Distributed Prototyping from Validated Specifications

David Hansel

Reactive Systems, 120-B East Broad Street, Falls Church VA 22046, USA

Rance Cleaveland, Scott A. Smolka [*]

Department of Computer Science, SUNY at Stony Brook, Stony Brook NY 11794, USA

Abstract

We present `vpl2cxx`, a translator that automatically generates efficient, fully distributed C++ code from high-level system models specified in the mathematically well-founded VPL design language. As the Concurrency Workbench of the New Century (CWB-NC) verification tool includes a front-end for VPL, designers may use the full range of automatic verification and simulation checks provided by this tool on their VPL system designs before invoking the translator, thereby generating distributed prototypes from validated specifications. Besides being fully distributed, the code generated by `vpl2cxx` is highly readable and portable to a host of execution environments and real-time operating systems. This is achieved by encapsulating all generated code dealing with low-level interprocess communication issues in a library for synchronous communication, which in turn is built upon the Adaptive Communication Environment (ACE) client-server network programming interface. Finally, example applications show that the performance of the generated code is very good, especially for prototyping purposes. We discuss two such examples, including the RETHER real-time Ethernet protocol for voice and video applications.

Key words: Automatic code generation, Concurrency Workbench of the New Century (CWB-NC), Adaptive Communication Environment (ACE), model checking, rapid system prototyping

* Corresponding Author.
Email addresses: `hansel@reactive-systems.com` (David Hansel), `rance@cs.sunysb.edu` (Rance Cleaveland), `sas@cs.sunysb.edu` (Scott A. Smolka).
URLs: `www.reactive-systems.com` (David Hansel), `www.cs.sunysb.edu/ rance` (Rance Cleaveland), `www.cs.sunysb.edu/ sas` (Scott A. Smolka).

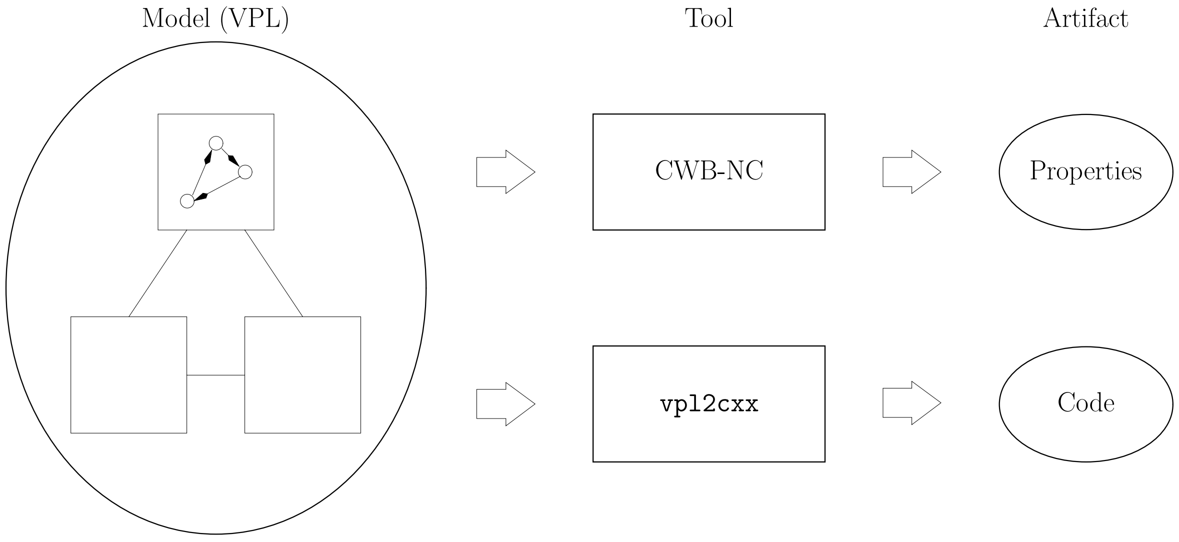

Fig. 1. Overview of model-based code generation.

1 Introduction

Ideally, developers of software for embedded and reactive systems would like to be able to produce high-level designs of their software that are abstract, understandable to domain/application experts, executable, and automatically verifiable. They would then be able to automatically generate efficient and certified code from their designs that can run on a wide variety of execution platforms and real-time operating systems. Even if the code produced was not production-quality, it would at least function as a serviceable *rapid system prototype* that could be used to obtain useful feedback on run-time issues.

In this paper, we report on work inspired by this ideal. In particular, we present a framework in which systems are specified in VPL, an abstract system design language based on value-passing CCS, Milner's well-known process calculus (Milner, 1989). VPL specifications can be processed using the front-end generated by the Process Algebra Compiler (PAC) (Cleaveland et al., 1995) for the Concurrency Workbench of the New Century (CWB-NC) (Cleaveland and Sims, 1996).[1] The automatic simulation and verification facilities of the CWB-NC, which include model checking, equivalence checking and refinement checking, can then be applied to VPL designs in order to check its correctness. The tool described in the current paper may be used to generate C++ prototypes automatically from the analyzed designs. We refer to this approach to system design and implementation as *model-based code generation*; Figure 1 gives an overview of this idea.

Our code generator, named `vpl2cxx`, uses the ACE (ADAPTIVE Communication Environment) network programming interface (Schmidt and Huston, 2002) so that the code it produces is portable to any operating system containing an implementation of the relevant ACE libraries. `vpl2cxx` also implements two efficient, fully distributed algorithms for scheduling

[1] The CWB-NC was formerly known as the NCSU Concurrency Workbench.

input and output statements offered by different processes in a system: a randomized algorithm for the case where no restrictions are placed on the use of input and output commands as guards in VPL nondeterministic `select` statements; and a deterministic algorithm for VPL specifications in which output guards are prohibited, *á la* the programming languages occam (Jones, 1989) and Ada 95 (Taft and Duff, 1997). The randomized algorithm places no restrictions at the VPL level regarding the use of nondeterminism, and the resulting code is a faithful implementation of the VPL description from which it is derived.

Finally, the generated C++ code is highly readable—owing largely to the fact that all generated code dealing with low-level interprocess communication issues is encapsulated in a library for synchronous communication, which is implemented as a layer on top of the ACE NPI—and bears a strong structural resemblance to the corresponding VPL specification. Readability of the generated code facilitates code maintenance and evolution, and it is also important for promoting acceptance of this technology, since engineers can inspect the resulting C++ code manually.

To assess the feasibility of the proposed approach, we have applied `vpl2cxx` to several examples, including the well-known dining philosophers problem and the RETHER real-time Ethernet protocol for voice and video applications (RNI01, 2001). Our performance figures indicate that code produced by `vpl2cxx` delivers more-than-acceptable performance for a distributed prototype.

The focus of this paper will be code generation, that is, the rapid prototyping of distributed implementations from VPL designs. A closer examination of the PAC front-end generator and the Workbench's verification and simulation capabilities can be found in (Cleaveland et al., 1995) and (Cleaveland and Sims, 1996), respectively.

The rest of the paper develops along the following lines. Section 2 presents the VPL specification language. Section 3 discusses the library for synchronous communication utilized by the code generator to achieve maximal platform independence, and its implementation using the ACE NPI. Section 4 describes the VPL-to-C++ translator underlying the code generator. The dining philosophers and RETHER case studies are the subject of Section 6, while Section 8 offers our concluding remarks. For more details on the code-generation work described here, the reader is referred to (Hansel, 2000).

2 VPL

VPL is intended to support the specification of hierarchical systems of concurrent processes that communicate via message passing over typed channels. The VPL type system includes integers of limited range as well as array and record type constructors. There is also a special type `synch` for channels that transport no data and can only be used for process synchronization. Message passing in VPL is binary and synchronous, requiring a handshake between the sender and receiver for communication to occur. The language is equipped with a formal operational semantics that precisely and unambiguously defines the execution steps VPL designs may engage in. Consequently, VPL system designs are mathematical artifacts and are thus candidates for model checking and other kinds of machine-assisted formal analysis.

A VPL specification consists of a sequence of *declarations*, which may include declarations of constants, types and *systems*. A system is either a network, consisting a collection of subsystems and their interconnections, or a process, consisting of statements describing the actions the process should perform. Each system specification, whether process or network, consists of a header, local declarations, and body. The header specifies a unique name for the subsystem and a list of "formal channels" that will be bound to actual channels when the subsystem is instantiated. Actual channels must be type-consistent with the formal ones introduced in the header.

Declarations local to a network may include specifications of the subsystems of the network and local channels for communication between subsystems. The body of a network is a parallel composition of subsystems. A subsystem declared within a network can be instantiated arbitrarily many times within the subsystem's body.

Declarations local to a process consist of variable and procedure declarations. Procedure bodies, like process bodies, are sequences of statements. Simple statements of VPL are assignments of arithmetic or boolean expressions to variables, and input/output operations on channels of the form: `cname ! expression`, to send a message, and `cname ? variable`, to receive a message. Complex statements include sequential composition, `if-then-else`, `while-do`, and an "or-waiting" style of nondeterministic choice in the form of the `select` statement.

To illustrate the language, the VPL specification given in Figure 2 defines a simple network, `net1`, consisting of the parallel composition of three subsystems: processes `sender` (lines 5–17), `receiver` (lines 18–24), and `buffer` (lines 25–40). Lines 41 and 42 contain declarations of local channels that are used in the definition of the body (lines 43–45) to connect the subsystems together. Process `sender` communicates with process `buffer` over channel `c1` while `buffer` and `receiver` communicate over channel `c2`. Note that `buffer` is bidirectional; its body contains a `select` statement that allows it initially to accept inputs on either of its channels, after which it delivers an output on the other channel. In this example, it should be noted that data only flows in one direction.

3 Library for Synchronous Communication

This section describes the library for synchronous communication upon which `vpl2cxx` is built. We were faced with two primary challenges in implementing the library. Firstly, communication in VPL is synchronous, whereas the basic communication primitives accessible from ACE are asynchronous. The second problem is the presence of input and output guards

```
value MAX : 5
type t: MAX

network net1()
begin

  process sender( chan : t )
  begin
    var i : t
    i := 0;
    while (1=1)
      begin
        chan!i;
        if i=MAX-1 then
          i := 0
        else
          i := i + 1
      end
  end;

  process receiver( chan : t )
  begin
    var i : t
    while (1=1)
      chan?i
    end
  end;

  process buffer( chan1 : t, chan2: t )
  begin
    var buf : t
    while (1=1)
      select
        begin
          chan1?buf;
          chan2!buf
        end
      % begin
          chan2?buf;
          chan1!buf
        end
      end
    end
  end;


  Channel c1 : t
  Channel c2 : t

    sender( c1 )
  | receiver( c2 )
  | buffer( c1, c2 )
end
```

Fig. 2. An example VPL specification.

in VPL, i.e. send and receive statements appearing as the initial commands in `select` statement alternatives. In such a setting, obtaining a symmetric, fully distributed, deterministic solution to the *input/output guard-scheduling problem* is known to be impossible (Francez and Rodeh, 1980). [2]

To cope with the problem of input/output guard scheduling, we implemented two scheduling protocols as part of our library for synchronous communication. One of these is deterministic and its use is limited to VPL programs in which output guards are not allowed. (In practice, many VPL specifications do not deploy output guards, so this restriction is a reasonable one.) Our second protocol is randomized and allows the use of both input and output guards in a VPL program. Both protocols are described in detail in Section 5.

Additionally, our implementation was guided by the following principles.

[2] By *symmetric* we mean that all processes execute the same code; *Fully distributed* means that there is no centralized structure on which the protocol relies. For example, a fully distributed protocol can deploy shared variables but only if they are shared by at most two processes each. *Deterministic* means that processes may not toss coins.

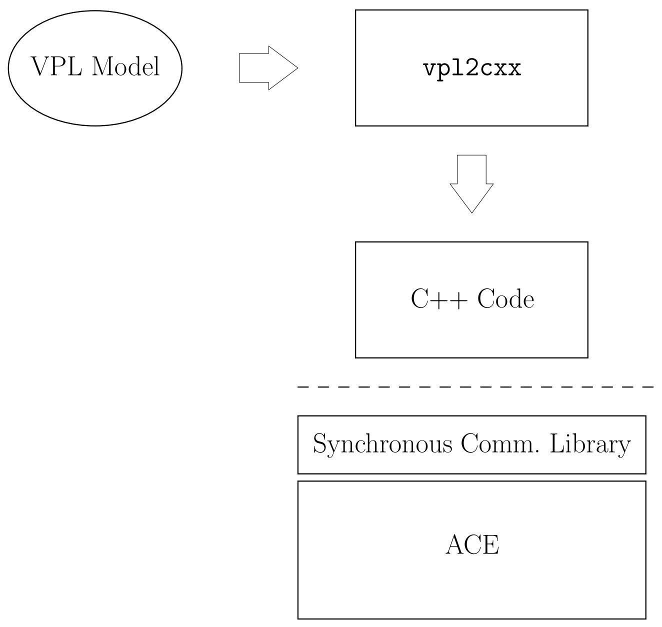

Fig. 3. Overview of the `vpl2cxx` tool.

- *Distributability.* The generated code should be able to run over both local- and wide-area networks using TCP/IP, in such a way that every process can run on a different machine. This means that each VPL process should be translated into a separate executable C++ program that can be run in a stand-alone fashion, automatically establishing the connections it needs to communicate.
- *Portability.* The generated code should be easily portable to different operating systems. This led to the choice of C++ as the target language for translation and of the ACE library (see Section 3.3) as the basis on which this communication library is built.
- *Readability.* It should be possible to easily compare the generated code with the input VPL code. Therefore, the syntax of the communication library's exported methods should not differ from the syntax of the corresponding VPL statements more than is syntactically necessary.

The role played by the library for synchronous communication, and the underlying ACE package, in the overall scheme of the `vpl2cxx` tool is illustrated in Figure 3.

3.1 The API

The API of the synchronous communication library exports the following.

3.1.0.1 Name servers. Each channel is represented by the IP address of the machine on which the associated server is running and the port of that machine on which the server is listening. Name servers provide the needed mapping from channels names to IP/port addresses. The command

```
server -sns[port]
```

initiates a name server that listens for commands over the specified port. If the port argument is omitted, `9999` is chosen by default.

3.1.0.2 Channel servers. A channel server can be initiated via the command

```
server [-nsaddress:port] channel_name
```

Upon creation, the server opens a free port on which it will listen for requests; waits until it is able to connect to the name server specified by the `-ns` option; registers its channel with the name server; waits for clients to connect on the opened port; and then processes client requests until the server is shut down. The waiting on the name server is due to the potentially distributed nature of the execution environment the channel server and name server cohabit.

Although channels in VPL have a designated type, no type is assigned to a server when it is started. To send a value over a channel, the sending process must convert the value to a string of bytes which are then transferred to the receiving process. The receiver has to transform the received byte string back to the channel's data type. The transformations between VPL data types and byte strings are performed automatically by the `VPLTypes` class (see below).

3.1.0.3 The Channel class. The `Channel` class supplies the basic methods for a client to connect to a server and perform synchronous input and output. In particular, there are methods to create a channel (constructor), open a connection to the server for a channel of a given name, close the connection to a channel, and perform synchronous input and output to a channel.

3.1.0.4 The Action class. The `Action` class implements the VPL `select` statement and contains methods to instantiate input, output and internal actions, construct an *action pool* (i.e. pool of requested actions from which one should be selected for execution), add an action to a pool, nondeterministically choose an action from a pool, return the index of the action chosen, and clear a pool.

3.1.0.5 Classes for transforming types. As discussed above, no type is assigned to a channel server. To send a value over a channel, the sending process must convert the value to a string of bytes which are then transferred to the receiving process. The receiver has to transform the received byte string back to the channel's data type. The transformations between VPL data types and byte strings are performed automatically by the set of classes `VPLXXX`, where `XXX` is one of the basic VPL types: `Synch`, `INT`, `Array`, `Record`.

3.2 Development Tools

The communication library is also equipped with several useful development tools:

- Given a C++ program produced by the `vpl2cxx` code generator, the *process simulation tool* allows one to interact with the program during execution via communication channels. In particular, the user can connect to a name server, retrieve a list of all available channels, and connect to their corresponding channel servers. The user can then interact with the C++ program, writing communication actions as they would appear in a VPL `select` statement in order to initiate communications.
- The *logging tool* can be used to track and record various kinds of events, such as the ids of server threads (and corresponding channel names) and client threads (and corresponding program names), and the data values successfully transmitted between sender and receiver threads. A logging-level parameter allows the user to control the density of the log files produced by the logging tool.
- The *Message Sequence Chart generator* produces a postscript or `xfig` file containing a graphical rendering, in the form of a message sequence chart (ITU-TS, 1996), of the communication events recorded in one or more log files.

3.3 The ACE Network Programming Interface

To make the communication library portable, we have implemented it on top of the ACE (Adaptive Communication Environment) library (Schmidt and Huston, 2002). The ACE library is a powerful tool that provides an array of communication services that run on a large variety of underlying execution platforms. The functionality provided by ACE is grouped into three main layers:

- The operating system adaptation layer shields the higher ACE layers from the underlying operating system, making the ACE library platform independent. The methods of this layer are called by our synchronous communication library to leverage ACE's platform independence.
- The C++ wrappers layer provides easy-to-use C++ wrapper classes for tasks like concurrency and synchronization (thread management, semaphores, timers).
- The framework layer provides high-level components for connection management and communication-related event handling and demultiplexing. The synchronous communication library uses components in the framework layer to handle the connections between clients and servers as well as timer scheduling.

4 The VPL to C++ Translator

The `vpl2cxx` translator translates VPL processes and networks into stand-alone programs. In the case of a network, the resulting program contains code for starting servers for local channels and for executing the subsystems containing in the network body. The user has the option of generating a Unix shell script or stand-alone C++ program for this purpose. The translation of VPL processes is relatively straightforward due to the presence of the synchronous communication library discussed above. Two VPL control structures, however, require special attention.

- In VPL, procedures are defined within the definition section of a process and therefore all variables declared within the process and all channels passed to the process are also visible within a procedure. Unfortunately, C++ (unlike C) does not allow the nesting of function definitions. This problem can be worked around by defining all local variables of a process *globally* in the target file of the process and translating the nested procedures into functions within the same file.
- `select` statements in VPL may be nested. In order to determine the relevant set of communication guards, the translator *flattens* nested `select` statements so that the branches of the inner `selects` are brought to the outermost level and executed together with the outer-most `select` alternatives.

The translator, like the CWB-NC, is implemented in Standard ML using the *Standard ML of New Jersey* compiler (Appel and MacQueen, 1991). Moreover, it re-uses the CWB-NC's routines for parsing VPL, which were automatically generated by the PAC (Cleaveland et al., 1995). The VPL parsing routines produce a tree of SML structures which the translator recursively processes to generate the target C++ files.

5 Protocols for Input/Output Guard Scheduling

We now present the deterministic and randomized protocols for input/output guard scheduling we implemented as part of our library for synchronous communication. The two protocols share several important attributes:

- Both protocols possess a client/server architecture, with a client process for each VPL process and a server process for each VPL channel. (Recall that communication in VPL is binary, synchronous, and channel-oriented.)
- The client code for handling synchronous input and output in a non-guard context, which appears in Figure 5, is the same for both protocols: a process reaching a synchronous input or output statement in a non-guard context sends a request message to the appropriate server and waits until the server responds affirmatively. The message type encodes the fact that the communication request is occurring in a non-guard context.
- Both protocols make use of the data types `MessageType` and `ChoiceType` given in Figure 4.

```
type MessageType = record
                     type       : { REQ_IN, REQ_IN_GUARD,
                                    REQ_OUT, REQ_OUT_GUARD,
                                    ACK, NACK, CANCELED }
                     id         : int
                     value      : ValueType
                     sender     : ConnectionType
                   end

type ChoiceType =  record
                     type       : { INPUT, OUTPUT, INTERNAL }
                     connection : ConnectionType
                     value      : ValueType
                   end
```

Fig. 4. Common data types for both protocol versions.

Message Type	Direction	Meaning
`REQ_IN`	Client →Server	A client requests an input
`REQ_OUT`	Client →Server	A client requests an output
`REQ_IN_GUARD`	Client →Server	A client requests an input within a non-deterministic choice
`REQ_OUT_GUARD`	Client →Server	A client requests an output within a non-deterministic choice (only in randomized polling protocol)
`ACK`	Server →Client	A match was found
	Client →Server	A reported match was accepted (only in restriction protocol)
`NACK`	Server →Client	No match was found
	Client →Server	A reported match was rejected / cancel previously sent requests (only in restriction protocol)
`CANCELED`	Server →Client	Previously sent requests have been canceled (only in restriction protocol)

Table 1
Meaning of messages within the protocols.

`MessageType` defines a structure for the messages that are exchanged between clients and servers, and contains the following fields:

- `type` holds the type of the message; the significance of these message types is explained in Table 1.
- `id` is used to identify a message with a specific request sent by a client.
- `value`: If a message has to transport a value, it is stored in this field.

```
procedure input( connection : ConnectionType, ref value : ValueType)
begin
  send <REQ_IN> to connection
  receive message from connection
  if message.type = ACK then
    value = message.value
  else
    Protocol - Error !
  endif
end

procedure output( connection : ConnectionType, value : ValueType)
begin
  send <REQ_OUT, value> to connection
  receive message from connection
  if message.type <> ACK then
    Protocol - Error !
  endif
end
```

Fig. 5. Simple input and output handling on client side.

- The field `sender` contains the address of the sender; it is set implicitly by the sender and allows the receiver to ascertain the sender's identity.

In the pseudo-code for the protocols, `MessageType` is instantiated using the shorthand tuple notation `<message_type, id, value>`. Moreover, `id` and/or `value` are sometimes omitted, depending on the context.

Regarding the second common type, `ChoiceType`, an array of `ChoiceType` is used to represent the possible choices in a non-deterministic choice, i.e., VPL `select` statement. `ChoiceType` contains the following fields:

- `type` defines the type of a choice: INPUT, OUTPUT or INTERNAL. In the case of an INTERNAL choice, no communication takes place and this corresponds to the situation where a non-communication statement such as an assignment statement appears in a guard position.
- `connection` identifies the connection to the server that represents the channel on which an input or output is to be performed (undefined if type is INTERNAL).
- `value` holds the value to be sent if `type` is OUTPUT, or the received value if `type` is INPUT.

5.1 Deterministic Protocol

The deterministic protocol for input/output guard scheduling is applicable to VPL specifications in which output statements do not appear as guards in `select` statements. Figure 6

```
function select(choice_count : int, choices : Array of ChoiceType) : int
begin
  var connections         : set of ConnectionType
      connection_states   : array[choice_count] of {ENABLED,DISABLED,UNKNOWN}
      unknown_state_count : int
      selected            : int

  unknown_state_count:=0
  for i=1 to choice_count do
    if choices[i].type=INPUT then
      send <REQ_IN_GUARDED,i> to choices[i].connection
      connection_states[i] := UNKNOWN
      unknown_state_count  := unknown_state_count + 1
      add choices[i].connection to connections
    else if choices[i].type=INTERNAL then
      connection_states[i] := ENABLED
    else
      print("Output guards not handled by deterministic protocol.")
    endif
  endfor
  selected:=-1
  while selected<0 do
    wait for message on connections
    if message.type=ACK then
      if connection_states[message.id]=UNKNOWN then
        unknown_state_count:=unknown_state_count - 1
      endif
      connection_states[message.id]:=ENABLED
      choices[message.id].value:=message.value
    else if message.type=NACK then
      if connection_states[message.id]=UNKNOWN then
        unknown_state_count:=unknown_state_count - 1
      endif
      connection_states[message.id]:=DISABLED
    endif
    if unknown_state_count=0 then
      selected := {randomly choose an enabled action, -1 if all disabled}
      if selected>=0 and choices[selected].type<>INTERNAL then
        send <ACK, selected> to choices[selected].connection
      endif
    endif
  endwhile
  for i=1 to choice_count do
    if choices[i].type<>INTERNAL AND i<>selected
      send <NACK> to choices[i].connection
      repeat
        wait for message on choices[i].connection
      until message.type=CANCELED
    endif
  endfor
  return selected
end
```

Fig. 6. Client side of the deterministic protocol.

```
 1 var inQueue  : Queue of MessageType
 2 var outQueue : Queue of MessageType

 3 procedure server( socket )
 4 begin
 5 while true do
 6   listen on socket for incoming message
 7   if message.type=NACK then
 8     remove all messages sent by message.sender from inQueue
 9     send <CANCELED, message.id> to message.sender
10   else if message.type=REQ_IN_* then
11     server_process_inreq( message )
12   else if message=REQ_OUT then
13     server_process_outreq( message )
14   endif
15 endwhile
```

Fig. 7. Server main-loop of the deterministic protocol.

contains the code executed by clients (i.e., translated VPL processes) upon reaching a `select` statement. It is structured as the function `select()` which takes as an input parameter an array of records of type `ChoiceType`, representing all of the VPL statements appearing as guards in the `select` statement in question. A guard may be an input statement or an internal action corresponding to a non-communication statement. In the latter case, no action is taken by the function.

For each input guard in the array, the protocol sends a message to the server corresponding to the input channel named in the guard (recall that the address of the server is available in field `connection` of `ChoiceType`). Such a message represents a request by the client to the server to find a matching communication partner for the input guard in question. The client then listens for ACK ("enabled") or NACK ("not enabled") messages from all (channel) servers to which requests were sent. When all statuses are known, the client randomly picks one of the enabled actions and confirms it by sending the appropriate server an ACK. If no action is enabled, the client waits until a server responds with ACK. Assuming a client has chosen an enabled action on which to proceed, it cancels its other pending requests by sending a NACK to the other servers. The client can proceed with its communication after receiving CANCELED messages in response to the NACKs it has sent out.

The server, on the other hand, maintains two queues, one for pending input requests (`inQueue`) and one for pending output requests (`outQueue`). It executes an infinite loop (Figure 7) listening for incoming messages, and takes the following actions depending upon the type of the message received (the notation `REQ_IN_*` in line 10 stands for "`REQ_IN` or `REQ_IN_GUARD`"):

NACK: In this case, a client wants to cancel its pending requests. The server dequeues these requests and responds with CANCELED.

Input Request: The code executed by the server for input requests is given in Figure 8. It first checks if a matching request is available in the queue of pending output requests. If

```
procedure server_process_inreq( message : MessageType )
begin
  var p_message : MessageType
  if outQueue={} then
    // no partner available
    enqueue message to inQueue
    if message.type=REQ_IN_GUARD then
      send <NACK, message.id> to message.sender
    endif
  else
    // partner available
    p_message:=head of outQueue
    send <ACK, message.id, p_message.value> to message.sender
    if message.type=REQ_IN_GUARD then
      // send confirmation and wait for response
      do
        receive message from message.sender
        if message.type=REQ_IN_GUARD then
          send <ACK, message.id, p_message.value> to message.sender
        endif
      while message.type=REQ_IN_GUARD
    end
    if message.type=NACK then
      send <CANCELED, message.id> to message.sender
    else
      send <ACK, message.id> to p_message.sender
      dequeue head of outQueue
    endif
  endif
end
```

Fig. 8. Server input-request-handling code of the deterministic protocol.

not, the request is stored as pending and, if the request was a guard, the client is notified via a NACK that no match was found. If a matching output request is found, the server reports the match back to the client with an ACK message. If the request was a guard, it is possible that the client has in the meanwhile acted on some other alternative. Thus, the server must wait to discover whether the client accepts or rejects the reported match. In the former case, the client sends the server an ACK to which the server responds with ACK. In the latter case, the client sends a NACK which the server responds to with CANCELED.

Output request: The code executed by the server for output requests is given in Figure 9. The actions taken by the server in this case are similar to the case of input requests, with the following exception. Since output statements may not appear as guards, the server must only wait for confirmation for the matching *input* request it has identified as a match to the output request (and only if the matching input request represents a communication guard), but not for the output request itself.

```
procedure server_process_outreq( message : MessageType )
begin
  var done      : boolean
  var p_message : MessageType
  done := FALSE
  while (NOT done) AND (inQueue<>{}) do
    // try next partner
    p_message := dequeue head of inQueue
    send <ACK, message.id, message.value> to p_message.sender
    if p_message.type=REQ_IN_GUARD then
      // send confirmation and wait for response
      do
       receive p_message from p_message.sender
       if p_message.type = REQ_IN_* then
          send <ACK, message.id, message.value> to p_message.sender
       end
      while p_message.type=REQ_IN_*
    endif
    if p_message.type=NACK then
      // received NACK => respond with CANCELED
      send <CANCELED, p_message.id> to p_message.sender
    else
      done := TRUE
    endif
    remove all messages sent by p_message.sender from inQueue
  endwhile
  if done then
    // success => inform sender of message
    send <ACK, message.id> to message.sender
  else
    enqueue message to outQueue
    if message.type = REQ_IN_GUARD then
      send <NACK, message.id> to message.sender
    endif
  endif
end
```

Fig. 9. Server output-request-handling code of the deterministic protocol.

5.2 Example

To illustrate how the deterministic protocol works, consider the simple VPL program of Figure 10 consisting of a network of two sending processes, one sending on channel `a` and the other on channel `b`, and a receiving process listening on both (nondeterministically). Figure 11 contains a Message Sequence Chart (see Section 3.2) depicting the actual messages transmitted by the deterministic protocol. The participating servers and clients are each represented as a vertical line. Servers are titled by the name of the channel they are serving,

```
network net3()
begin
  process sender(chan : synch)
  begin
    while true do chan!* end
  end;

  process selecter(chan1 : synch, chan2 : synch)
  begin
    while true do select chan1?* % chan2?* end end
  end;

  channel a : synch
  channel b : synch

  sender(a) | selecter(a, b) | sender(b)
end;
```

Fig. 10. Example VPL program with two senders and one receiver.

33pc

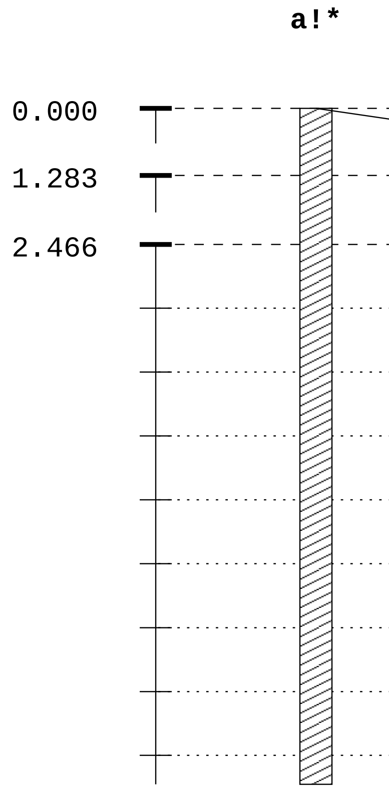

Fig. 11. Run of the example in Figure 10 using the deterministic protocol. This is a diagram of an actual run of this network. The dashed horizontal lines represent a time span of one millisecond.

clients by the communication actions they are trying to perform. The shaded boxes indicate intervals where a client/server is blocked while waiting for a response. Time is displayed along the vertical axis, each division representing one millisecond. To avoid empty space in the diagram, the time axis is interrupted where a longer period of inactivity occurs.

In Figure 11, processes 1 (a!*) and 3 (b!*) start by sending their (non-guard) output requests to the servers. Next, process 2 (a?* % b?*) sends its (guarded) input requests, to which the servers both respond with `ACK` since they can offer a match. After process 2 receives both responses, it chooses a match (here it chooses channel "b") and sends a confirmation back to channel b's server who reports it to process 3. Because process 2 has made its choice, it must cancel its other request on channel a. Meanwhile, process 3 sends its next output request.

5.3 Randomized Protocol

For VPL specifications in which both input and output statements can appear as guards in `select` statements, we have implemented a guard-scheduling algorithm that relies on randomized polling and is inspired by the randomized algorithms of (Francez and Rodeh, 1980; Joung, 2000). Such algorithms achieve a notion of "correctness (freedom from deadlock and starvation) with probability 1".

Like the deterministic protocol, the randomized protocol is client/server-based, with clients corresponding to VPL processes and servers corresponding to VPL channels. The protocol works as follows. Upon reaching a `select` statement at which a certain collection of input/output guards is enabled, a process (client) chooses one of the guards randomly in search of a communication partner. The client thus sends a request to the appropriate server indicating the nature of the communication guard. If the server is successful in finding a match, it will return an ACK to the client with the information the client needs to effect the communication. If unsuccessful, it will return a NACK and the client will randomly choose another guard and repeat the protocol. This is in contrast to the deterministic protocol, which launches requests for all of its input guards at once.

The pseudo-code for the client-side of the randomized protocol is given in Figure 12. As for the deterministic protocol, the client code is organized as function `select()`, which has an input parameter an array of records of type `ChoiceType`.

The server, on the other hand, maintains a *pool* of pending input and output requests, some of which may have come from clients attempting a communication in a non-guard position, and are thus willing to wait "indefinitely". The server, upon receiving a new request, searches the pool for a match. If a match is found, both clients (communication is binary) are sent an ACK and the matching request dequeued. If a match is not found, the server starts a timer and waits for incoming requests until a match is found or the timer expires. In the former case, an ACK is sent to both clients and the timer is canceled. In the latter case, a NACK is sent to the requesting client and its request is dequeued. The pseudo-code for the server

```
function select(choice_count:int, choices:Array of ChoiceType) : int
begin
  var selected : int
  selected := -1
  while selected<0 do
    selected := choose random number between 0 and choices_count-1
    if choices[selected].type=INTERNAL then
      // action was internal => we're done
    else
      // action was either INPUT or OUTPUT
      if choices[selected].type=INPUT then
        send <REQ_IN_GUARD> to choices[selected].connection
      else
        send <REQ_OUT_GUARD,choices[selected].value>
          to choices[selected].connection
      endif
      receive message from choices[selected].connection
      if message.type = ACK then
        // response was ACK => we're done
        if choices[selected].type = INPUT then
          choices[selected].value = message.value
        endif
      else if message.type = NACK then
        // response was NACK => action was not performed
        selected := -1
      else
        protocol-error !
    endif
  endwhile
  return selected
end
```

Fig. 12. Handling of a non-deterministic choice in the randomized polling protocol

side of the randomized protocol is given in Figures 13 and 14.

5.4 Example

The need for randomization in the protocol can be understood by considering a ring network of three processes, each of which attempts to execute a `select` statement consisting of two communication guards: an input guard targeting the process on the left and an output guard targeting the process on the right. The VPL code for such a network is given in Figure 15. In a distributed, symmetric (all processes execute the same code) environment in which processes are not allowed to toss coins, a potential outcome is that all processes independently choose their input guards, leading to deadlock. Randomization breaks such symmetry and ensures that if a communication between two processes is enabled infinitely

```
procedure server( socket )
begin
  var inQueue     : Queue of MessageType
  var outQueue    : Queue of MessageType
  var bufferQueue : Queue of MessageType
  var request     : MessageType

  while true
    if bufferQueue is empty then
      dequeue request from bufferQueue
    else
      wait for request on socket
    endif
    if (request.type=REQ_IN_*) AND (outQueue<>{}) then
      dequeue match from outQueue
      send <ACK, match.value>   to request.connection
      send <ACK>                to match.connection
    else if (request.type=REQ_OUT_*) AND (inQueue<>{}) then
      dequeue match from inQueue, set found:=TRUE if found
      send <ACK, request.value> to match.connection
      send <ACK>                to request.connection
    else
      if request = *_GUARD then
        call server_delay( request, bufferQueue )
      else
        if request.type=REQ_IN_* then
          enqueue request to inQueue
        else
          enqueue request to outQueue
        endif
      endif
    endwhile
  endwhile
end
```

Fig. 13. The server main loop of the randomized polling protocol

often, then some communication will eventually take place with probability 1 (Francez and Rodeh, 1980).

Figure 16 contains a message sequence chart (MSC) depicting the messages exchanged by the randomized protocol during an execution of the ring network example of Figure 15. The MSC illustrates how the randomized protocol is able to find communication partners for this ring of nondeterministic processes: during the underlying execution, a communication over channel `a` succeeds, followed by a communication over channel `b`, followed by a communication over channel `c`, etc. The MSC also illustrates the situation where a client's request cannot be satisfied and must be rejected with `NACK`. This occurs in process 2 (b?* % c!*) at time 0.444 and in process 3 (a?* % b!*) at time 0.464. In both cases, the process again chooses another

```
procedure server_delay( request : MessageType,
                        ref bufferQueue : Queue of MessageType )
begin
  var newRequest : MessageType
  var done       : BOOLEAN
  set timer to expire in x milliseconds
  done := FALSE
  while NOT done do
    wait until newRequest is received on port or timer is expired
    if timer is expired then
      send <NACK> to request.connection
      done := TRUE
    else
      if (newRequest=REQ_IN_*  and request=REQ_OUT_*) or
         (newRequest=REQ_OUT_* and request=REQ_IN_*)  then
        cancel timer
        if request.type=REQ_OUT_*
          send <ACK, processedRequest.value> to newRequest.connection
          send <ACK>                         to request.connection
        else
          send <ACK, newRequest.value> to request.connection
          send <ACK>                   to newRequest.connection
        endif
        done := TRUE
      else
        enqueue newRequest to bufferQueue
      endif
    endif
  endwhile
end
```

Fig. 14. The server delay code of the randomized polling protocol

channel on which it wants to communicate and continues the protocol.

5.5 Complexity Analysis

The computational complexity of the deterministic and randomized guard-scheduling protocols can be analyzed in terms of the expected number of message exchanged between clients and servers during the execution of function `select()` (Figures 6 and 12). Assume a VPL `select` statement having k alternatives; i.e. the value of parameter `choice_count` of function `select()` is k. For the deterministic protocol, the expected number of messages exchanged is linear in k. More precisely, the following messages are transmitted:

(1) The client sends k requests to the servers.
(2) All k servers send back a `NACK` or `ACK` message. Every server that sent `ACK`, waits for

```
network net2()
begin
  process p( in : synch, out : synch )
  begin
    while true do
      select
        in?*
      %
        out!*
      end
    end
  end;

channel a : synch
channel b : synch
channel c : synch

p(c, a) | p(a, b) | p(b, c)
end;
```

Fig. 15. A VPL network forming a ring of three nondeterministic processes.

the client's response. Servers that sent NACK might subsequently send ACK.

(3) The client sends ACK to one server and NACK to the $k - 1$ other servers.

(4) The $k - 1$ servers receiving NACK respond with CANCELED.

In the worst case, all servers first send NACK, followed by ACK, resulting in $k * 5 - 1$ messages. In the best case, the number of messages is reduced to $k * 4 - 1$.

In the case of the randomized protocol, the expected number of messages needed to achieve communication is $O(k^2)$. To understand how this bound is derived, first recall that communication in VPL is binary (and synchronous). Therefore, effecting communication in a VPL system is tantamount to an agreement problem between two processes. Secondly, note that the probability that two processes (independently) choose matching alternatives is $\frac{1}{k^2}$. Given that, for each attempt a process makes at finding a communication partner, the corresponding client and server processes each transmit a constant number of messages, the bound of $O(k^2)$ expected messages follows.

5.6 Verification Results

To gain confidence in the correctness of our guard-scheduling protocols, we specified them in VPL and used the CWB-NC's equivalence checking and model checking facilities to verify several (finite-state) instances of the protocols. In the case of the deterministic protocol, one such instance we considered involved the example VPL network of Figure 10. In particu-

33pc

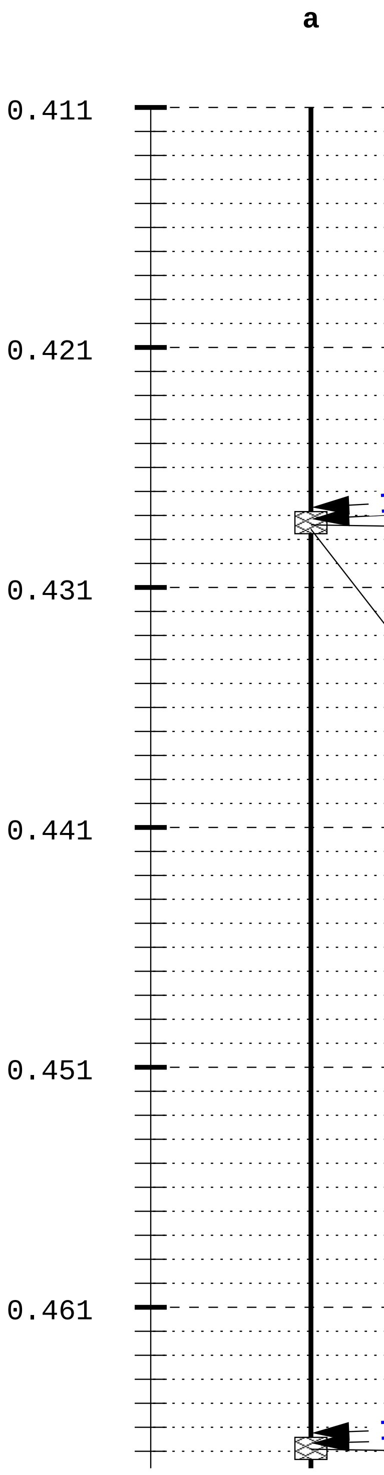

Fig. 16. An example run of the network in Figure 15 using the randomized protocol.

lar, we turned this VPL program into a specification of the protocol by augmenting each sender process with an external channel, and having the sender processes signal success over their respective external channels whenever they succeed in communicating with the selector process. The VPL encoding of the deterministic protocol (Figures 6 to 9) was similarly augmented with external channels, and the CWB-NC's decision procedure for observational equivalence (Milner, 1989) was applied to the VPL encoding of the protocol and its specification. A result of true was obtained indicating that the protocol implementation is indeed observationally equivalent to its specification.

The CWB-NC transforms each VPL program it processes into a labeled transition system (LTS), and by looking at the number of states and transitions in such an LTS, we can get an idea of the complexity of the specification. In the case of the VPL encoding of the deterministic protocol instantiated on the VPL network of Figure 10, the underlying LTS had 11,095 states and 31,052 transitions. Checking this encoding for observational equivalence with its specification took 33 minutes of user time on a SUN Ultra workstation, using a maximum of 150MB of memory.

We also used the CWB-NC to check this example for the absence of livelock. In particular, the CWB-NC's model checker was applied to the VPL encoding of the deterministic protocol to check the following CTL temporal-logic formula:

```
AG AF (<p1c1!*>tt \/ <p3c2!*>tt)
```

This formula states that, for every state, all computations originating from it eventually reach another state in which one of the senders signals successful communication. This model-checking computation also returned a result of true and took about one minute of computation time.

Several instances of the randomized protocol were also verified in a similar manner. One such instance we considered was the ring example of Figure 15. The LTS corresponding to the VPL encoding of the randomized protocol on this example consisted of 9,107 states and 30,300 transition. The total time need for equivalence checking was approximately two minutes. Model checking the encoding for livelock did indeed reveal a livelock whose origin can be traced to the fact that we encoded random choice using VPL nondeterministic choice. As a result, a livelock manifested corresponding to each process repeatedly choosing its input guard, or to each process repeatedly choosing its output guard. In practice, such livelocks are not expected to occur since they correspond to zero-probability events when processes are allowed to toss coins.

6 Experimental Results

This section compares the performance of `vpl2cxx`-generated C++ code with that of hand-written C++ code. Two applications are considered: the well-known dining philosophers

```
cwb-nc> chk diningphil "AG <->tt"
Invoking alternation-free model checker.
Building automaton...
..
States: 270
Transitions: 918
Done building automaton.
TRUE, the agent satisfies the formula.
Execution time (user,system,gc,real):(1.950,0.010,0.020,1.954)
```

Fig. 17. Output of CWB-NC on VPL specification for dining philosophers.

problem, and a real-time Ethernet protocol.

6.1 Dining Philosophers

We coded a solution to the dining philosophers problem in VPL, verified its correctness using the CWB-NC, and ran `vpl2cxx` to automatically generate C++ code. The problem is simple enough that its VPL encoding fits on one page. There is also a C++ solution based on POSIX threads available from the web which we used for benchmarking purposes: `http://hissa.nist.gov/dads/HTML/diningphilos.html`

In the dining philosophers problem, there are five philosophers (processes) sitting around a circular table. Between each pair of philosophers lies a single chopstick and in the center of the table lies a bowl of spaghetti. Each of the philosophers repeatedly cycles through three stages of behavior: thinking, hungry and eating. To transition from the hungry to eating stage, a philosopher must gain control of the chopsticks to his immediate left and right. This is therefore a problem of contention for shared resources and can be solved by associating a binary semaphore with each chopstick. To avoid deadlock, we insist that odd-numbered philosophers first gain control of the chopstick on the left and subsequently the chopstick on the right. For even-numbered philosophers we require just the opposite: first gain control of the chopstick on the right and then the one on the left.

Appendix: A contains our VPL encoding of this protocol along with the C++ code generated from running `vpl2cxx` on this specification. Note the similarity of the C++ code to the VPL specification. The results of applying the CWB-NC to the VPL specification are given in Figure 17. In particular, the CWB-NC's model checker for the modal mu-calculus reports that the specification satisfies the temporal-logic formula `AG <->tt` meaning that from every reachable state in the specification's underlying state space, a transition is possible (i.e. the system is deadlock-free). The CWB-NC took less than 2 seconds of CPU time to make its determination and found that the state space consisted of 270 states and 918 transitions.

To assess the performance of the C++ code generated by `vpl2cxx` for dining philosophers, we instrumented it with code that generates random timing delays for the thinking and

Table 2
Performance comparison between C++ implementations of dining philosophers.

	C++ from web		`vpl2cxx` (randomized)		`vpl2cxx` (deterministic)	
Phil. Id	No. Meals	Avg. Wait	No. Meals	Avg. Wait	No. Meals	Avg. Wait
1	353	1.06705	437	0.51016	419	0.541981
2	361	0.904072	407	0.759533	393	0.738142
3	401	0.744364	404	0.776832	403	0.759603
4	407	0.723563	373	0.963298	376	0.89
5	423	0.53539	378	0.927725	366	0.951749

eating phases, and computes, for each philosopher, the total number of meals eaten and the average time (in seconds) spent in the hungry (waiting) phase. Table 2 contains the results of our benchmarking activity for the C++ code taken off the web, and for two versions of the C++ code generated by `vpl2cxx`: the first of these implements the randomized algorithm for input/output guard scheduling; the second is for the deterministic algorithm. Since there are no output guards in the VPL specification, both versions may be used.

All results were obtained running Linux Kernel version 2.2.19 on a 700Mhz Pentium III machine with 512MB of RAM. Each program was allowed to run for 1,000 seconds. The results show that there is essentially no difference in performance between the C++ code taken of the web and the two versions of the C++ code generated by `vpl2cxx`.[3]

6.2 *The RETHER Case Study*

Rether is a software-based real-time Ethernet protocol originally developed at SUNY Stony Brook and now sold commercially at RETHER Networks, Inc. (RNI01, 2001). The purpose of this protocol is to provide guaranteed bandwidth and deterministic, periodic network access to multimedia applications over commodity Ethernet hardware. It is designed as a contention-free token bus protocol for the datalink layer of the ISO protocol stack, running on top of a CSMA/CD physical layer. In (Du et al., 1999), we modeled RETHER in VPL and verified, for a particular network configuration, that the protocol indeed makes good on its bandwidth guarantees to real-time nodes without exposing non real-time nodes to the possibility of starvation.

To assess the feasibility of our approach to automatically generating distributed prototypes from VPL specifications, we ran `vpl2cxx` on the VPL specification of a four-node RETHER network given in (Du et al., 1999). The results were very encouraging. The resulting C++

[3] Due to the absence of output guards in the VPL specification for dining philosophers, little difference, if any, was expected between the two `vpl2cxx`-generated versions. The results obtained confirm this expectation.

Tool	Specification Notation	Graphical?	Target Language
ACE (Bosco et al., 1997)	TINA	Yes	CORBA/C++
CAPS (Luqi and Ketabchi, 1998)	PSDL	Yes	Ada/C++/C
IOA (?)	I/O Automata	No	Java
Open Cæsar (Garavel, 1998)	LOTOS	Yes	C
PEP (Grahlmann, 1999)	Petri Nets	Yes	C
Promela++ (Basu et al., 1998)	Promela	No	C
Real-Time Workshop (The MathWorks Inc., 2001a)	Simulink	Yes	C/Ada
Stateflow Coder (The MathWorks Inc., 2001b)	Stateflow	Yes	C
Statemate (Harel et al., 1990)	Statecharts	Yes	C/Ada
SYROCO (Sibertin-Blanc, 2001)	high-level Petri Nets	Yes	C++
`vpl2cxx`	VPL	No	C++

Table 3
Specification & Verification tools supporting code generation.

code is highly readable, bearing a close resemblance to the VPL specification from which it derives. It is fully distributed and runs over TCP/IP.

To gauge the performance of the generated code, we inserted a counter and timer into the C++ code for `node0`, measuring the intervals between receiving the token. The resulting token cycle-time was determined to be 75ms (averaged over 3,000 cycles) on a 10M-bps Ethernet with each node running the SunOS 5.6 operating system. This is approximately 10 times the value observed for the actual RETHER protocol running on a native execution platform. The primary reason for the increased cycle rotation time can be attributed to the fact that the version of RETHER implemented by the `vpl2cxx`-generated C++ code runs in user mode, as opposed to kernel mode as is the case for the actual RETHER protocol.

7 Related Work

A number of specification and verification tool suites provide some form of code generation, differing in terms of the specification language supported, the kinds of analyses allowed on specifications, the target programming language, and intended execution platform of the generated code (e.g. sequential, shared-memory, distributed). Table 3 gives a representative sampling of such tools. (The column entitled “Graphical?” in Table 3 indicates for each tool whether the specification notation supported by the tool is graphical in nature.) Further references can be found in the survey article (Hasselbring, 2000).

Three recent approaches to rapid system prototyping share our methodological goal of being able to generate implementations of distributed systems from formal specifications (Regep and Kordon, 2001; Navarre et al., 2001; Chachkov and Buchs, 2001). All three support an object-oriented approach to system prototyping and use some variant of Petri Nets as the underlying formal modeling technique. The approach of (Regep and Kordon, 2001) is linked to a UML-based methodology, while (Navarre et al., 2001) targets highly interactive applications such as air traffic control systems; (Chachkov and Buchs, 2001) targets embedded controllers. The approach of (Navarre et al., 2001) does not support the generation of code per se, but rather allows the user to associate graphical rendering methods with Petri Net transitions and state changes.

8 Conclusions

We have presented `vpl2cxx`, a translator that automatically generates distributed C++ prototypes from validated VPL specifications. The translator is built around a library for synchronous communication and nondeterministic selection of communication guards, resulting in generated code that is readable and portable. The library includes implementations of two new client/server-based algorithms for the input/output guard scheduling problem: a deterministic protocol for VPL specifications in which output statements do not appear as guards, and a randomized protocol for VPL specifications in which this restriction is lifted.

In terms of the correctness of our guard-scheduling algorithms, we have verified a number of (finite-state) instances of a VPL encoding of both protocols using the CWB-NC's equivalence and model-checking facilities. For future work we would like to obtain general correctness proofs, not limited to any particular instances, of both protocols.

Regarding future work, we are interested in extending `vpl2cxx` to code generation for hard real-time systems, a class of distributed applications of growing importance. The ZEN (Klefstad et al., 2002) open-source real-time CORBA ORB (Object Request Broker), which is inspired by the ACE-based ORB, TAO (Schmidt et al., 1998), appears to be a well-suited middleware platform of choice for such an extension.

8.0.0.6 Acknowledgments. We would like to thank the anonymous referees for their valuable comments and Yuh-Jzer Joung for answering our questions about the complexity of the guard-scheduling problem. This research was supported in part by the National Science Foundation, grants CCR-9505562 and CCR-9988155; the Army Research Office, grants DAAD190110003 and DAAD190110019; and the Office of Naval Research, grant ONR N000140110967 (DOD University Research Initiative (URI) award).

References

Appel, A. W., MacQueen, D. B., Aug. 1991. Standard ML of New Jersey. In: Proc. of Third Int'l Sump. on. Prog. Lang. Implementation and Logic Programming. Springer-Verlag, pp. 1–13.

Basu, A., Morrisett, G., von Eieken, T., Jan. 1998. Promela++: A language for constructing correct and efficient protocols. In: Proc. of IEEE INFOCOMM'98.

Bosco, B. G., Martini, G., Giudice, D. L., Moiso, C., Mar. 1997. An environment for specifying, developing and generating TINA services. In: IFIP/IEEE Int'l Symp. on Integrated Network Management.

Chachkov, S., Buchs, D., Jun. 2001. From an abstract object-oriented model to a ready-to-use embedded system controller. In: Dollas and Wills (2001), pp. 142–148.

Cleaveland, R., Madelaine, E., Sims, S., May 1995. Generating front-ends for verification tools. In: Brinksma, E., Cleaveland, R., Larsen, K., Steffen, B. (Eds.), Tools and Algorithms for the Construction and Analysis of Algorithms (TACAS '95). Vol. 1019 of Lecture Notes in Computer Science. Springer-Verlag, Aarhus, Denmark, pp. 153–173.

Cleaveland, R., Sims, S., Jul. 1996. The NCSU Concurrency Workbench. In: Alur, R., Henzinger, T. A. (Eds.), Computer Aided Verification (CAV '96). Vol. 1102 of Lecture Notes in Computer Science. Springer-Verlag, New Brunswick, New Jersey, pp. 394–397.

Dollas, A., Wills, L. (Eds.), Jun. 2001. 12th IEEE International Workshop on Rapid System Prototyping (RSP 2001). IEEE Computer Society Press, Monterey, CA.

Du, X., Smolka, S. A., Cleaveland, R., Nov. 1999. Local model checking and protocol analysis. Software Tools for Technology Transfer 2 (3), 219–241.

Francez, N., Rodeh, M., 1980. A distributed abstract data type implemented by a probabilistic communication scheme. In: Proceedings of 21st Symposium on Foundations of Computer Science. pp. 373–379.

Garavel, H., 1998. OPEN/CÆSAR: An open software architecture for verification, simulation, and testing. In: Proceedings of the Fourth International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '98), Lecture Notes in Computer Science. pp. 68–84.

Grahlmann, B., Jan. 1999. The State of PEP. In: A. M. H. (Ed.), Proc. of AMAST'98. Vol. 1548 of Lecture Notes in Computer Science. Springer-Verlag.

Hansel, D., May 2000. Generating C++ Code from VPL Specifications. Master's thesis, Technical University of Munich.

Harel, D., Lachover, H., Naamad, A., Pnueli, A., Politi, M., Sherman, R., Shtull-Trauring, A., Traktenbrot, M., Apr. 1990. STATEMATE: A working environment for the development of complex reactive systems. IEEE Transactions on Software Engineering 16 (4), 403–414.

Hasselbring, W., Mar. 2000. Programming languages and systems for prototyping concurrent applications. ACM Computing Surveys 32 (1), 43–79.

ITU-TS, 1996. ITU-TS Recommendation Z.120. Message Sequence Charts (MSC).

Jones, G., 1989. Programming in Occam 2, 2E. Prentice Hall.

Joung, Y.-J., Jul. 2000. Two decentralized algorithms for strong interaction fairness for systems with unbounded speed variability. Theoretical Computer Science 243 (1-2), 307–338.

Klefstad, R., Schmidt, D. C., O'Ryan, C., 2002. Towards highly configurable real-time object request brokers. In: Proceedings of the Fifth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing.

Luqi, Ketabchi, M., Mar. 1998. A computer-aided prototyping system. IEEE Software , 66–72.

Milner, R., 1989. Communication and Concurrency. International Series in Computer Science. Prentice Hall.

Navarre, D., Palanque, P., Bastide, R., Sy, O., Jun. 2001. A model-based tool for interactive prototyping of highly interactive applications. In: Dollas and Wills (2001), pp. 136–141.

Regep, D., Kordon, F., Jun. 2001. LfP: A specification language for rapid prototyping of concurrent systems. In: Dollas and Wills (2001), pp. 90–96.

RNI01, 2001. RETHER Networks, Inc. Web Site. `http://www.rether.com/`.

Schmidt, D. C., Huston, S. D., 2002. C++ Network Programming: Mastering Complexity Using ACE and Patterns. Addison-Wesley Longman.

Schmidt, D. C., Levine, D. L., Mungee, S., apr 1998. The design and performance of real-time object request brokers. Computer Communications 21, 294–324.

Sibertin-Blanc, C., 2001. Cooperative objects: Principles, use and implementation. In: Agha, G., De Cindio, F. (Eds.), Concurrent Object-Oriented Programming and Petri Nets. Lecture Notes in Computer Science. Springer-Verlag, pp. 216–246.

Taft, S. T., Duff, R. A. (Eds.), 1997. Ada 95 Reference Manual: Language and Standard Libraries, International Standard ISO/IEC 8652:1995(E). Lecture Notes in Computer Science, vol. 1246, Springer-Verlag, iSBN 3-540-63144-5.

The MathWorks Inc., 2001a. `http://www.mathworks.com/products/rtw/`.

The MathWorks Inc., 2001b. `http://www.mathworks.com/products/sfcoder/`.

A VPL Source and Generated C++ Code for Dining Philosophers

A.1 VPL Source

```
network diningphil()

begin
process philosopher1(left_obtain      : synch,
                     left_release     : synch,
                     right_obtain     : synch,
                     right_release    : synch)
begin
        while (true) do
                {Eating process - Obtaining chopsticks}

                { odd numbered philosophers   - first right, then left}
                right_obtain!*;
                left_obtain!*;

                {Release process - releasing chopsticks}
                left_release!*;
                right_release!*
         end
end;  {===Specifications of four other philosohpers similar===}

process chopstick  (obtain        : synch,
                    release       : synch )
begin
   while (1=1)  do
         obtain?*;
         release?*
   end
end;

channel obtain1  : synch
channel release1 : synch
channel obtain2  : synch
channel release2 : synch
channel obtain3  : synch
channel release3 : synch
channel obtain4  : synch
channel release4 : synch
channel obtain5  : synch
channel release5 : synch
```

```
  philosopher1(obtain2,release2,obtain1,release1)
| philosopher2(obtain3,release3,obtain2,release2)
| philosopher3(obtain4,release4,obtain3,release3)
| philosopher4(obtain5,release5,obtain4,release4)
| philosopher5(obtain1,release1,obtain5,release5)
| chopstick(obtain1,release1)
| chopstick(obtain2,release2)
| chopstick(obtain3,release3)
| chopstick(obtain4,release4)
| chopstick(obtain5,release5)

end;
```

*A.2 Translated Process "**diningphil**"*

```
//============================================================================
// diningphil.cxx
//
// Generated by vpl2cxx. Source file was 'diningphil.vpl'.
// Mon Oct  1 08:37:57 2001
//============================================================================

#include <ace/SString.h>

void exec( int background, char *proc, ... )
{
  // construct command
  ACE_CString cmd = proc, *arg;

  va_list argp;
  va_start(argp, proc);

  do
    {
      arg = va_arg(argp, ACE_CString *);
      if( arg ) cmd += ACE_CString(" ") += *arg;
    }
  while( arg );

  // execute command
  if( background )
    {
      if( ACE_OS::fork()== 0 )
        {
          system( cmd.c_str() );
        exit(0);
      }
    }
  else
    system( cmd.c_str() );
}

int main (int argc, char *argv[])
{
  // get switches

  ACE_CString switches;

  int n=1, go=1;
  while( (n < argc) && go )
    if( argv[n][0] != '-' )
      go = 0;
    else
      switches += ACE_CString(" ") += ACE_CString(argv[n++]);

  if( argc < n+1 )
    {
      cout << "argument missing" << endl;
      return -1;
    }

  // get scope

  ACE_CString scope;
  if( argv[n][0] != 0 ) scope += ACE_CString( argv[n] ) += ACE_CString(".");
  scope += ACE_CString("diningphil");

  // get passed channel-names


  // construct names of internal channels

  ACE_CString _obtain1 = scope; _obtain1 += ".obtain1";
  ACE_CString _release1 = scope; _release1 += ".release1";
  ACE_CString _obtain2 = scope; _obtain2 += ".obtain2";
  ACE_CString _release2 = scope; _release2 += ".release2";
  ACE_CString _obtain3 = scope; _obtain3 += ".obtain3";
  ACE_CString _release3 = scope; _release3 += ".release3";
  ACE_CString _obtain4 = scope; _obtain4 += ".obtain4";
  ACE_CString _release4 = scope; _release4 += ".release4";
  ACE_CString _obtain5 = scope; _obtain5 += ".obtain5";
  ACE_CString _release5 = scope; _release5 += ".release5";
  // start subsystems

  exec(1, "diningphil_philosopher1", &switches, &scope, &_obtain2, &_release2, &_obtain1, &_rel
  exec(1, "diningphil_philosopher2", &switches, &scope, &_obtain3, &_release3, &_obtain2, &_rel
  exec(1, "diningphil_philosopher3", &switches, &scope, &_obtain4, &_release4, &_obtain3, &_rel
  exec(1, "diningphil_philosopher4", &switches, &scope, &_obtain5, &_release5, &_obtain4, &_rel
  exec(1, "diningphil_philosopher5", &switches, &scope, &_obtain1, &_release1, &_obtain5, &_rel
  exec(1, "diningphil_chopstick", &switches, &scope, &_obtain1, &_release1, NULL);
  exec(1, "diningphil_chopstick", &switches, &scope, &_obtain2, &_release2, NULL);
  exec(1, "diningphil_chopstick", &switches, &scope, &_obtain3, &_release3, NULL);
  exec(1, "diningphil_chopstick", &switches, &scope, &_obtain4, &_release4, NULL);
  exec(1, "diningphil_chopstick", &switches, &scope, &_obtain5, &_release5, NULL);

  // start server for network-internal channels

  exec(0, "server", &switches, &_obtain1, &_release1, &_obtain2, &_release2,
    &_obtain3, &_release3, &_obtain4, &_release4, &_obtain5, &_release5, NULL);
}
```

A.3 Translated Process “*diningphil_philosopher1*”

```
//==========================================================================
// diningphil_philosopher1.cxx
//
// Generated by vpl2cxx. Source file was 'diningphil.vpl'.
// Mon Oct  1 08:37:57 2001
//==========================================================================

#include "Channel.hxx"
#include "Action.hxx"
#include "VPLTypes.hxx"

VPLSynch   _synch;   // used to translate '[channel]?*'
ActionPool _choices; // used to store choices of a 'select'-statement

// process-local type declarations

// channels passed to process
Channel< VPLSynch > left_obtain;
Channel< VPLSynch > left_release;
Channel< VPLSynch > right_obtain;
Channel< VPLSynch > right_release;

// process-local variable declarations

void philosopher1()
{
  while( 1 )
    {
      right_obtain << _synch;
      left_obtain << _synch;
      left_release << _synch;
      right_release << _synch;
    } // while

}

//-------------------------------- main -------------------------------------

int main (int argc, char *argv[])
{
  // parse arguments
  int i=1, go=1;
  while( (i < argc) && go )
    {
       if( strncmp( argv[i], "-", 1 ) != 0 )
         go = 0;
       else if( strncmp( argv[i], "-v", 2 ) == 0 )
         { i++; Global_LogFileWriter.startLogging(atoi(argv[i-1]+2), 0); }
       else if( strncmp( argv[i], "-w", 2 ) == 0 )
         { i++; Global_LogFileWriter.startLogging(atoi(argv[i-1]+2), 1); }
       else if( strncmp( argv[i], "-ns", 3 ) == 0 )
         { i++; ChannelGeneric::setNameServer( ACE_INET_Addr( argv[i-1]+3 ) ); }
       else
         cout << "Warning: Unrecognized switch: " << argv[i++] << endl;
     }

  if( argc < i+5)
    {
      cout << "argument missing" << endl;
      return -1;
    }

  try
    {
      ChannelGeneric::setProcessName(argv[0]);

      // open channels
      left_obtain.open( argv[i+1] );
      left_release.open( argv[i+2] );
      right_obtain.open( argv[i+3] );
      right_release.open( argv[i+4] );

      // run process
      philosopher1();
    }
  catch( SE_SendRec se )
    {}
  catch( SocketException se )
    {
      cout << "Exception: " << se << endl;
    }

  return 0;
}
```

*A.4 Translated Process "**diningphil_chopstick**"*

```
//=========================================================================
// diningphil_chopstick.cxx
//
// Generated by vpl2cxx. Source file was 'diningphil.vpl'.
// Mon Oct  1 08:37:57 2001
//=========================================================================

#include "Channel.hxx"
#include "Action.hxx"
#include "VPLTypes.hxx"

VPLSynch   _synch;   // used to translate '[channel]?*'
ActionPool _choices; // used to store choices of a 'select'-statement

// process-local type declarations

// channels passed to process
Channel< VPLSynch > obtain;
Channel< VPLSynch > release;

// process-local variable declarations

void chopstick()
{
  while( (1==1) )
    {
      obtain >> _synch;
      release >> _synch;
    } // while

}

//-------------------------------- main -----------------------------------

int main (int argc, char *argv[])
{
  // parse arguments
  int i=1, go=1;
  while( (i < argc) && go )
    {
       if( strncmp( argv[i], "-", 1 ) != 0 )
         go = 0;
       else if( strncmp( argv[i], "-v", 2 ) == 0 )
         { i++; Global_LogFileWriter.startLogging(atoi(argv[i-1]+2), 0); }
```

```
        else if( strncmp( argv[i], "-w", 2 ) == 0 )
          { i++; Global_LogFileWriter.startLogging(atoi(argv[i-1]+2), 1); }
        else if( strncmp( argv[i], "-ns", 3 ) == 0 )
          { i++; ChannelGeneric::setNameServer( ACE_INET_Addr( argv[i-1]+3 ) ); }
        else
          cout << "Warning: Unrecognized switch: " << argv[i++] << endl;
      }

  if( argc < i+3)
    {
      cout << "argument missing" << endl;
      return -1;
    }

  try
    {
      ChannelGeneric::setProcessName(argv[0]);

      // open channels
      obtain.open( argv[i+1] );
      release.open( argv[i+2] );

      // run process
      chopstick();
    }
  catch( SE_SendRec se )
    {}
  catch( SocketException se )
    {
      cout << "Exception: " << se << endl;
    }

  return 0;
}
```