Abstract. This paper describes CPA, a framework for specifying the behavior of control-oriented digital systems. CPA allows the expression of sequential behavior, data transformations and timing constraints in an integrated representation. It supports the abstract, declarative specification of behavior independent of any particular implementation. Thus it is especially useful for verifying a design against a set of externally-defined requirements.

INTRODUCTION

The goal of this work is to discover what properties are required for the correct operation of a digital system and how best to express them in a conceptually sound and coherent manner. We concentrate here on control-oriented systems, such as protocol handlers and interfaces.

Section 1 describes the results of a study we undertook to learn what properties must be specified for control-oriented systems and how those properties are best expressed. In Section 2 we propose a new approach to specifying system correctness. Finally, Section 3 contains a discussion and evaluation of our method.

1. REQUIREMENTS FOR A BEHAVIORAL SPECIFICATION SYSTEM

1.1 Method

Instead of choosing a formalism a priori, we decided to look at how designers and system architects write specifications. We studied several specifications for communication chips, including one published as a formal standard [1] and some others contained in internal working papers. We also looked at some specifications of bus protocols and of software modules for communications systems.

1.2 Observations

The specifications focused on the input/output behavior of the systems they described. Each system or subsystem was described by its input/output connections, ports or variables, and possible events at those inputs and outputs.

It is true that some of the specifications included a partial description of the internal structure of the system, either by providing a state machine representation or by describing some of the internal memory elements and how data was processed in them. The designers told us, however, that they did not feel bound by these internal descriptions, and the designs they produced were different in internal structure from what was suggested in the specifications. Only the requirements for the behavior at the interface had to be satisfied.

The properties specified in the descriptions we studied fell into four basic categories:

1. Format Information. The information flowing through a port usually has a defined structure, with different subfields having different meanings. In a communications protocol, for example, data is sent and received in frames, where the first few bytes serve as frame control and format characters. They are followed by a variable number of information bytes, then error checking codes and more format information.

2. Precedence Relations. These relations specify the order in which inputs and outputs must occur, for example, When a Data Request is received, the interface places the next byte to be transferred on the Data lines.

3. Functional Relations. These relations show how the values input and output on the various ports of the system are related. Sometimes an output depends on a whole sequence of inputs or outputs, as in the calculation of a check character: The CRC appended to the frame is the sum of all the data bytes in the frame mod 2^n.

4. Execution Constraints. Execution constraints are bounds (both upper and lower) on the number of times certain events can occur and on the delay between events. One example of a timing constraint is: The interface must be able to process one byte of data every clock period.

Based on the previous observations, as well as our discussions with designers and our own intended uses, we have compiled the following list of desirable traits for a specification system:

Abstract view of behavior. It should be possible to describe just the input/output behavior of a system
without constraining its internal structure. It should also be possible to specify only those properties that are required for the correct operation of the system.

**Behavioral Properties.** It must be possible to express both precedence and functional relations between sequences of inputs and outputs. There should also be a mechanism, integrated into the representation, for showing bounds on how many times certain events can occur and on the relative timing between events.

**Usability.** The system should support the way designers work. It should allow incremental development and hierarchical description, and it should have facilities for analysis of the specification under development.

**Relation to Implementation.** It should be straightforward to derive properties from the specification against which the implementation can be verified.

### 2. CPA: A FRAMEWORK FOR SPECIFICATION

Here we present a framework for specification that is constructed to satisfy the criteria listed in Section 1.2 as completely as possible. It is not intended to be just a new specification language or hardware description language, although we do develop an example language here.

We call our representation CPA, for Conditions, Precedence relations and Assertions, the three elements from which our specifications are constructed. CPA models a system as one or more processes, where each process is described as a sequential machine. This means that each process interacts with its environment, including other processes, through a series of discrete events, each representing the transmission or reception of some digitally coded information. Thus, the model does not describe the electrical properties of systems, but it does include real-time behavior.

Each process in a CPA description has a set of ports defined for it, where each port is either an input or an output and has a specific type. The type gives the bit width of the port, or equivalently the range of values it can take.

A process interacts with the external environment and other processes by reading and writing values through its ports. All behavior is described through these reads and writes. A process has control of its reads as well as its writes. In other words, a process reads an input when it decides to, not necessarily when the input appears. A system specification, however, will usually contain constraints that determine how quickly a process must respond to externally supplied inputs.

The act of reading or writing a value on a port is called an event. A fully determined event has three components: The port \( N \) where the event occurs (including the process to which the port belongs); The value \( V \) read or written; The time \( T \) when the event occurs. An event \( E \) can therefore be written as a triple \( < N, V, T > \). For any such event \( E \), we use selectors \( .n, .v, .t \) to identify the individual components. In other words, if \( E = < N_0, V_0, T_0 > \), \( E.n = N_0 \), the port name for \( E \), \( E.v = V_0 \), the value for \( E \) and \( E.t = T_0 \), the time of \( E \).

#### 2.1 Elementary Event Expressions

Within this model, specifying the behavior of a system means determining the acceptable sequences of input and output events. We do this in CPA by writing a series of expressions called event expressions. Each event expression has the form:

\[
\text{when } C \{ \theta_1 -> \theta_2 \} \text{ where } A:\]

\( C \) is the Condition, a logical formula giving the conditions under which this expression applies. \( \theta_1 -> \theta_2 \), the precedence relation, gives a required ordering between two or more events. Finally, \( A \), the Assertion, is a logical formula specifying constraints on the sequence numbers, values, and times of the events named in the precedence relation.

An event expression "when \( C \{ \theta_1 -> \theta_2 \} \) where \( A;" \) expresses a relationship between events. It can be read as follows: Whenever an event matching \( \theta_1 \) occurs and \( C \) is true, it must be followed by an event matching \( \theta_2 \) and \( A \) must be true of these events.

\( \theta_1 \) and \( \theta_2 \) are called event schemas. In event schemas, events are identified primarily by the ports on which they occur. Thus, for example "sender::data" stands for an event that outputs a value on the \( data \) port of process \( sender \). An event name can be qualified by adding a constraint on the value read or written by that event. "Sender::data(v := 0);" for example, stands for an event that writes a nonzero value on port \( data \).

We further distinguish among the events that match a given event schema by using indexing. In our notation we put the index in square brackets. Sender::data[5], for example, is the fifth output from port \( data \) of process \( sender \); and \( sender::ack[j](v == 1) \) represents the \( j \)th time a 1 is read on port \( ack \). Because we are interested in infinite, repetitive sequences of events, where absolute indices are not usually significant, we will often use relative indexing of events. Putting \( a + \) or \( - \) before an index expression means that index is relative to the reference event, which is the event matching \( \theta_1 \) in the event expression "when \( C \{ \theta_1 -> \theta_2 \} \) where \( A;". For example, in the event expression
when (sender::ack[-1].v == 0) 
  [sender::ack(.v == 1) \rightarrow sender::data[+1]]
where (sender::data[+1].v == x);

sender::data[+1] is the first output at the data port following
the input of 1 at the ack port. Sender::ack[-1] refers
to the last time ack was read before reading a 1 on it.
Therefore, the previous event expression can be translated
as follows: If the last time sender checked ack, it saw a 0
and now it sees a 1, then its next output on data must be x.

If the reference event \( \Theta_i \) is referred to in C or A and
it does not have an absolute index, it is referenced as \( \Theta_i[0] \).

For brevity we will sometimes omit the process part of a
port name when it is clearly understood.

Timing constraints on events are expressed in terms of the
time components of those events. For example, if \( E_2 \)
must occur within 50 time units of \( E_1 \), we write
\[ E_2.t - E_1.t < 50 \]

For synchronous processes, we also allow time to be
expressed in clock cycles. \( E.c \) stands for the cycle in
which the event \( E \) occurs, and \( E_2.c - E_1.c \) is the
number of cycles from the occurrence of \( E_1 \) to the
occurrence of \( E_2 \).

The condition \( C \) in an event expressions is useful for
specifying conditional actions. Consider, for example, an
interface circuit B, which transfers data from its input Din
to its output Dout. After each output, it waits for an acknowl-
agement on line Ack from the receiver before it
sends the next output. If the acknowledgement is not
received within a certain time, say 200 clock cycles, B
raises a Timeout flag. The specification for this behavior is:

\[
\text{process B}(\text{in Din, out Dout, in Ack, out Timeout})
\]
\[
\begin{array}{l}
\text{out_eq_in: when TRUE [Din \rightarrow Dout[+1]]}
\quad \text{where Dout[+1].v == Din[0].v};
\text{check_ack: when TRUE [Dout \rightarrow Ack[+1]]}
\quad \text{where Ack[+1].c == Dout[0].c <= 10};
\text{next_output: when (Ack[0].c <= Dout[-1].c <= 200)
\quad =\rightarrow Dout[+1]};
\text{check_again: when TRUE [Ack(.v == 0) \rightarrow Ack[+1]]}
\quad \text{where Ack[+1].c == Ack[0].c <= 10};
\text{not_next_out: when Dout[0].c == Ack[+1].(v == 1).c >= 200)
\quad \rightarrow Timeout[+1](v == 1)}
\quad \text{where Timeout[+1].c == Dout[0].c == 200;}
\end{array}
\]

The first three lines describe a normal transfer. Line
\text{out\_eq\_in} states that the value of the output is always the
last value read at the input. Line \text{check\_ack} requires that
after every output, the system must check the Ack line
within 10 clock cycles. If a 1 is seen on Ack within 200
cycles of the last output, according to line \text{next\_output},
the system proceeds with the next output. Line
\text{check\_again} states that whenever a 0 is seen on Ack, the
system should check Ack again within 10 cycles. Finally,
line \text{not\_next\_out} says that if an acknowledge is not read
within 200 clock cycles of the last output, the \text{Timeout}
flag is raised.

Each event expression can be visualized as an arc in an
event graph [3], where each node in the graph is an
input/output event, and each arc is a path the system can
take from one event to the next. Arcs can be labeled by
the conditions under which they can be traversed (C) and
constraints on how they are traversed and what their
effects are (A). Figure 1 shows how the specification
fragment for the timeout example can be interpreted as an
event graph. The paths in the graph are annotated with
the conditions under which they are taken, but the assertions
are not shown.

CPA has been extended to include chains of events, parallel
event schemas, and hierarchical events. Details are
given in an earlier report [9], along with some more com-
plex examples, including a serial-to-parallel converter
with parity check and a VME Bus interface. The report
also has a formal semantics of CPA.

3. DISCUSSION

3.1 Relation to Other Work

We took the idea of describing behavior with predicates
over sequences of inputs and outputs combined with
sequencing information and indexing from the Behavior
Expressions of McFarland and Parker [10]. The idea of
attaching timing constraints to a behavior description has
been used in hardware description languages [5], [6] and
in graphical specification systems [2].

The style of description used in CPA, where behavior is
described with individual relations and properties, is similar
to that used in temporal logic [11]. CPA, however,
expresses a wider range of properties. CPA is similar to
the Real-Time Logic of Jahanian and Mok [8], especially
in the way it handles timing constraints. The underlying
model and the semantics are different, however. Moreover,
while Real-Time Logic deals only with individual
signals, CPA can express properties of complex data
transformations and transfers.

CPA is a good complement to formalisms that use an
operational style of description, such as state machines
[4], [7]. The operational style of description allows the
designer to think in familiar terms about behavior and to
Figure 1. EVENT GRAPH FOR A CPA SPECIFICATION

give a precise and detailed description of it. CPA, on the other hand, is abstract and declarative. It provides another way of thinking about the behavior of a system that can be checked against an operational description, and it does not overspecify the system.

3.2 Use

Our most immediate interest is the verification of state machine representations against CPA descriptions. We also need to develop a usable front end for CPA and provide tools for analyzing the specifications.

ACKNOWLEDGEMENTS

Many colleagues at Bell Laboratories have made valuable contributions to this work, including Kurt Keutzer, A. S. Krishnakumar, Carl Seaquist, Chuck Stroud, Dave Pierce, Leon Mekly, Tim Lippold, Lisa Kowalski, James Piccarello, and James Rowland. We are most grateful to them.

REFERENCES


