POSTER PAPER

Synthesis of C++ Software for Automated Teller from CSPm Specifications

Stephen Doxsee
Dept. of Computing & Information Science
University of Guelph
Guelph, Ontario, Canada
1(416)620-9783
sdoxsee@uoguelph.ca

W. B. Gardner
Dept. of Computing & Information Science
University of Guelph
Guelph, Ontario, Canada
1(519)824-4120 x52696
wgardner@cis.uoguelph.ca

ABSTRACT
CSP++ is an object-oriented application framework for execution of CSP specifications that have been automatically translated into C++ source code by a tool called **cspt**. This approach makes CSP specifications directly executable, and extensible via the ability to incorporate user-coded functions. Designers can exploit "selective formalism" to code some system functionality in CSP for formal verification purposes, and other functionality directly in C++. The translator has now been enhanced to accept input in CSPm syntax, the same dialect processed by the commercial verification tool, FDR2, and we demonstrate this with a new ATM case study.

Categories and Subject Descriptors
D.1.2 [**Programming Techniques**]: Automatic Programming—*Program synthesis, Program verification;* C.0 [**General**] Systems specification methodology; D.1.3 [**Programming Techniques**]: Concurrent Programming; D.2.1 [**Software Engineering**]: Requirements/Specifications—*Languages, Tools, CSP, C++;* D.2.2 [**Software Engineering**]: Design Tools and Techniques; D.2.4 [**Software Engineering**]: Software/Program Verification —*Formal methods, Model checking;* D.2.10 [**Software Engineering**]: Design—*Methodologies;* I.6.5 [**Simulation and Modeling**]: Model Development—*Modeling methodologies.*

General Terms
Design, Languages, Verification.

Keywords
Executable specifications, Object-oriented Application Frameworks.

1. INTRODUCTION
Formal methods have yet to achieve any great impact on typical software engineering practices. As a middle ground between pure formal methods and conventional software programming, we are advocating a technique called "selective formalism." It attempts to capitalize on both formal methods and traditional software practices by making formal specifications both *executable* and *extensible*. By selectively choosing to formally specify only the critical control portions of a system, one can synthesize such a formal specification into executable code for a popular programming language and allow other programmers to write modular extensions that will tie into the synthesized control backbone. Selective formalism requires three main ingredients: (i) a suitable formal notation that preferably has verification tool support; (ii) a popular programming language; and (iii) some type of framework to tie them together. The synthesis of executable code from formal specifications should be done automatically because of the errors that can be introduced by hand translation and the time it would take.

We chose the process algebra CSP [5] as the formally verifiable notation, C++ as our target programming language, and built an integrating framework dubbed CSP++. CSP statements can be used to model a concurrent system's control and data flow in an intuitive way, constituting a kind of hierarchical behavioral specification. It is supported by sophisticated commercial tools such as FDR2 and ProBE from Formal Systems (Europe) Limited (http://www.fsel.com).

CSP++ development has been underway for several years [2,3,4] based on translating a local dialect of CSP called csp12. Here we demonstrate, by means of an ATM case study, a new front-end to the CSP++ translator, cspt, that supports CSPm syntax. The upgraded translator allows CSP specifications verified by FDR2 to be directly translated to C++.

2. CSPM AND SYNTHESIZED C++
In brief, each statement in a CSP specification is the description of a *process*. The process engages in a sequence of named *events*, which may include point-to-point communication with another process via a nonbuffered, unidirectional *channel*. The set of all events that a process may ever engage in is called its *alphabet*. These may correspond to real-world occurrences such as sensor input, device actuation, and so on. Processes can define themselves in terms of other processes, including several processes running in parallel. Then, the formalism provides for interprocess synchronization each time an event occurs that is in

SAC'05, March 13-17, 2005, Santa Fe, New Mexico, USA.

their common alphabet. This includes synchronization around channel communication.

A simple CSPm specification illustrates the CSP++ translator:

```
channel p,q,r,s,z
A = p->q->z->p->SKIP
B = r->s->z->r->SKIP
SYS = (A [|{z}|] B)
```

This C++ code will be generated by cspt for the SYS process:

```
static ActionRef z_r( z );
AGENTPROC( SYS_ )
   z_r.sync();
   {
      Agent::compose( 2 );
      Agent* a1 = START0( A_, 0 );
      Agent* a2 = START0( B_, 1 );
      WAIT( a1 );
      WAIT( a2 );
   }
   Agent::popEnv( 1 );
   END_AGENT;
}
```

Objects z_r, a1, and a2 are instances of classes in the CSP++ framework which supply CSP execution semantics. The code begins by registering z for synchronization, and anticipating a two-process composition. Then it starts the processes, waits for them to finish, and finally pops the z synchronization off the environment stack. A and B will run their initial events in no particular order until they reach their respective z events. Once one process arrives at z, it waits for the next one. Then they perform z together and continue separately to their conclusions (SKIP).

CSP++ implements concurrent processes using the thread model of GNU Pth (http://www.gnu.org/software/pth/). Pth is a portable POSIX/ANSI-C library that provides non-preemptive scheduling.

3. ATM CASE STUDY

To illustrate the use of CSP++ with translation of CSPm input, we have implemented a small and easy-to-understand system, an Automated Teller Machine (ATM), based on a design by R. Bjork [1], whose requirements, analysis, and design were used as a starting point.

3.1 CSP in the Design Cycle

CSP notation can be utilized in four roles in the system design cycle, constituting four complementary models: (1) a Functional Model that captures the desired system behavior in terms of CSP processes engaging in named events; (2) an Environmental Model that simulates the behavior of entities in the system's target environment, also in terms of processes engaging in events; (3) an optional Constraint Model used to limit or constrain the event sequences of the functional model; and (4) an Implementation Model that completes, in CSP, details unspecified in the initial functional model.

A functional model can be derived by various means, including English system descriptions or Statecharts. The ATM Statecharts translated readily into high level functional CSP descriptions of the overall behavior of the system. The resulting ATM process was simulated by making environmental model processes CLIENT, OPERATOR, and BANK run concurrently with it. A constraint model was unnecessary in this particular system, but the details missing from the high level Statecharts (e.g., how to handle invalid PINs) were added to the final implementation model.

3.2 Verification and Functional Testing

There are some system properties that FDR2 can check on its own with no special instructions, including the absence of deadlock, livelock (divergence), and non-determinism. In our study, we also used trace and failures refinement to prove functional test cases (such as showing that a client's card will indeed be held after entering the wrong PIN three times in a row).

3.3 User-coded Function Integration

When the environmental model is removed, the implementation model, synthesized into executable C++, is free to interact with its real environment (i.e. keypads, network connections, etc.) via CSP events that are linked to user-coded C++ functions (UCF). In this way, the ATM program reads PINs, displays balances, establishes network connections, and so on.

3.4 Debugging

The C++ code generated by cspt is debuggable by GNU gdb. The C++ source code closely corresponds to the statements of the input CSP specification, making it easy to follow. One may also set breakpoints in and step through the statements of user-coded C++ functions.

4. CONCLUSIONS

We presented a new front-end to the CSP++ translator that supports CSPm syntax and demonstrated it using an ATM case study. We believe that selective formalism is an attractive approach to concurrent system design and development because of its flexible combination of formal verification and conventional programming. The latest version of CSP++ and the code for the ATM case study can be downloaded from the author's website (http://www.cis.uoguelph.ca/~wgardner).

5. REFERENCES

[1] Bjork, R. C. *An Example of Object-Oriented Design: An ATM Simulation.* http://www.math-cs.gordon.edu/local/courses/cs211/ATMExample/.

[2] Gardner, W. B. Bridging CSP and C++ with Selective Formalism and Executable Specifications. In *First ACM & IEEE International Conference on Formal Methods and Models for Co-design* (MEMOCODE '03), Mont St-Michel, France, June 2003, p. 237-245.

[3] Gardner, W. B. Converging CSP Specifications and C++ Programming via Selective Formalism. To appear in *ACM Transactions on Embedded Computing Systems* (TECS), Special Issue on Models & Methodologies for Co-Design of Embedded Systems.

[4] Gardner, W. B. and Serra, Micaela. *CSP++: A Framework for Executable Specifications,* chapter 9. In Fayad, M., Schmidt, D., and Johnson, R., editors. *Implementing Application Frameworks: Object-Oriented Frameworks at Work.* John Wiley & Sons. 1999.

[5] Schneider, Steve. *Concurrent and Real Time Systems: The CSP Approach*, John Wiley & Sons, Inc., New York, NY, 2000.