Distributed Prototyping from Validated Specifications

David Hansel
Reactive Systems
120-B East Broad Street
Falls Church, VA 22046, USA
hansel@reactive-systems.com

Rance Cleaveland   Scott A. Smolka
SUNY at Stony Brook
Department of Computer Science
Stony Brook, NY 11794, USA
{rance, sas}@cs.sunysb.edu

Abstract

We present `vpl2cxx`*, a translator that automatically generates efficient, fully distributed C++ code from high-level system designs specified in the mathematically rigorous VPL design language. The Concurrency Workbench of the New Century (CWB-NC) verification tool includes a front end for VPL, and this means designers may use the full range of automatic verification and simulation checks provided by the tool 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 (RTOSes). 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 ACE client-server network programming interface. Finally, benchmarks show that the performance of the generated code is more than acceptable for a distributed prototype. We discuss one such example in the RETHER real-time ethernet protocol for voice and video applications.*

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 (RTOSes). Even if the code produced was not production-quality, it would at least function as a serviceable *rapid system prototype*.

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 [15]. VPL specifications can be processed using the front-end generated generated by the Process Algebra Compiler (PAC) [4] for the Concurrency Workbench of the New Century (CWB-NC, formerly the NCSU Concurrency Workbench) [5]. The automatic verification and simulation 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 for correctness. The tool described in the current paper may be used to generate C++ prototypes automatically from the analyzed designs.

Our code generator, named `vpl2cxx`, utilizes the ACE (ADAPTIVE Communication Environment) network programming interface [17] so that the code it produces is portable to a host of RTOSes. `vpl2cxx` also implements an efficient, randomized solution to problem of scheduling input and output guards in a distributed system. Therefore, there are 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—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++ manually.

To assess the feasibility of the proposed approach, we have applied `vpl2cxx` to several examples, including the

RETHER real-time ethernet protocol for voice and video applications [16]. The code produced by `vpl2cxx` for RETHER is fully distributed, highly readable, and 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 [4] and [5], respectively.

In related work, a number of specification and verification tool suites provide some form of code generation, differing in terms of the specification language supported and the language and intended platform (e.g. sequential, shared-memory, distributed, etc.) for the generated code. Examples include: Statemate [12], generating C/Ada from Statecharts; CAPS [14], generating Ada/C++/C from graphical PSDL designs; SYROCO [18], generating C++ code from high-level Petri Nets; Open Cæsar [8], generating C from LOTOS specifications; ACE (Application Construction Environment), generating CORBA and C++ code from TINA models [3]; PEP, generating C code from Petri nets [10]; Promela++, generating C code from Promela specifications [2]; and the IOA tool [9], generating Java from I/O Automata.

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 RETHER case study is the subject of Section 5 while Section 6 offers our concluding remarks. For a fuller account of the code-generation work described here, the reader is referred to Hansel's Diplomarbreit thesis [11].

2 VPL

VPL is designed for specifying 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 synchronization of processes. Message passing in VPL is binary and synchronous, requiring a handshake between the sender and receiver for communication to occur. The language also has a formal operational semantics; thus, VPL specifications are candidates for model checking and other kinds of formal analysis.

The basic structure of a VPL specification is given by system declarations. A system is either a network, defining a collection of subsystems and their connections, or a process, consisting of actual code. 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 within the body of the subsystem. Actual channels must be type-consistent with the formal ones introduced in the header and are declared either as local channels or formal channels of the network immediately containing the subsystem.

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 its 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 nondeterministic choice in the form of the `select` statement.

To illustrate the language, the VPL specification given in Figure 1 defines a simple network consisting of the parallel composition of three subsystems: processes `sender` (lines 5–17), `receiver` (lines 18–24), and `buffer` (lines 25–32). Process `sender` communicates with process `buffer` over channel `c1` while `buffer` and `receiver` communicate over channel `c2`.

3 Library for Synchronous Communication

This section describes the library for synchronous communication and nondeterministic selection of communication guards 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 C++ are asynchronous. The second problem is the presence of input and output guards in VPL, i.e. send and receive statements appearing as the first statements in `select` statement alternatives. In such a setting, obtaining a symmetric,

```
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( inChannel : t,
                  outChannel: t)
  begin
    var buf : t
    while (1=1)
       inChannel?buf;
       outChannel!buf
    end
  end;

  Channel c1 : t
  Channel c2 : t

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

Figure 1. An example VPL specification.

fully distributed, deterministic solution to the *input/output guard-scheduling problem* is known to be impossible [7].

Additionally, our implementation was guided by the following principles.

- *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 stand-alone, 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 the choice of the ACE library (see Section 3.3) as the portable 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.

3.1 Protocol for Input/Output Guard Scheduling

The protocol implemented in the communication library for scheduling input/output guards relies on randomized polling and is inspired by the randomized algorithms of [7, 13]. Such algorithms achieve a notion of "correctness (freedom from deadlock and starvation) with probability 1".

The protocol is client/server-based and 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. Communication in VPL is channel-based and the implementation of the protocol uses a server process for each channel. 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.

The server, on the other hand, maintains a *pool* of pending requests—some of which may have come from clients attempting a communication in a non-guard position, and thus willing to wait "indefinitely"—and 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 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. 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 often, then some communication will eventually take place with probability 1 [7].

3.2 The API

The API of the synchronous communication library exports the following methods and classes:

- Create a name server. 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.
- Create a channel server. Upon creation, the server opens a free port on which it will listen for requests, 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 `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.
- 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.
- Transform types. Although channels in VPL have a designated type, no type is assigned to a started 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 `VPLTypes` class.

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

- The *process simulation tool* allows one to connect to a name server, retrieve a list of all channel servers, and connect to them. The user can then interact with the C++ program, writing communication actions as they would appear in a VPL `select` statement.
- 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) 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 file containing a graphical rendering of the events recorded in one or more log files.

3.3 The ACE Network Programming Interface

To render the communication library highly portable, we have based its implementation on the ACE (ADAPTIVE Communication Environment) library [17]. 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, required 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 Concurrency Workbench, is implemented in Standard ML using the *Standard ML of New Jersey* compiler [1]. Moreover, it re-uses the Workbench's routines for parsing VPL, which were automatically generated by the PAC [4]. The VPL parsing routines produce a tree of SML structures which the translator recursively processes to generate the target C++ files.

5 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. [16]. 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 [6], 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 `vpl2cc` on the VPL specification of a four-node RETHER network given in [6]. The results were very encouraging. The resulting C++ 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.

6 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.

Due to the randomized nature of the guard-scheduling algorithm implemented in the library, message complexity can become an issue. For a VPL `select` statement having k alternatives, the expected number of messages needed to achieve communication is $O(k^2)$. Alternatively, one can restrict the use of nondeterminism at the VPL level, disallowing, for example, output guards. Languages such as Ada 95 and occam are designed in this manner. In this case, an algorithm for guard scheduling can be devised whose average message complexity is linear in the number of `select` alternatives. Indeed, we have produced an alternative implementation of the synchronous communication library based on this approach.

Regarding the correctness of the guard-scheduling algorithms, we have specified them in VPL and used the Workbench's model checker for the modal mu-calculus to analyze several (finite-state) instances, the largest consisting of approximately 10,00 states and 34,000 transitions. We have also used the communication library's simulator and logging tools to monitor the run-time behavior of the generated code for any discrepancies in behavior. A goal of our future work will be to obtain correctness proofs of the general algorithms.

Acknowledgements. 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, and the Army Research Office, grants DAAD190110003 and DAAD190110019.

References

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

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

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

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

[5] R. Cleaveland and S. Sims. The NCSU Concurrency Workbench. In R. Alur and T. A. Henzinger, editors, *Computer Aided Verification (CAV '96)*, volume 1102 of *Lecture Notes in Computer Science*, pages 394–397, New Brunswick, New Jersey, July 1996. Springer-Verlag.

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

[7] N. Francez and M. Rodeh. A distributed abstract data type implemented by a probabilistic communication scheme. In *Proceedings of 21st Symposium on Foundations of Computer Science*, pages 373–379, 1980.

[8] H. Garavel. OPEN/CÆSAR: An open software architecture for verification, simulation, and testing. Technical Report 3352, INRIA Rhône-Alpes, France, Jan. 1998.

[9] S. J. Garland and N. A. Lynch. The IOA language and toolset: Support for designing, analyzing, and building distributed systems. Technical Report MIT/LCS/TR-762, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA, 1998.

[10] B. Grahlmann. The State of PEP. In M. H. A. editor, *Proc. of AMAST'98*, volume 1548 of *Lecture Notes in Computer Science*. Springer-Verlag, Jan. 1999.

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

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

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

[14] Luqi and M. Ketabchi. A computer-aided prototyping system. *IEEE Software*, pages 66–72, Mar. 1998.

[15] R. Milner. *Communication and Concurrency*. International Series in Computer Science. Prentice Hall, 1989.

[16] Rether networks, inc. web site. http://www.rether.com/.

[17] D. C. Schmidt. The ADAPTIVE Communication Environment: An Object-Oriented Network Programming Toolkit for Developing Communication Software. In *Proceedings of the 12^{th} Annual Sun Users Group Conference*, pages 214–225, San Francisco, CA, June 1994. SUG.

[18] C. Sibertin-Blanc. Cooperative objects: Principles, use and implementation. In G. Agha and F. De Cindio, editors, *Concurrent Object-Oriented Programming and Petri Nets*, Lecture Note in Computer Science. Springer-Verlag, 1998.