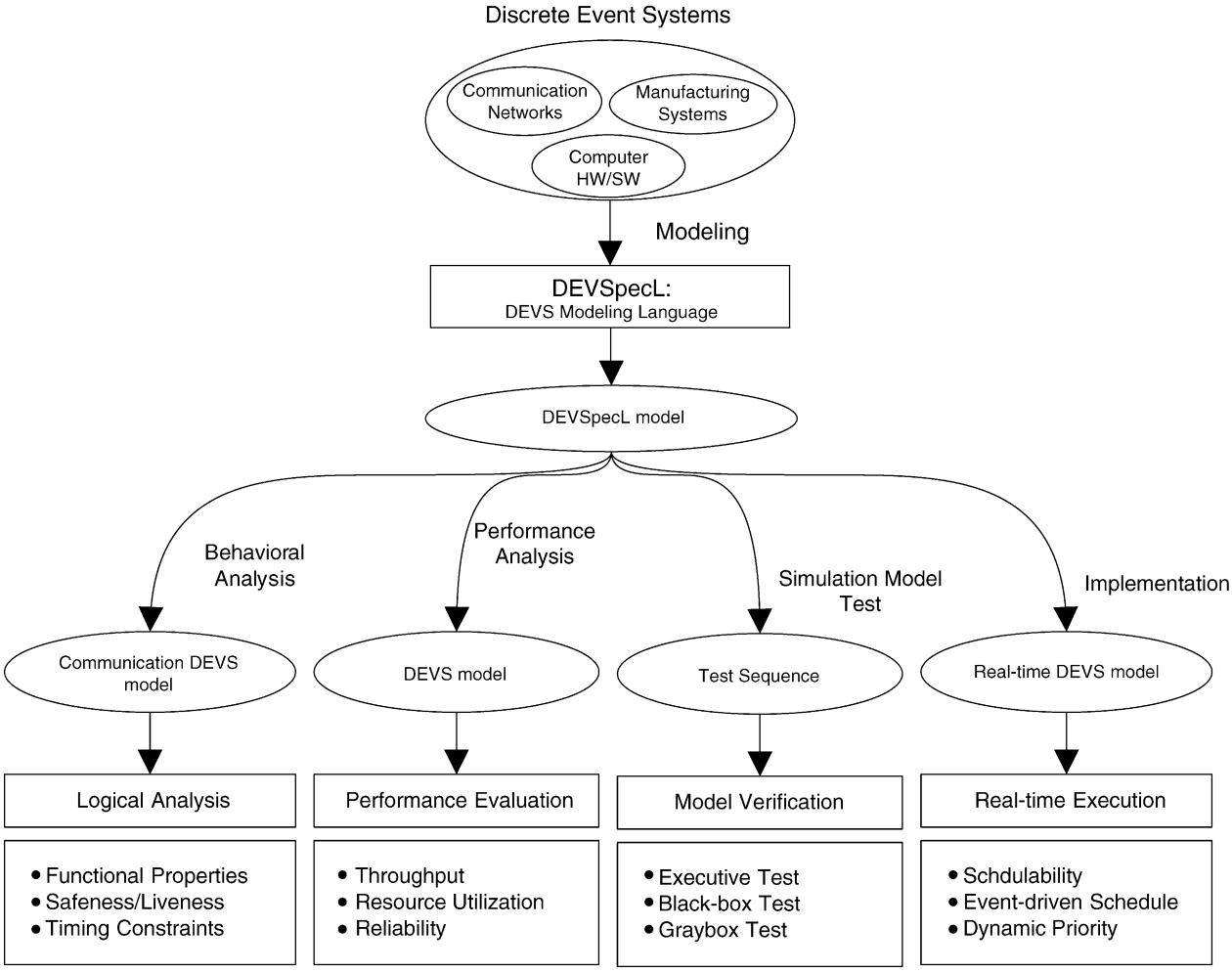

Fig. 1. DEVS-based design process using DEVSpecL.

δ_{int}: $Q \times X \to S$, external state transition function
λ: $S \to Y$, output function
ta: $S \to \Re^{+}_{0,\infty}$, time advance function.

The input event set and the output event set are associated with the input ports and the output ports, respectively. These events provide a solid interface to its environment. The sequential state set includes all states that are related to the behavior of the model. The last four characteristic functions in the 7-tuple of the formalism specify the behavior by defining the relationship among the first three elements.

The second form of the DEVS model, called a coupled model (or coupled DEVS), is a specification of the hierarchical model structure. It describes how to couple component models together to form a new model. This new model can be employed as another component in a larger coupled model, thereby giving rise to the construction of complex models in a hierarchical fashion. Formally, a coupled model CM is defined as [20]:

$$\text{CM} = \langle X, Y, \{M_i\}, \text{EIC}, \text{EOC}, \text{IC}, \text{SELECT} \rangle$$

X, input events set
Y, output events set
$\{M_i\}$, DEVS components set
EIC $\subseteq X \times \cup_i X_i$, external input coupling relation
EOC $\subseteq \cup_i Y_i \times Y$, external output coupling relation
IC $\subseteq \cup_i Y_i \times \cup_i X_i$, internal coupling relation
SELECT: $2^{\{M_i\}} - \phi \to \{M_i\}$, tie breaking selector.

The input and output events provide an interface to the component's neighborhood by associating the events with the input and output ports. The internal coupling defines the connections among the component models in a coupled model. The interaction with outer components is defined by using the external input/output couplings. A detailed discussion about the DEVS formalism and modeling is found in [20].

2.3. DEVS modeling example: alternate bit protocol

DEVS models can be represented graphically for a visual aid. A coupled model is represented by a set of empty boxes each with input/output ports. Each box may be either an

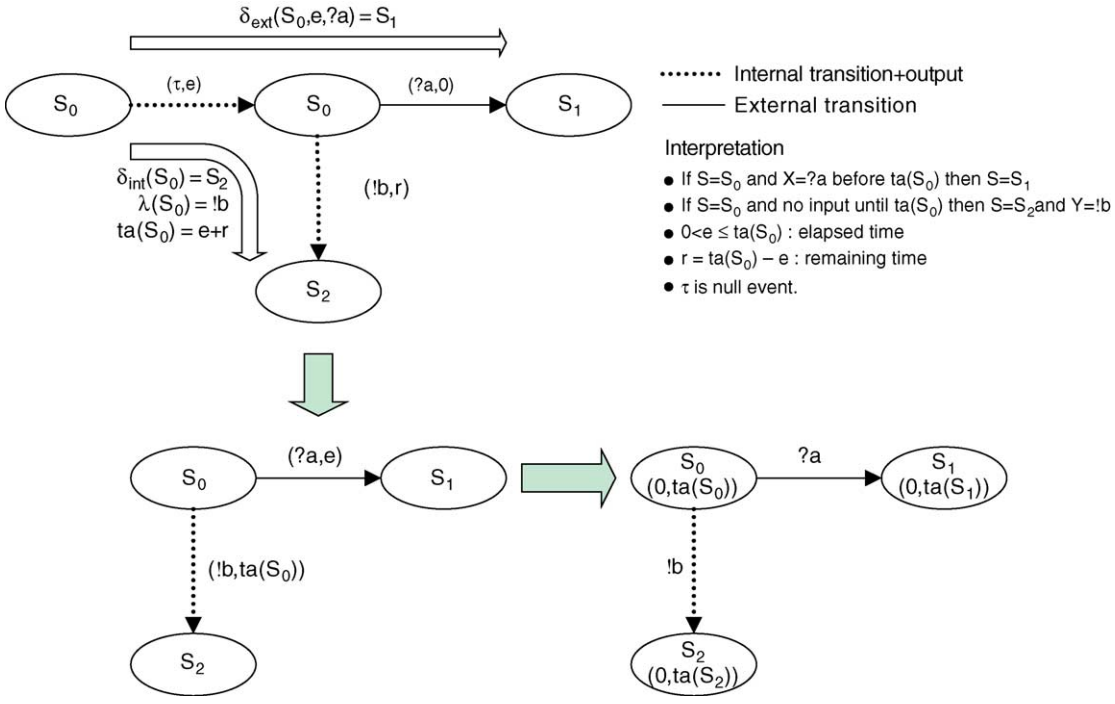

Fig. 2. Semantics of DEVS state transition graph.

atomic or a coupled model. A coupling scheme of the coupled model is represented by directed connections from input to output ports as shown in Fig. 3. On the other hand, an atomic model is represented by a non-empty box with input/output ports in which labeled state transitions are represented. Semantics of the labeled state transitions is shown in Fig. 2.

Fig. 3 shows an overall system model of Alternate bit protocol (ABP), which consists of six atomic models. The Sender delivers a message through input/output ports, then the Receiver acknowledges through Ack message. We shall assume that the transferring/receiving line may lose or duplicate messages (but not corrupt) and the Ack line may lose messages. To determine whether the message is lost or not, both the Sender and the Receiver are set a timer to check if a message is arrived in a specified time interval. Once a timeout is notified, retransmission is made assuming that a message transmitted previously has been lost. Messages are sent with a tag of a bit 0 and 1 alternatively, and also the acknowledgements are constituted of the bits. Source sends 0/1 data to sinker. A received data of sinker should have the same information of the data, which is generated, from the source. Details of such description can be found in [6,18]. Figs. 4 and 5 shows an atomic DEVS model for the ABP Sender in which inputs, outputs, states and state transitions are represented. Note that an external transition arc has an input event which consists of two parts, a name of an input and conditions, which are connected by the symbol '@'. The arc is a representation of a conditional state transition, meaning that the transition is enabled only if the conditions after @ are satisfied. The representation is used in other atomic models of the ABP model. Figs. 6–8 is a model for transmission line fault. Transmission line faults causes loss of sending data and loss of receiving acknowledgement signal, but ABP Sender and Receiver model have retransmission mechanism against transmission line faults.

3. EVS modeling language: DEVSpecL

3.1. Overview of DEVSpecL

A discrete event system may have a collection of subsystems, each of which again has a collection of subsystems, thus being in a complex structure. Accordingly, DEVSpecL should be able to treat a model of such a complex discrete event system, which may be too large and complex to contain in one file. The model can be specified in a collection of files which are compatible to the modular structure of DEVS property.

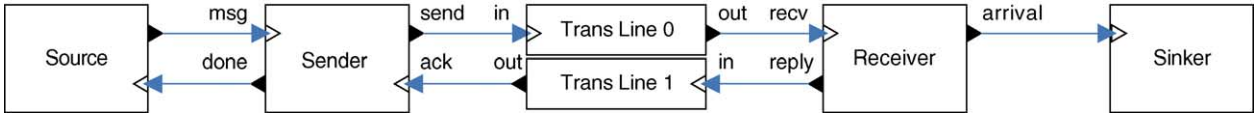

Fig. 3. Coupled DEVS model: alternate bit protocol.

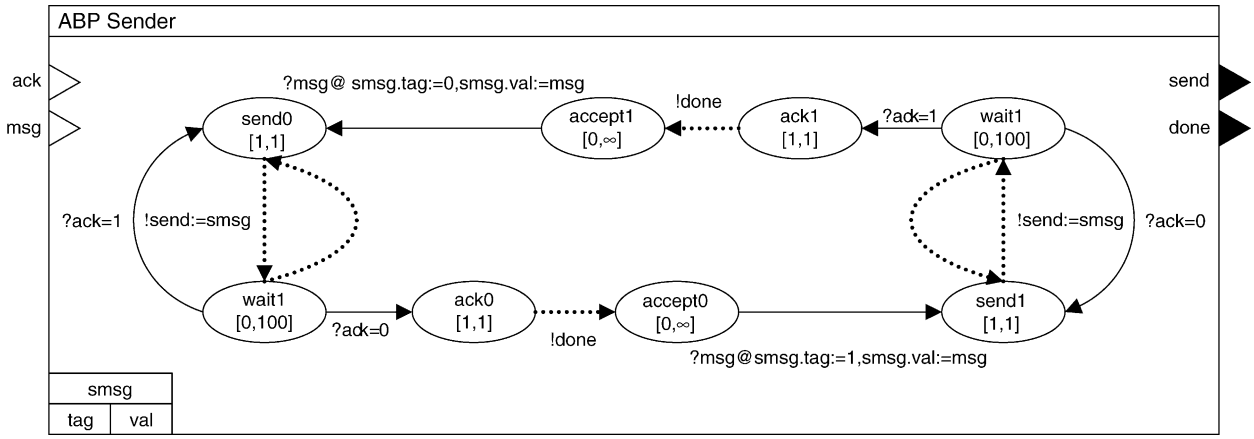

Fig. 4. Atomic DEVS model: ABP sender.

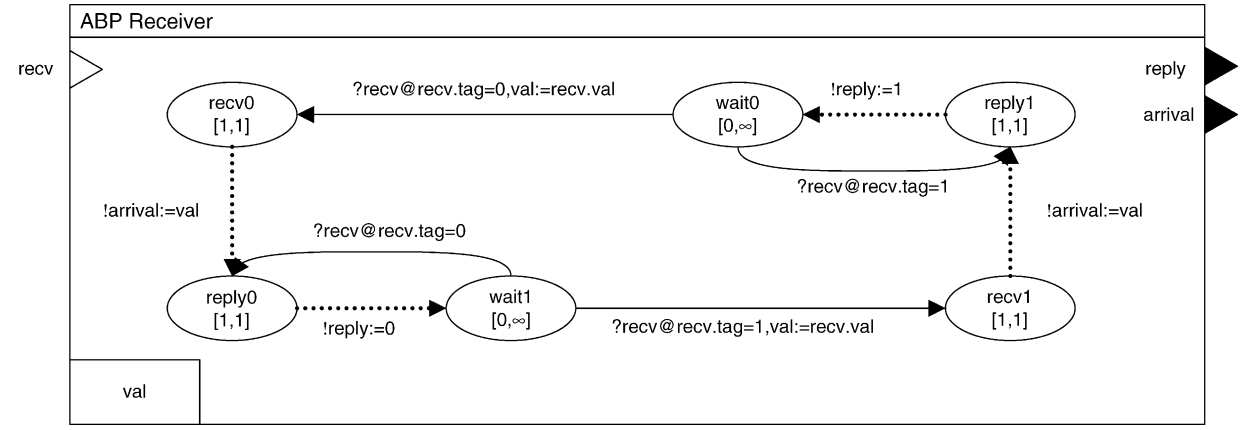

Fig. 5. Atomic DEVS model: ABP receiver.

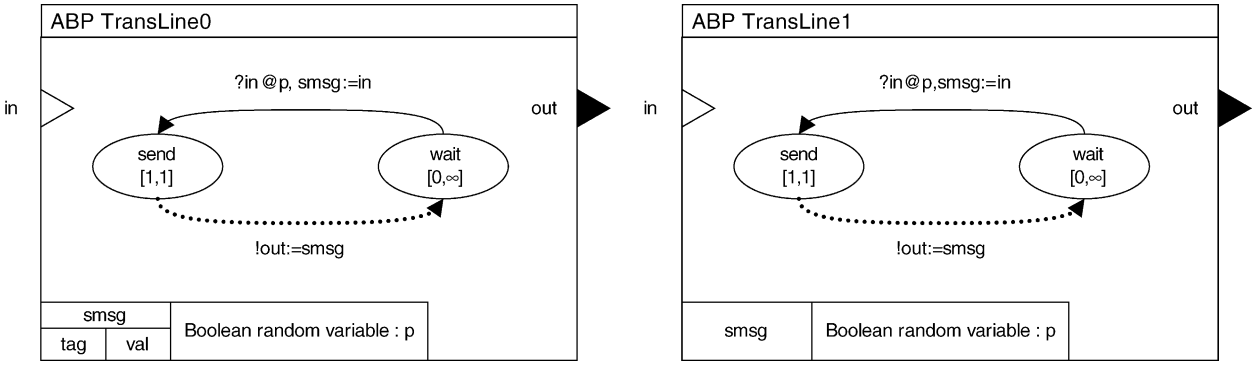

Fig. 6. Atomic DEVS model: ABP TransLine0/TransLine1.

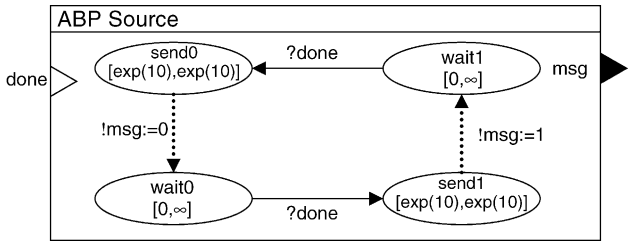

Fig. 7. Atomic DEVS model: ABP source.

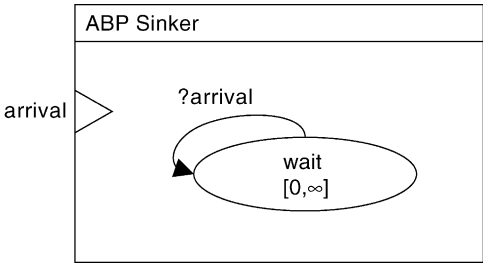

Fig. 8. Atomic DEVS model: ABP sinker.

DEVSpecL is devised to be a tool which supports the seamless design process shown in Fig. 1. As will be explained in Section 4, DEVSpecL is a basis on which code generators for specific applications executed in various programming environments are implemented. Basically, DEVSpecL supports only basic features, such as modular and object-oriented. To be general, however, the design concepts for DEVSpecL is such that DEVSpecL supports user defined APIs through which general features of programming languages, such as concurrency control, process/thread synchronization, interrupt treatment, and others, can be used. A modeler should implements functions for APIs in a programming environment in which an application is executed. Thus, DEVSpecL can virtually do almost everything that a general programming environment can do.

Message definition has inheritance relation from a parent to a child messages and consists of field variables and member functions. If a name of a member function is identical to that of a parent message, polymorphism is applied to that function from inheritance relation.

Variables can be defined as information of an event and a state. A state variable can be classified into either shared or not. A shared state variable means a global variable.

DEVSpecL grammer : variable definition

```
defvar ::= name ':' type | name ':' type '=' expr
type ::= name | type '[' integer_value ']' | [name] '{' enumlst '}'
       | float | integer | string | void | time | seq of  type
enumlst ::= enum | enumlst ',' enum
enum ::= name | name '=' integer_value
```

Built-in types in DEVSpecL include float, integer, string, void, and time. Seq type is specified as the message type of queue, which has push and pop operations. The push operation stands at the tail of a queue, and the pop operation removes at the head of the queue. And, enum type is used for enumeration of state variables.

Function is defined by the following BNF form:

DEVSpecL grammer : function definition

```
deffnt ::= type name '(' [argslst] ')' '{' [ expr ';']* '}'
argslst ::= argslst ',' defvar | defvar
```

A function can be defined in global area, message structure and model structure. The defined function call is performed by the scoping rule, which depends on global and local function. A function in a message and a model structure has polymorphism property like Java language's one. An example of message definition in ABP DEVS model is as follows:

3.2. Syntax and features

Modules in DEVSpecL can be loaded by the following rules.

DEVSpecL grammer : module load

```
include  name;
```

ABP msgtype DEVSpecL description

```
include DEVSIF.dif
message msgtype
       tag : integer;
       val : integer;
end msgtype;
```

Required modules are loaded sequentially in order of the include module description.

Events in a discrete event system can contain additional information which can be represented by a structural type as defined in other general purpose programming languages such as C++. Such information structure is defined by message syntax as the following BNF rules:

DEVSpecL grammer : message definition

```
message_definition ::=  message  name [: parent_name]
                        [ deffnt | defvar ; ]*
                        end name ;
```

DEVSpecL specifies an overall model in three parts: interface, atomic model, and coupled model. An atomic model and a coupled model have input/output events set. Such a common set is defined as interface specification in DEVSpecL. A DEVSpecL model preserves model information defined in the DEVS formalism and supports object-oriented feature. In DEVSpecL, input/output events are

defined by interface, which is described in the extended BNF (EBNF) format as:

```
DEVSpecL grammer : interface definition
  interface_definition ::=  interface name
              input ':' '{' defportlst '}'
              output ':' '{' defportlst '}'
       end name ';'
  defportlst ::= defportlst ',' defport | defport
  defport ::= name ':' type  | name
```

Input/output events set X, Y of an atomic DEVS model AM is defined in interface specification. The state set S, characteristic function δ_{int}, δ_{ext}, λ, and ta of atomic model is described in the atomic model definition as the following extended BNF format:

```
DEVSpecL grammer : atomic model definition
  atomic_mdl_definition ::= atomic model  name [ ':' name ]
                            state variables ':'[ defvar ]*
                            initial condition ':' [ expr ';']*;
                            internal transition ':' [ expr ]*
                            external transition ':' [ expr ]*
                            output function ':' [ expr ]*
                            time advance ':' [ expr ]*
                            [member ':' [ deffnt ]*]
                      end name ';'
```

An atomic model in DEVS formalism does not have specification of an initial condition, but DEVSpecL specifies an initial condition explicitly for simulation and analysis in practice.

To specify an atomic DEVS model the four characteristic functions, namely external transition, internal transition, output, and time advance functions, should be defined. An external transition can occur whenever an input event arrived before a time specified in time advance function is completely elapsed. An internal transition and an output event would occur simultaneously at a time specified in time advance function unless an input event arrives before the time. However, if an input event occurs before the time, the pre-specified time for an internal transition and an output event may or may not be changed. If the time is not changed, we call such condition CONTINUE, meaning that the time continues advancing to the pre-defined time. If the time is changed then a new internal transition time needs to be specified for a next internal transition.

The ABP sender model shown in Fig. 4 can be specified in DEVSpecL as follows.

```
ABP Sender atomic model DEVSpecL description
include msgtype.dif
interface Sender
       inputs: {msg:integer, ack:integer}
       outputs: {send:msgtype, done}
end Sender;
atomic model Sender
       state variables:
              phase: {accept0,send0,wait0,ack0,accept1,send1,wait1,ack1};
              smsg: msgtype;
       initial condition:
              phase := accept0;
              smsg:= new msgtype;
       internal transition:
              (phase == send0) => { phase:=wait0; }

       external transition:
              (phase == accept1) * msg => { smsg.tag = 1; smsg.val = msg; }

       output function:
              (phase == ack0) => done;

       time advance:
              (phase == accept0) => infinity;

end Sender;
```

Let's now consider specification of DEVS coupled models. Input/output events set X, Y of a DEVS coupled model CM is defined in interface specification. The rest tuples used in definition of a coupled model, such as $\{M_i\}$, EIC, EOC, IC, and SELECT, are described in a coupled model definition as the following extended BNF format:

```
DEVSpecL grammer : coupled model definition

  coupled_mdl_definition ::= coupled model name [ ':' name ]
       component ':' '{' argslst '}'
       external input coupling ':''{' [this '.' inport '->' child '.' inport ';']* '}'
       external output coupling ':' '{'[child '.' outport '->' this '.' outport ';']*'}'
       internal coupling ':' '{' [child '.' outport '->' child '.' inport ';' ]* '}'
      [select ':' '{' [ enum ]* '}']
   end name ';'
```

Definition for three coupling relations—external input coupling, external output coupling, and internal coupling—has an identical semantics in DEVS formalism. Definition of a tie-breaking selector in DEVSpecL is optional, which can be defined by textbfselect syntax. If definition of **select** is given, priority between components is assigned by the order of component definitions. A DEVSpecL example of the ABP shown in Fig. 3 is the following as:

```
ABP coupled model DEVSpecL description

include Sender.dif
include Receiver.dif

interface ABP
       inputs: {}
       outputs: {}
end ABP;
coupled model ABP
       component: {
              sender : Sender,
              receiver: Receiver,

       }
       external input coupling : {}
       external output coupling : {}
       internal coupling : {
              source.msg -> sender.msg,
              sender.done -> source.done,

       }
end ABP;
```

Function in DEVSpecL supports not only built-in random number generation functions, but also an external function which assists to link other high level programming languages such as Java and C++.

4. Applications of DEVSpecL

4.1. Approach for application: code generation

This section describes an approach to application of DEVSpecL in discrete event systems modeling, analysis and testing. A model specification in DEVSpecL is translated into DEVSpecL Abstract Syntax Tree (AST) through a lexical and a syntactic analyzer. DEVSpecL AST is used to check ill-structured coupling relations and a type mismatch of states/events by a type checker. In addition, DEVSpecL AST is an intermediate form based on which various conversions can be made easily. Fig. 9 shows some applications of DEVSpecL in which various code generators are used to translate DEVSpecL codes to ones specific to applications. For example, the C++ code generator is used for generation of a DEVS model for performance evaluation in C++. The generated C++ code is a DEVS model in C++ whose semantics is the same as one in DEVSpecL. The C++ DEVS model can be simulated with DEVSim++, an object-oriented simulation/simulation environment in C++ [15]. Likewise, the Java code generator is used for generation of a DEVS model for performance simulation using DEVSimJava which is the same as DEVSim++ except for the implementation language. Another application is an automatic generation of test sequences for verification of an implemented model against a specified DEVS model. For the application DEVSpecL is transformed to a timed state reachability graph (TSRG) from which test sequences are generated. TSRG is a graph theory based tool, which expresses the timed input/output behavior of a discrete event system. Details of models

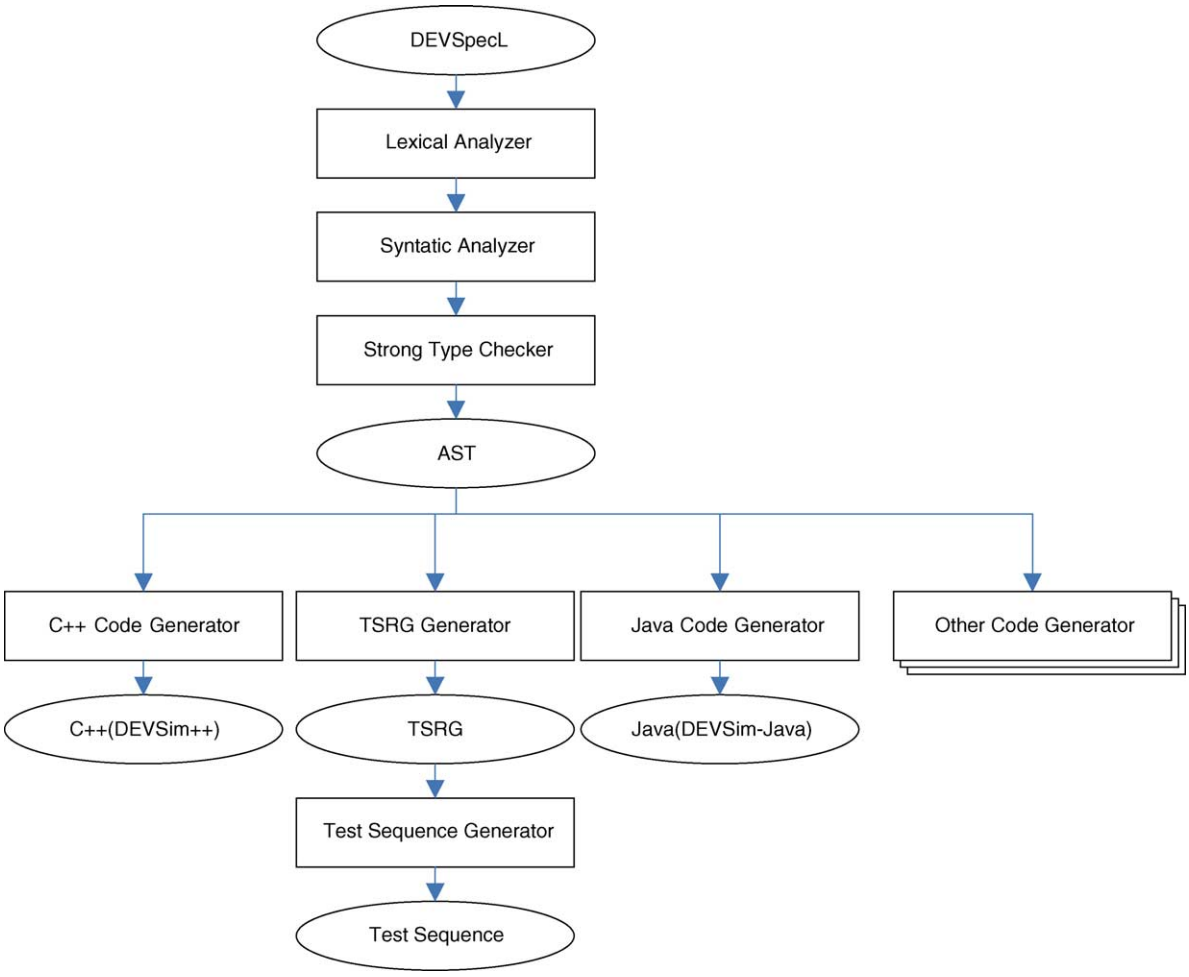

Fig. 9. Applications of DEVSpecL using code generators.

verification using TSRG-based on an black-box approach can be found in [9]. A main advantage of the DEVSpecL approach is that a new application can easily be supported by adding a new code generator, which the application requires. This section introduces two examples for DEVSpecL applications.

4.2. Example I: ABP model

This section describes two examples of code generation from DEVSpecL specification of the ABP model described in Section 2.3. The code generation is based on AST for the ABP model which is shown in Fig. 10.

4.2.1. Performance evaluation of ABP model

As described earlier the first step of performance evaluation is generation of a C++ DEVS model (DEVSim++ model) from DEVSpecL, which can be simulated in the DEVSim++ environment. The DEVSim++ code is generated from the definition of DEVSpecL msgtype message shown in Figs. 3 and 4. Generated code of ABP msgtype message definition is as follows:

ABP msgtype message C++ header

```
#include "DEVSIF.h"
class msgtype:public CDEVSObject{
public:
        int val;
        msgtype() {};
        virtual ~msgtype() {};
        virtual void PrintOn(ostream& strm=cout) const {
                strm << "val =" << val << endl;
        }
};
```

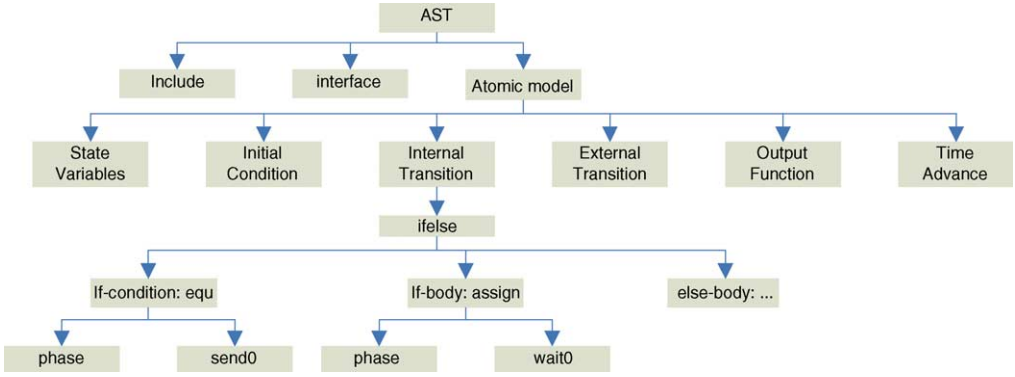

Fig. 10. Abstract syntax tree for ABP sender.

The atomic model ABP Sender in DEVSpecL shown in Fig. 4 is translated as:

ABP Sender atomic model C++ header

```
#include "DEVSIF.h"
#include "msgtype.h"
class Sender: public CAtomic {
public:
        enum { accept0, send0, wait0, ack0, accept1, send1, wait1, ack1 };
        int phase;
        msgtype* smsg;
};
```

ABP Sender atomic model C++ body

```
#include "Sender.h"
Sender::Sender()
{
        add_inports(2,"msg","ack");
        add_outports(2,"send","done");
        phase = accept0;
}
bool Sender::IntTransFn()
{
        if (phase == send0) { phase = wait0; }
        return true;
}
bool Sender::ExtTransFn(const CMessage& m)
{
        if (phase == accept0 && *(m.get_port())=="msg") {
                phase = send0; smsg->tag = 0; smsg->val = *(Integer*)m.get_value();
        } else if (state == wait0 && *(m.get_port())=="ack") {
        return true;
}
bool Sender::OutputFn(CMessage &m)
{
        static Integer* s_mesg = new Integer;
        if (state == ack0) m.set_port_val("done",s_mesg);
        return true;
}
TimeType Sender::TimeAdvanceFn()
{
        if (state == accept) return Infinity;
}
```

The next step for performance evaluation is to determine a performance index to be evaluated. Suppose that we consider an average message delay between a source and a sinker as a performance index. Simulation of the generated DEVS model using DEVSim++ is shown in Fig. 11, since DEVSpecL supports hierarchical modular structure, test sequences from a single component can be easily extracted by using DEVSpecL description.

Test sequences extraction from the ABP Sender model is shown below.

```
ABP Sender model : test sequences
1. ?msg@[0,∞],!send:=(0,0)@[1,1],?ack=0@[0,100],!done@[1,1],
      ?msg@[0,∞],!send:=(1,0)@[1,1],?ack=1@[0,100],!done@[1,1]
2. Prefix(?msg@[0,∞]),!send:=(0,0)@[1,1],t@[100,100],
      Suffix(?msg@[0,∞],!send:=(1,0)@[1,1],?ack=1@[0,100],!done@[1,1])
3. Prefix(?msg@[0,∞]),!send:=(0,0)@[1,1],?ack=1@[0,100],
      Suffix(?msg@[0,∞],!send:=(1,0)@[1,1],?ack=1@[0,100],!done@[1,1])
4. Prefix(?msg@[0,∞],!send:=(0,0)@[1,1],?ack=0@[0,100],!done@[1,1],?msg@[0,∞]),
      !send:=(1,0)@[1,1],t@[100,100], Suffix(!send:=(1,0)@[1,1],?ack=1@[0,100],!done@[1,1])
5. Prefix(?msg@[0,∞],!send:=(0,0)@[1,1],?ack=0@[0,100],!done@[1,1],?msg@[0,∞]),
      !send:=(1,0)@[1,1],?ack=0@[0,100], Suffix(!send:=(1,0)@[1,1],?ack=1@[0,100],!done@[1,1])
```

where an average delay against a transmission line fault ratio is exponentially increasing.

4.2.2. Test sequences generation for ABP model

Test generation in our approach is based on a fundamental assumption on implementation under test (IUP). The assumption is that an input/output interface and a size of states of IPU are known. However, no assumption on implementation languages is made. With the assumption test sequences for the ABP Sender shown in Fig. 4 can be obtained through TSRG transformation. Test sequences generation method is based on conformance test [2,9]. Test sequences are composed of three components: a prefix path, a test path, and a suffix path. The prefix path makes a test target to reach a specified state from an initial state. The test path covers all nodes and edges of the test target. Finally, the suffix path is a homing path of the target machine to reach the initial state. If an implementation under test has many sub-components, the test sequences generation method cannot directly be applied. However,

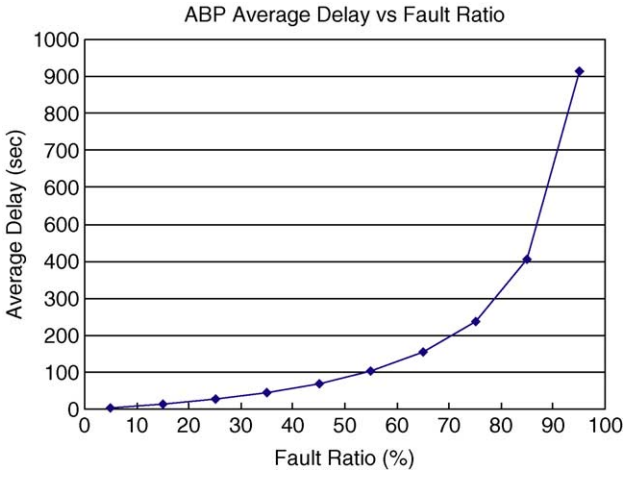

Fig. 11. Average delay vs transmission line fault ratio.

All five test sequences for the ABP sender model in a form of timed input/output events sequences are shown in Fig. 4. The test sequences are employed to check whether an implementation obeys the DEVSpecL specification or not. As shown in Fig. 12, timed input events in the test sequences are first sent to the implementation at a specified time. Then, timed output events of test sequences are expected to be received from the implementation in the specified time interval. If expected output events is not observed in a proper time interval, a fault in the implementation is concluded.

4.3. Example II: carrier sense multiple access with collision detection

This section presents another example for simulation, simulation and verification process using the code generation approach for CSMA/CD protocol shown in Fig. 13. The CSMA/CD protocol has two major facilities for the collision detection and re-transmission mechanism. The overall CSMA/CD model consists of the coupled model STATION and the atomic model MEDIA. STATION means the network node, which is connected to the physical network media and has two atomic models GEN and SEND. GEN merely generates data when SEND transmits data successfully or a collision occurs in MEDIA. SEND checks the status of the transmission line by sending an inquiry message to MEDIA. If the transmission media is available, it sends data to MEDIA. SEND goes to the jamming state when it receives a collision message and tries to resend the collided date after back-off time. MEDIA broadcasts its status to STATIONs when it receives an inquiry message. MEDIA broadcasts the collision message to the STATIONs when more than one STATIONs try to send data simultaneously [11].

The complete DEVS model for the whole CSMA/CD protocol may not be presented in this paper. Instead, the DEVS atomic model SEND, a core model in CSMA/CD is

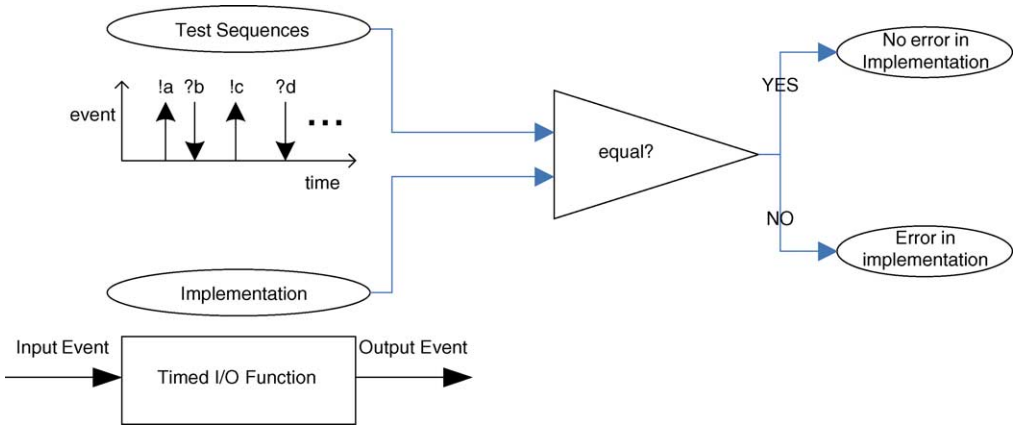

Fig. 12. Verification of implementation using test sequences.

shown in Fig. 14. The SEND model shows the collision detection and retransmission mechanism in CSMA/CD. The SENSE state means the collision detection; the JAMMING state processes the retransmission. From Fig. 14, the following DEVSpecL description can be obtained.

```
CSMA/CD SEND : DEVSpecL

interface SEND
      input : {job, busy, free, collision, done}
      output: {send, is_busy, done, retry}
end SEND;
atomic model SEND
      state variables:
             phase : {READY,SENSE,WAIT,SEND,SENDING,DONE,JAMMING};
      initial condition: phase := READY;
      internal transition:
             (phase=SENSE)=>{phase:=WAIT;}
             else(phase=SEND)=>{phase:=SENDING;}

      external transition:
             (phase=READY)*job => (phase:=SENSE;}

      output function:
             (phase=SENSE)=>is_busy;

      time advance:
             (phase=READY)=>infinity;

end SEND;
```

By the same process used in Example I of ABP model the CSMA/CD model in C++ is obtained and successfully simulated in DEVSim++. The performance index of CSMA/CD model is delay, which is defined as an average time delay per an average sending data size (byte). Figs. 15 and 16 shows such an example of CSMA/CD performance evaluation by the proposed framework. Likewise, we can obtain the following test sequences to verify CSMA/CD SEND model.

5. Conclusion

A DEVS-based framework for discrete event systems design needs a unified specification language from which specific tools required in design processes can be developed. This paper has developed one such language called DEVSpecL, which supports development of tools for a seamless systems development environment. Abstract syntax tree in DEVSpecL has played an important role to generate various codes specific to applications. Generated codes ranged from C++ DEVS models for performance evaluation to TSRG for test sequences generation for verification of a model implementation. To demonstrate effectiveness of the proposed framework, examples of

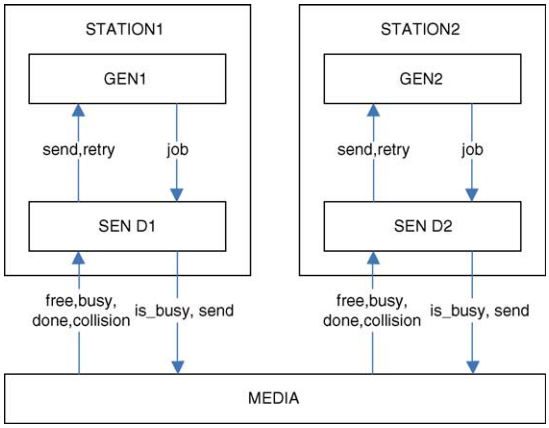

Fig. 13. The overall structure of CSMA/CD.

performance evaluation and test sequences generation for ABP and CSMA/CD has been presented. Although the DEVS formalism is widely used in modeling/simulation of discrete event systems no specification language which supports development of design tools has been proposed. To our best knowledge, DEVSpecL is the first specification language in the DEVS research area that meets such a goal. We believe that DEVSpecL and the proposed design framework are a sound direction for future research on DEVS-based systems design. DEVSpecL would be a vehicle based on which practical tools for analysis of discrete event systems, which may be different from ones shown in this paper, can be developed.

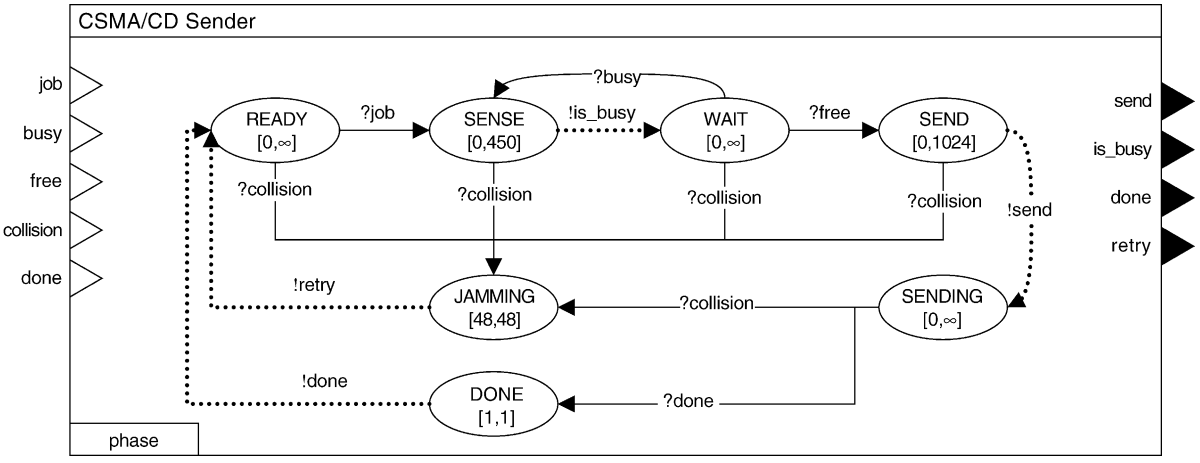

Fig. 14. DEVS model of SEND in CSMA/CD.

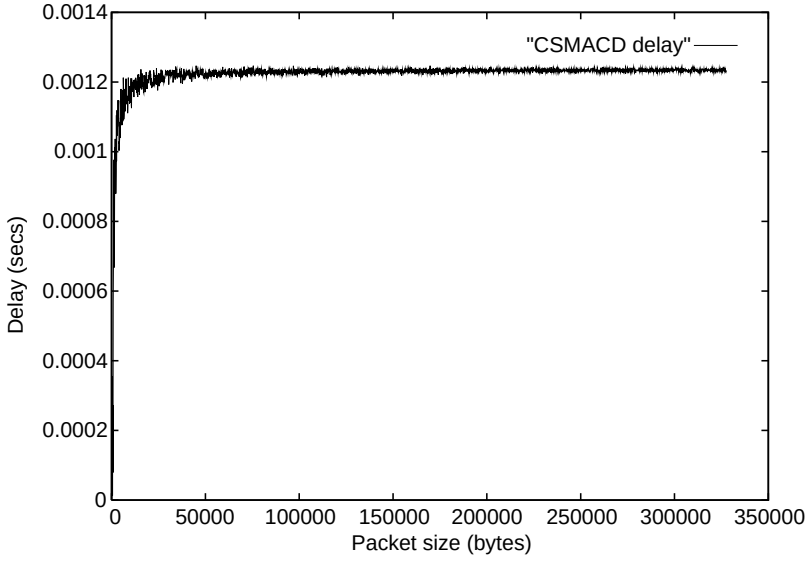

Fig. 15. CSMA/CD: simulation results for delay evaluation.

CSMA/CD Sender model : test sequences
1. ?collision@[0,∞],!retry[48,48]
2. ?job@[0,∞],?collision@[0,450],!retry@[48,48]
3.*Prefix*(?job@[0,∞]),!is_busy@[0,450],?busy@[0,∞]
4. ?job@[0,∞],!is_busy@[0,450],?collision@[0,∞]!retry[48,48]
5. ?job@[0,∞],!is_busy@[0,450],?free@[0,∞],?collision@[0,1024]!retry[48,48]
6. ?job@[0,∞],!is_busy@[0,450],?free@[0,∞],!send@[1024,1024]?collision@[0,]!retry[48,48]
7.?job@[0,∞],!is_busy@[0,450],?free@[0,∞],!send@[1024,1024]?done@[0,]!done[1,1]

Fig. 16. CSMA/CD: test sequences for SEND model.

References

[1] S.M. Cho, T.G. Kim, Real time simulation framework for RT-DEVS models, Transactions of the Society for Computer Simulation 18 (4) (2001) 178–190.

[2] A.T. Dahbura, K.K. Sabnani, M. Ümit Uyar, Formal methods for generating protocol conformance test sequences, Proceedings of the IEEE 78 (1990) 1317–1325.

[3] Chie Dou, A timed-SDL for performance modeling of communication protocols, IEEE Globecom'95 3 (1995) 1585–1589.

[4] J. Helovuo, S. Leppanen, Exploration testing, Proceedings of the International Conference on Application of Concurrency to System Design 2001 (2001) 201–210.

[5] J.E. Heiser, An overview of software testing, Proceedings of the IEEE Autotestcon 1997 (1997) 204–211.

[6] G.J. Holzmann, Design and validation of computer protocols, Prentice Hall, Englewood Cliffs, 1991.

[7] G.P. Hong, T.G. Kim, A framework for verifying discrete event models within a DEVS-based system development methodology, Transaction of the S.C.S International 13 (1996) 19–34.

[8] J.S. Hong, H.S. Song, T.G. Kim, K.H. Park, A real-time discrete event system specification formalism for seamless real-time software development, Discrete Event Dynamic Systems 7 (1997) 355–375.

[9] K.J. Hong, Discrete event model verification methodology using system morphism, Doctoral Thesis, Department of EECS, KAIST, 2005.

[10] IEEE standard for information technology requirements and guidelines for test methods specification and test method implementation for measuring conformance to POSIX standards, IEEE Standard 2003–1997 (1998).

[11] IEEE standard 802.3, Supplements to Carrier SEnse Multiple Access with Collision Detection 1988.

[12] ISO standard, Estelle—a formal description technique based on an extended state transition model, ISO 9074, 1987.

[13] ISO standard, LOTOS—a formal description technique based on the temporal ordering of observational behaviour, ISO 8807, 1989.

[14] ITU-T, CCITT specification and description language (SDL), ITU-T Z.100, Mar 1993.

[15] T.G. Kim, DEVSim++ user's manual: C++ based simulation with hierarchical, modular DEVS models, Technical Report, System Modeling Simulation Laboratory, KAIST, Daejon, Korea, 1994.

[16] T.G. Kim, W.B. Lee, S.M. Cho, DEVS framework for systems development: unified specification for logical analysis performance evaluation and implementation, in: H.S. Sarjoughian, F.E. Cellier (Eds.), Discrete Event Modeling and Simulation: a Tapestry of Systems and AI-based Theories and Methodologies A Tribute to the 60th Birthday of Bernard P. Zeigler, Springer, Berlin, 2001, pp. 131–166.

[17] T.G. Kim, S.B. Park, The DEVS formalism: hierarchical modular systems specification in C++, 1992 European Simulation Multiconference, York, UK (1992) 152–156.

[18] R. Milner, Communication and Concurrency, Prentice Hall, Englewood Cliffs, NJ, 1989.

[19] C. Thomas, H. Luckhoff, T.G. Kim, OpenDEVS: a proposal for a standardized DEVS model exchange format, Proceedings of AIS'96 (1996) 371–377.

[20] B.P. Zeigler, H. Praehofer, T.G. Kim, Theory of Modeling and Simulation, second ed., Academic Press, London, 2000.