Executable Specifications for Embedded Distributed Systems

Miroslav Sveda and Radimir Vrba, Brno University of Technology

Combining hardware components with an executable specification language facilitates the specification prototyping of embedded distributed systems. The specification language should cover process management, timing, and communication commands that real-time executive and communication task services of every node prototype can interpret. We use a technique that employs attribute grammars and either a macroprocessor or Prolog to execute the language.

The overall prototyping technique consists of four steps:

- defining a concrete specification language, including a description of its semantics through an attribute grammar;
- using text macros or Prolog definite clause grammar to implement a translator prototype that encodes this attribute grammar;
- designing a trial architecture and identifying its reusable components; and
- using the trial system architecture and the devised specification language to specify a target application system, followed by macroprocessor- or Prolog-driven expansion of that specification into executable code.

SPECIFICATION LANGUAGE

We use a time model that fits systems requirements to design embedded distributed applications. Currently, prevailing real-time specification and design methodologies are based on global clocks. The local-clocks alternative appeals to designers because it uses an approach that is similar to the techniques they use to implement applications.

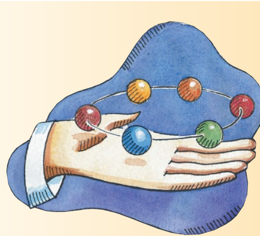
We derive local-time semantics from partial-order logical time and from a physical generator of periodic events. An event-count, E, counts the number of a specific type of events that have occurred during execution. Event occurrence invokes the implicit operation ADVANCE (E): E := E + 1. The explicitly callable operation AWAIT(E, s) suspends the calling process until the value of E is at least s. The call AWAIT(E, s) can reset the current value of E, enabling relative counting. An event-count monitors either a prescribed type of asynchronous external event or periodic internal events that an internal timer circuit implements as local-time clock ticks.

A system's logical structure employs distributed processes that use message passing to communicate asynchronously through an input buffer at the destination. The Asynchronous Specification Language (ASL), which describes these processes, respects true concurrency. The language's real-time operational semantics stem from the event-count model.

In the following example, the most important primitives relate to process specification, timing, communication, and control structure:

```
Process_name (
  is:list_of_s_inputs;
  os:list_of_s_outputs;
  ic:list_of_m_inputs;
  oc:list_of_m_outputs):
  ... endprocess;
wait(_, timeout);
wait(event, _);
wait(event, timeout, test);
send(message, destination);
loop ... [... when <cond>
  [action ...] exit]*
  ... endloop;
```

Each asynchronous process is equipped with an individually timed local clock. The process header contains lists labeled is, os, ic, and oc that act as the interface to the process's environment. The language distinguishes between signal inputs or outputs, denoting unbuffered events that either carry value or signal their occurrence. It identifies message inputs or outputs as typed asynchronous channels between process couples. These signals and messages declare the interprocess synchronization and communication, whose operations are driven by



Designers can use hardware components and an executable specification language to efficiently prototype embedded distributed systems.

the statements `wait(event, _)`, `wait (event, timeout, test)`, and `send (message, destination)`.

The value time-out defines the interval during which the primitive `wait(_, timeout)` suspends a process. In this case, the monitored event is every tick of the local clock, so the related event-count operation is `AWAIT(local_ticks, timeout_value)`. For the primitive `wait(event, _)`, which suspends a process until the specified event appears, the operation is `AWAIT(event_type, 1)`. The semantics of the combined statement `wait(event, timeout, test)` require two event-counts. The first event-count anticipates the specified event, and the second, with a lower priority, monitors the local clock. When the value of the Boolean parameter test is true, the event occurred within the interval time-out.

The primitive `send(message, destination)` uses message passing to implement asynchronous communication through an input buffer at the destination. To respect different local clocks, a clock common to the source and the destination controls the information transfer. Inputting a message induces the event for the related operation `AWAIT(message, 1)`.

The control structure primitive `loop... endloop` delimits an indefinite cycle, which is exited upon a true result of testing the condition following the primitive `when`. Consequently, the system executes statements occurring between the primitives `action` and `exit`, and, after that, those following the `endloop` primitive. This combined statement extends the language by providing additional control structures with simple macro-like text replacements such as

```
if <cond> then <s1> else
   <s2> fi;
   ~ loop when <cond>
     action <s1> exit <s2>
     when true exit endloop;
timeloop(timeinterval) <s>
   endloop;
   ~ loop wait(_,
     timeinterval) <s>
     endloop;
```

The control structure `timeloop (timeinterval)...endloop` specifies an isochronous loop, which the system periodically initiates whenever the time interval expires. The operation `AWAIT (local_ticks, timeinterval_value)` defines the operational semantics of timing these initiations.

The "Attribute Grammars" sidebar provides details about the L-attribute grammar. The example of a nested control structure explains the principle of control structure prototyping implemented over an unstructured assembler-type base language. A structured statement of the type `<LOOP>`, with primitives `loop – [...when [action...] – exit]* – endloop`, is accompanied by the attribute `BGN`, whose values distinguish individual instances of the `<LOOP>` structure during program compilation. The system stores the `BGN` attribute values in an expansion-time stack for nesting. The `CNT` counting type attribute is implemented as a global variable of compile time, and its value increases incrementally with each new `<LOOP>` instance. The relevant part of the attribute grammar is

```
<BLOCK>::=
    <LOOP><BLOCK>/<sb1>
  BGN   :=  CNT
<LOOP>::=loop <BLOCK>
    [<BLOCK>when <cond>
    [action <BLOCK>] exit]*
    <BLOCK> endloop;
  CNT   :=  CNT + 1
```

where `<sb1>` means a segment in the base language. In the syntactic rules, strings in angle brackets stand for nonterminals, and strings without angle brackets represent terminals; in the semantic rules, capital strings identify attributes. The timing and communication primitives map to real-time executive services and communication task services, and the control-structure statements and data structure specifications are encoded either in textual macros or in Prolog definite-clause grammar rules and normal Prolog goals, as the "Prototyping with Prolog" sidebar describes.

LOW-COST TRANSLATOR PROTOTYPING

The nested control structure `<LOOP>` helps to explain the principle of control structure translation prototyping. In addition to the L-attribute grammar, we use a single-pass textual macroprocessor, embedded as a preprocessor in an assembler. We use macros to implement the primitives `loop`, `when`, `action`, `exit`, and `endloop`; the macro `when` contains a proper decision condition—a more developed implementation uses parameter macros that evaluate complex conditions in the form of `and-or` decision trees. The synthesized attributes `BGN` and `CNT` appear as expansion-time variables. The macro `loop` embodies the semantic rules and pushes the actual value of `BGN` on the stack. If the `when` macro exists, it contains the stack-top inquiry. The `exit` and `endloop` macros remove the stack's old top. The decision condition translates into a conditional jump to the concatenation of a chosen symbol-string and a

Attribute Grammars

An attribute grammar is a context-free structure enriched by attributes and semantic rules. The context-free grammar G is an ordered quadruple $G = (N, T, P, S)$, where N is a set of nonterminals (textual variables), T is a set of terminals (textual constants), P is a set of syntactic rules that respect context-free constraints (the left side of the rule consists of only one nonterminal and the right side is a string of terminals, nonterminals, or both), and S is the starting symbol (initial nonterminal). The attribute is a data type with instances and values. A set of attributes ascribes every nonterminal; further, every syntactic rule has a set of semantic rules that enable calculation of left-side attribute values from right-side attribute values and constants. An L-attribute grammar allows calculation of all attribute values during only one pass through a derivation tree, created from the starting symbol by stepwise substitution of nonterminals with the help of syntactic rules.

Prototyping with Prolog

Many Prolog implementations provide a notational extension called definite-clause grammars, which also enable straightforward encoding of attribute grammars. A definite-clause grammar expresses context-free rules as logic statements. Certainly, notation is only a syntactic enhancement of Prolog: It brings no new expressive power because we can use standard Prolog to directly encode all rules. On the other hand, because we can use arguments with nonterminals, definite-clause grammars are more powerful than context-free grammars. This feature makes it possible to use attribute names to bind syntactic rules described by a definite-clause grammar with semantic rules encoded by normal Prolog goals. In this case, the translator prototype consists of a Prolog interpreter, the attribute grammar defined by definite-clause grammar rules for syntax, and normal Prolog goals for static semantics.

value of BGN to create the label. This algorithm binds adequate structure primitives, including forward referencing of yet-to-be-defined destination addresses for jumps.

The detailed implementation depends on the microprocessor's characteristics. When we embed the macroprocessor in a simple macro assembler, a less demanding option includes employing the auxiliary attribute NST as a global variable corresponding to the actual depth of nesting where NST is a pointer to BGN's stacked values. If the macroprocessor's frame has a string decomposition operation, a more advanced variant simulates the entire stack in the form of a symbol-string. Both techniques correspond to parsing with the help of a push-down automaton. We can also use the internal stack to perform a recursive-descent procedure when the macroprocessor allows a call according to a distributed pattern.

Members save 25%
on all conferences sponsored by the IEEE Computer Society.
Not a member? Join online today!
computer.org/join/

TRIAL ARCHITECTURE AND COMPONENTS

A node prototyping board supports experiments with embedded distributed systems using various topologies and communication protocols. The board's flexibility stems from its adaptable hardware architecture and the use of modifiable software components. The board contains two 8-bit microcontrollers interconnected through a programmable logical array and a shared-memory SRAM, configured as FIFO or two-port memory. Auxiliary circuitry supports both microcontrollers; the microcontrollers communicate externally through their parallel or serial ports.

The node prototyping board's software component consists of a simple real-time operating system kernel that maintains local physical clocks, monitors surrounding events, and synchronizes communication and application tasks. Real-time OS kernels for both microcontrollers create an environment that provides priority-based preemptive scheduling for event-driven communication and application tasks: external interrupts, timers, and messages. The communication task serves all messages routed to or from the microcontroller and offers remote communication services to application tasks.

To build a node prototype that fits our specification, generating the code for a process is the only application task for one of the two microcontrollers on the board; the other microcontroller is dedicated to extraboard communications only. These microcontrollers communicate with each other through onboard SRAM, an allocation that preserves maximal parallelism. The other tasks support monitoring and debugging throughout the rapid prototyping process. Moreover, they also support the next prototyping steps, reducing the number of nodes by scheduling some application processes in the same microcontroller.

We have successfully used this low-cost prototyping technique for real-world applications such as petrol dispenser control, multiple lift control, and interconnecting different types of low-level field buses. To date, we have developed two semantically close procedural ASL versions along with their related translator prototypes for use in microcontroller application domains. While we can classify the syntax for these ASLs as Pascal-like and C-like with distributed process extensions, both have formal operational semantics for use in verifying specifications with model checking. Current research areas include model-checking support tools and application domains.

Of course, this low-cost technique cannot compete against large prototyping environments in providing user assistance. On the other hand, this technique is immediately available and easily adaptable to special requirements as the local-time concept demonstrates.

__Miroslav Sveda__ is an associate professor in the Department of Computer Science and Engineering at the Brno University of Technology, Czech Republic. Contact him at sveda@dcse.fee.vutbr.cz.

__Radimir Vrba__ is a professor in the Department of Microelectronics at the Brno University of Technology, Czech Republic. Contact him at vrbar@umel.fee.vutbr.cz.

Editors: Jerzy W. Rozenblit, University of Arizona, ECE 320E, Tucson, AZ 85721, jr@ece.arizona.edu; and Sanjaya Kumar, Honeywell Technology Center, MS MN65-2200, 3660 Technology Dr., Minneapolis, MN 55418, skumar@htc.honeywell.com