Rapid Prototyping of Embedded Software Using Selective Formalism

John Carter, Ming Xu, W.B. Gardner
Modeling & Design Automation Group, Dept. of Computing & Information Science
University of Guelph
Guelph, Ontario, Canada
jcarter@uoguelph.ca, mxu@uoguelph.ca, wgardner@cis.uoguelph.ca

Abstract

Our software synthesis tool, CSP++, generates C++ source code from verifiable CSPm specifications, and includes a framework for runtime execution. Our technique of selective formalism allows the synthesized formal control backbone code to be linked with non-formal user-coded C++ functions that carry out I/O and data processing. This tool already facilitates rapid prototyping of formally-specified software by bypassing the customary manual translation from a formal notation. In this work, we extend the rapid prototyping capability to SOPC (system on programmable chip) by targeting the CSP++ execution framework to an FPGA processor core. This is demonstrated with a new point-of-sale case study.

1. Introduction

Advocating formal methods for software specification and development continues to receive a mixed reaction in industry. No doubt some categories of embedded systems, including safety-critical and mission-critical applications, could benefit from using formal verification to help ensure that specifications are met. As well, concurrent systems are subject to pitfalls such as deadlocks that could potentially be detected by verification tools. However, even if one wishes to go the formal route, that would seem to throw up a barrier to rapid prototyping. This is because a formal specification notation is generally not executable, so a time-consuming, error-prone hand translation to a conventional programming language is required. Furthermore, it would not be clear that the resulting hand-translated code retained the verified properties. This situation is exacerbated if the entire software system has to be specified in a formal notation and hand-translated.

Our solution, "selective formalism"—based on making a specification written in the process algebra CSP (Communicating Sequential Processes) [7] [6] executable in C++—is described in the following background section. This approach enables rapid prototyping of a CSP-specified system by eliminating the hand translation step. It has been applied to software running on general purpose computers under Unix variants. We then describe our current work, retargeting the run-time framework (which provides CSP synchronization and communication semantics) to an FPGA processor core. The aim is to enable rapid prototyping of SOPC (system on programmable chip) embedded software. A new case study is presented for illustrative purposes, based on a retail point-of-sale system. Our plans for future work are also outlined.

2. Selective Formalism and CSP++

The "selective" aspect of selective formalism is that formally specified and verified components of a system can be easily mixed with components coded in a conventional programming language (C++). Critical components of a system are specified in CSPm [2] [4] (the machine-readable dialect of CSP) to describe process composition, interaction, and synchronization. A translation tool synthesizes C++ source code from the CSPm statements for execution under the CSP++ runtime framework.

Non-critical behavior is implemented as user-coded functions in C++. These functions are linked to abstract, named CSPm events, and invoked by processes in the executable specification. They could be used for I/O, interfacing to packages such as database management systems, or performing calculations that are awkward to express in CSP, which is not intended to be a full-featured programming language. User-coded functions are subject to some restrictions to prevent them from breaking the formalism. The designer is free to strike an appropriate selective balance between the relative quantity of formal CSP versus non-formal C++, according to the nature of the system under construction.

Figure 1 depicts the CSP++ software synthesis process. A developer creates a CSPm specification, and can use com-

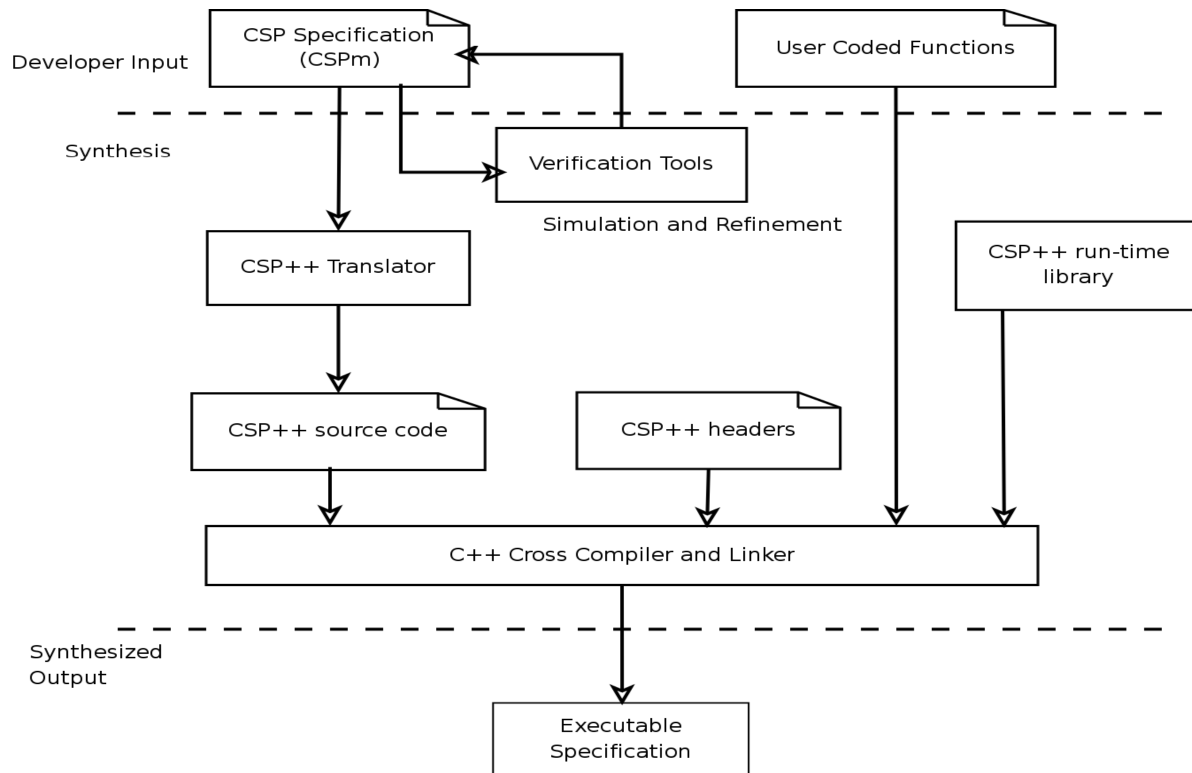

Figure 1. CSP++ Design Flow

mercial tools FDR2 and Probe, from Formal Systems Europe, to scrutinize, simulate, refine, and verify the CSP models. When used correctly, these tools allow the specification that forms the control backbone of the system to be tested for points of failure, and aids in exposing unintended and potentially unsafe behavior, such as deadlock and livelock.

Next, the CSPm specification is automatically translated, generating C++ source code based on the CSP++ object-oriented application framework [5] [3]. Table 1 shows the CSPm constructs presently implemented by the CSP++ translator. Boolean guard (&), replicated operators (@), untimed timeout ([>), and interrupt (/\) are not yet implemented.

The synthesized source code is then compiled with the CSP++ headers, and linked with the CSP++ run-time library and any user-coded functions the developer has added, resulting in an executable specification. The user-coded functions must not "go behind the back" of the CSP specification to carry out their own interprocess communication or synchronization. This insures that the trace of event execution adheres to the original specification.

The current version of CSP++ uses GNU Portable Threads (Pth) to provide nonpreemptible multithreading. It runs on any Unix variant that supports Pth. CSP++ is not wedded to Pth, and has been ported to other threads packages.

Table 1. CSPm constructs currently implemented by CSP++

<table>
<tr><th>Construct</th><th>CSPm Symbol</th></tr>
<tr><td>parallel synchronization</td><td>||</td></tr>
<tr><td>process interleaving</td><td>|||</td></tr>
<tr><td>sequential composition</td><td>;</td></tr>
<tr><td>prefix</td><td>-></td></tr>
<tr><td>channel I/O</td><td>? and !</td></tr>
<tr><td>external choice</td><td>[]</td></tr>
<tr><td>event renaming</td><td><-</td></tr>
<tr><td>event hiding</td><td>\\</td></tr>
<tr><td>conditional</td><td>if-then-else</td></tr>
</table>

3. SOPC Work In Progress

Our goal is to allow CSP++ source code to be cross-compiled, linked, and executed on an SOPC target platform (as shown in figure 2).

At present, we have selected a suitable FPGA device and

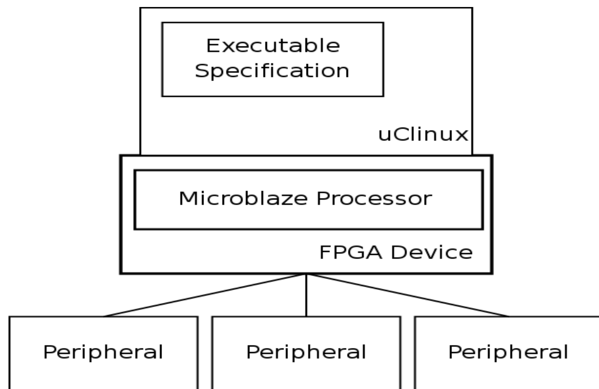

Figure 2. Target Platform

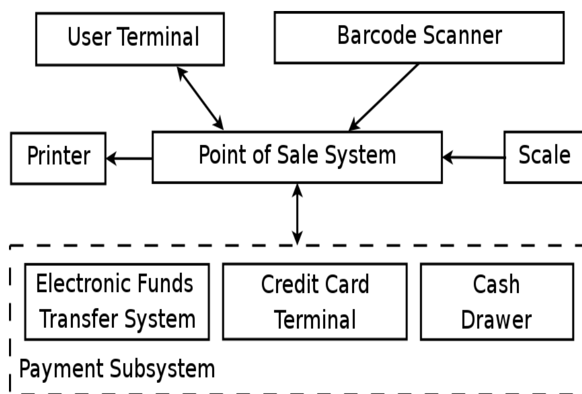

Figure 3. POS Overview

RTOS for our SOPC: the Xilinx MicroBlaze soft processor core configured on a Virtex FPGA device, and uClinux. The MicroBlaze processor is a 32-bit RISC processor architecture that is capable of hosting uClinux [10], a lightweight Linux distribution created for use in embedded devices. We now have a configured and bootable SOPC, with a working build chain (cross-compiler and binary tools).

Current work is focused on getting Pth to coexist happily with uClinux on the MicroBlaze. If this obstacle is insurmountable, we will port CSP++ to another RTOS.

4. POS Case Study

We have created a new case study to illustrate the advantages of rapid prototyping using our technique. An overview of the “point of sale” (POS) system architecture is given in figure 3, showing components of the system and their intercommunication. The system behaviour was first captured at a high level using finite state machines (FSM), shown in figure 4. FSM models are relatively easy to express in CSPm for functional refinement. The CSPm specification was simulated, debugged, and verified.

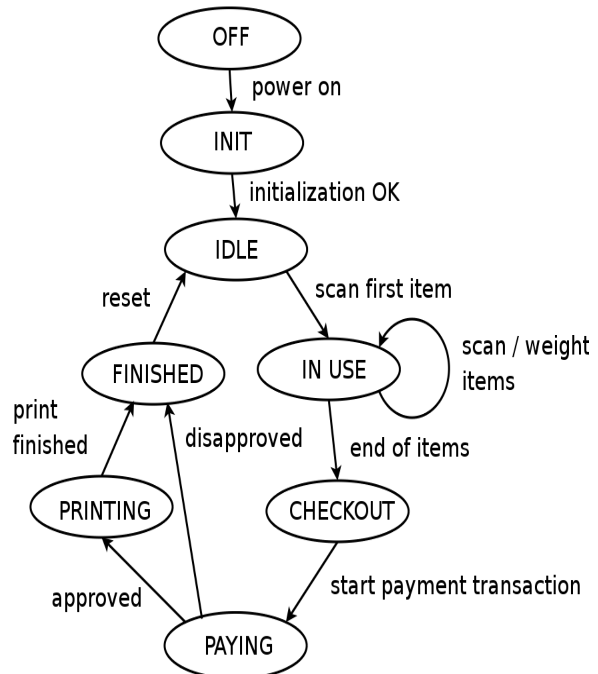

Figure 4. FSM description of system.

A POS terminal was chosen as the first SOPC case study for several reasons. Most people have a high degree of familiarity with these systems. POS systems feature a simple, well-known interface. POS terminals are composed of several interacting control-dominated subsystems. Lastly, POS terminals utilize peripherals such as barcode readers, segment displays, scales, and electronic funds transfer systems. These peripherals are supported by the FPGA development board we are working with.

The CSPm specification for the top-level system process SYS is shown below. It executes the INITIALIZATION process, followed by five concurrent subsystem processes:

```
SYS = INITIALIZATION;
((((TERMINAL [|{| barcode_requested,
    barcode_out |}|] BARCODE)
  [|{| weight_requested,
       weight_out |}|] SCALE)
   [|{| start_job,
        finish_job |}|] PRINTER)
    [|{| amount,
         approve,
         disapprove,
         display_change,
         payment_start |}|] PAYMENT)
```

The INITIALIZATION process is composed of a number of events to be executed sequentially, bring the various

components of the POS system online. “SKIP” is normal process termination.

```
INITIALIZATION = init_terminal ->
  init_scale -> init_printer ->
  init_barcode -> init_eftpos ->
  init_credit -> init_drawer ->
  SKIP
```

As an illustration of one of the concurrent processes, the payment subsystem is in turn composed of a PAYMENTCHOICE process, which determines which payment method is selected, and PAYMENT_SYSTEM, a set of three payment-specific subprocesses used for each of the possible payment methods. PAYMENTCHOICE_CMD provides an example of the CSP concept of parameterized processes. The invocation PAYMENTCHOICE_CMD(option) will be dynamically bound to the correct variant at run time, depending on the value of option.

```
PAYMENT=PAYMENTCHOICE[|{|cash_payment,
  credit_payment,
  debit_payment|}|] PAYMENT_SYSTEM

PAYMENTCHOICE=payment_start ->
  payment_selection?option ->
  PAYMENTCHOICE_CMD(option)

PAYMENTCHOICE_CMD(0) =
   cash_payment->PAYMENTCHOICE
PAYMENTCHOICE_CMD(1) =
   credit_payment->PAYMENTCHOICE
PAYMENTCHOICE_CMD(2) =
   debit_payment->PAYMENTCHOICE

PAYMENTCHOICE_CMD(x)=CANCELPAYMENT

CANCELPAYMENT=amount?trash ->
   display_change!0 ->
   disapprove->PAYMENTCHOICE

PAYMENT_SYSTEM=EFTPOS |||
   CREDITCARDTERMINAL |||
   CASHDRAWER
```

The PAYMENT process uses two of the types of parallelism provided by CSP. *Interface parallel* defines a set of events (enclosed in the [|{| and |}|] symbols) on which the participating processes must synchronize. The PAYMENT_SYSTEM employs *interleaving parallelism*, whereby a number of processes (in this case EFTPOS, CREDITCARDTERMINAL and CASHDRAWER) execute in parallel with no synchronization or communication among them.

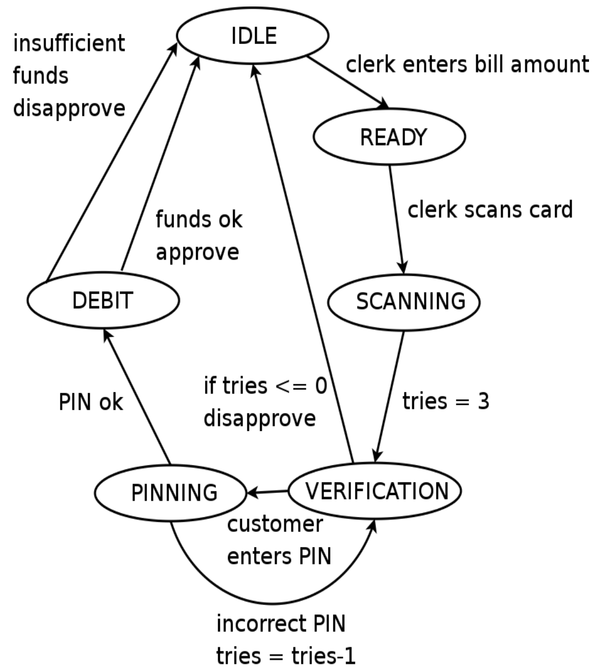

Figure 5. FSM description of EFTPOS portion of payment state.

A further illustration is the EFTPOS process (based loosely on the FSM description shown in figure 5) used by the payment subsystem. Its EFTPOS_VERIFICATION process tries 3 times to verify the customer's PIN. The line starting with “−−” is a comment. EFTPOS_VERIFICATION uses the “if, else” construct to provide conditional behavior, an alternative to defining parameterized processes like PAYMENTCHOICE_CMD above.

```
EFTPOS=debit_payment -> amount?bill ->
   debitcard_scan?card ->
   EFTPOS_VERIFICATION(3,bill,card)
-- (3 is number of pin 'tries' remaining)

EFTPOS_VERIFICATION(n,bill,card) =
   if (n == 0) then display_change!0 ->
     disapprove -> EFTPOS
   else
     pin_entry?pin ->
     EFTPOS_CHECKFUNDSPIN(n,
       bill,
       card,
       pin)

EFTPOS_CHECKFUNDSPIN(n,bill,card,pin) =
   senddebitrequest!card.bill.pin ->
   recvdebitresult?r ->
```

```
   if ( r == 0 ) then
      EFTPOS_VERIFICATION(n-1,
      bill,card)

   else if ( r == 1 ) then
      (display_change!0 ->
      approve -> EFTPOS)

   else (display_change!0 ->
      disapprove -> EFTPOS)
```

At the implementation stage, user-coded functions were added to carry out I/O such as console input and output, to calculate totals, and to simulate operations such as retrieving inventory from a database. The ability to attach user-defined functionality to a verified CSP model makes the selective formalism approach convenient. At present, our case study relies on simple console input and output from a serial terminal. Support will later be added for additional hardware, as appropriate to a realistic POS terminal. The console I/O functions will be replaced with a different set of user-coded functions interfacing with the uClinux drivers running on the MicroBlaze core. Such a change requires only relinking, not a rewriting or re-synthesis of the CSPm specification.

Shown below is an example of a user-coded function to initialize the printer. It would be invoked by the init_printer event of the INITIALIZATION process. The linkage of CSPm events to user-coded functions is provided by the developer at compile time in the makefile.

```
void init_printer_q( ActionType t,
 ActionRef* a,
 Var* v, Lit* l ) {

   printf("init: printer.\r\n");

   // open printer
   fopen("PRN","w");
}
```

Here is a sample user-coded function to send a request for an EFTPOS transaction. This one takes the place of a channel output event, senddebitrequest!card.bill.pin, in EFTPOS_CHECKFUNDSPIN above. The getList() method is used to extract the data from the channel object.

```
void senddebitrequest_q( ActionType t,
 ActionRef* a,
 Var* v, Lit* l )
{
   List<Lit>* temp = l->getList();

   // extract parameters
   int card = int((*temp)[0]);
   int bill = int((*temp)[1]);
   int pin = int((*temp)[2]);

   // to be replaced with actual I/O:
   fprintf(EFTPOS,"<#%d:$%d:!%d>",
   card, bill, pin);
}
```

This user-coded function receives the response from the EFTPOS terminal. It takes the place of channel input, recvdebitresult?r, and passes the data received from the peripheral into the CSPm channel.

```
void recvdebitresult_q( ActionType t,
 ActionRef* a,
 Var* v, Lit* l )
{
   int r;

   // to be replaced with actual I/O:
   fscanf(EFTPOS,"<%d>",&r);

   // return result thru 'v'
   *v = Lit(r);
}
```

5. Future Work

The current CSP++ tool is based on the original definition of CSP, which does not include timing and interrupts. Until timing support is added, perhaps on the basis of “timed CSP” [9], specifications can be written in terms of event based “tock” [8] timing. The real-time nature of many embedded systems justifies the effort needed for its inclusion. Looking ahead, it is intended to extend CSP++’s synthesis capabilities to include hardware description language, with applications to hardware/software codesign [1]. Developers will be able to partition the CSPm specification between software and hardware, and inter-process synchronization and channel communication will be automatically synthesized by the tool.

6. Conclusion

Formal methods can be a key link in the design flow of embedded systems, but automatic translation of specifications, as provided by CSP++, is important for facilitating

rapid prototyping. By using the selective formalism approach, and by using SOPC as a vehicle, it will be possible to quickly create prototypes that can be turned into reliable embedded systems.

References

[1] G. D. Micheli, R. Ernst and Wayne Wolf, *Readings In Hardware / Software Co-design*, Morgan Kaufman Publishers, San Francisco, CA, 2002.

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

[3] W.B. Gardner, "CSP++: An Object-Oriented Application Framework for Software Synthesis from CSP Specifications," Ph. D. dissertation, Department of Computer Science, University of Victoria, Canada. 2000. `http://www.cis.uoguelph.ca/~wgardner/`, Research link.

[4] W.B. Gardner, "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.

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

[6] Michael G. Hinchey and Stephen A. Jarvis, *Concurrent Systems: Formal Development in CSP*, McGraw Hill, Berkshire, UK, 1995.

[7] C.A.R. Hoare, *Communicating Sequential Processes*, Prentice Hall, 1985

[8] A.W. Roscoe, *The Theory and Practice of Concurrency*, Prentice Hall Europe, Hertfordshire, UK, 1998.

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

[10] John Williams, uClinux on MicroBlaze Project Home Page, University of Queensland, Australia. 2005. `http://www.itee.uq.edu.au/~jwilliams/mblaze-uclinux/`, Research link.