A common compiler for LOTOS and SDL specifications

by C. Binding
W. Bouma
M. Dauphin
G. Karjoth
Y. Yang

This paper describes a translation of LOTOS and SDL specification languages into executable code, as it was prototyped in the Specification and Programming Environment for Communication Software (SPECS) project under the Research and Development in Advanced Communications in Europe (RACE) program. Both languages are translated into a common intermediate representation in the form of a network of state machines with both synchronous and asynchronous communications. By a series of transformations that make full use of the equivalence relations defined on LOTOS processes, this translation solves unique problems stemming from the highly abstract nature of LOTOS. The common intermediate representation is mapped into C code that can be executed in a specific run-time environment, implemented on a UNIX®-like operating system. SPECS has also developed a pragmatic approach to represent implementable data types in the algebraic framework of LOTOS and SDL, based on a set of predefined type constructors.

The introduction of a European integrated broadband communications network (IBCN) and its associated services requires that a huge amount of software be developed. This software will have a high degree of complexity, due to factors such as real-time constraints and the distribution over many processors in a heterogeneous environment (multivendor, multicountry, and multilanguage). The reliability requirements on this software will be as high as on current telephone networks. The combination of the characteristics of this software is such that new software development methods and tools are needed to achieve the required level of quality at an economically justifiable price. Therefore, a set of projects has been set up in the European Research and Development in Advanced Communications in Europe (RACE) program, whose objective is to define a suitable programming infrastructure for the IBCN software. One of these projects, Specification and Programming Environment for Communication Software (SPECS), has as its primary objective the definition of methods and tools for the specification, design, implementation, and testing of telecommunications software. A general presentation of the RACE program and the SPECS project can be found in Reference 1.

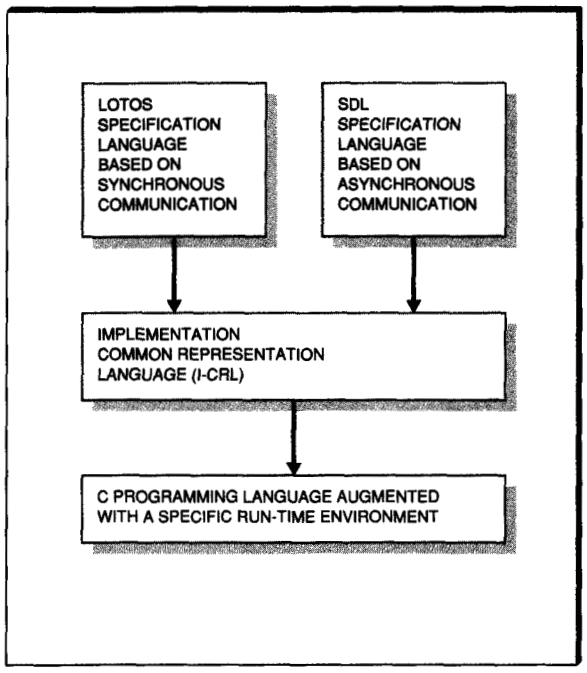
The approach of SPECS is based on the use of formal languages very early in the development process to overcome the problems stemming from the inherent ambiguity of natural language and to allow the early application of semantic tools on formal specifications for their verification and validation.[2,3] After an evaluation of currently

available formal languages for the specification of telecommunication systems and software, SPECS has focused on the internationally standardized languages, Language of Temporal Ordering Specification (LOTOS) and Specification and Description Language (SDL). In the SPECS approach, these languages are used as broad-spectrum languages. They support the design and implementation activities, in which abstract functional specifications, expressed in one of these languages, are transformed into implementation-oriented descriptions of the system. (SPECS also supports the mixing of these languages in the description of a system. This aspect is addressed in more detail in Reference 4.) These transformations address the system structure as well as the algorithmic parts of the specification. From the implementation-oriented description, implementations are then generated automatically. To achieve this goal, SPECS provides translation algorithms and a prototype of a compiler that translates large subsets of SDL and LOTOS into the programming language C. The design of this compiler and the experience gained in its development are the topics of this paper.

The architecture of this compiler is outlined in Figure 1. It is centered around the Implementation Common Representation Language (I-CRL), based on a formal model of dynamically configurable networks of communicating state machines, with possibilities for synchronous (LOTOS-like) as well as asynchronous (SDL-like) communication. The generated code relies on a specific, but portable, run-time environment that provides facilities for both synchronous and asynchronous interprocess communication.

The major problems encountered in the definition and the development of the prototype were due to the abstract nature of the source languages, especially LOTOS, which have been designed as specification languages. LOTOS provides mechanisms to express nondeterministic choice—an important feature to achieve abstractness and implementation independence. To overcome the inherent inefficiency of a straightforward implementation of the powerful communication mechanism of LOTOS (multiway synchronization with value negotiation), specific optimization techniques had to be developed, e.g., the merging of closely coupled processes into a single state machine, and the transformation of tail recursion into loops.

Figure 1 Compiler architecture defined by SPECS



The remainder of this paper contains an overview of LOTOS and SDL, a description of the translators from LOTOS and SDL to I-CRL and the ensuing phase of code generation from I-CRL to C, a discussion of the run-time environment to model the concurrency in I-CRL on the C+ UNIX** level, and finally, a discussion of implementation issues.

LOTOS

LOTOS has been developed by experts in formal description techniques within the International Organization for Standardization (ISO) as a standard language for formal specification of communication protocols and, in particular, for Open Systems Interconnection (OSI) protocols. It is based on the idea that systems can be described by defining the temporal ordering of events that are externally observable. The language has two components, one dealing with the algebraic description of data based on the algebraic specification language ACT ONE[5] and one for the description of process behaviors and interactions based on a modification of the Calculus of Communicating Systems (CCS)[6] with elements from Communicating Sequential Processes (CSP).[7] The language is defined in Reference 8.

Data are defined by specifying *sorts* (sets of values), typed operator signatures, and equations, which can be seen as recursive function definitions. Semantics for data descriptions are obtained by combining them into one big data description and constructing the corresponding data algebra.

Behavior is described by so-called *behavior expressions*. The simplest behavior expressions are *stop*, which defines a process not able to perform any action, and *exit*, which communicates successful termination possibly with value passing. The *action-prefix* operator composes behavior with an observable communication or internal (silent) action. Such communication takes place across interaction points called *gates* with the possible exchange of data. This communication is *synchronous*: an executing LOTOS behavior expression can only communicate across a gate if there exists a partner to do so. Furthermore, during synchronization, data values may be negotiated. Note that in LOTOS there is no direction of exchange of data. Other ways of composing behavior are illustrated in the example below.

After obtaining a description of behavior with the above constructs, *process definitions* allow a name to be given to behavior expressions and to make them dependent on data variables and gate names. Such a name constitutes a simple behavior expression and is called *process instantiation*. Process instantiation then makes it possible to have recursive definitions of behavior. A complete specification is obtained by a set of process and data-type definitions and the description of the system behavior in terms of these definitions.

Semantics is given to a LOTOS specification by first constructing the data algebra, and then, based upon this data model, a process graph or *labeled transition system* with nodes consisting of behavior expressions and edges labeled with atomic actions. Only those nodes that are derivable from the root node (the whole specification) by means of derivation rules are taken into consideration.

In the following example, we describe a timer facility as it may be used in the implementation of communication protocols to recover from the loss of messages over an unreliable channel. Assume each transmitted message carries a sequence number of identification. If there are NumberOfTimers timers, each timer can control the acknowledgment for message SeqNo modulo NumberOfTimers if there are not more than NumberOfTimers messages outstanding at any moment. For a detailed explanation, see e.g., Reference 9.

The process TimerBank forks (creates) a number of timer processes. The data used here are integers as defined in a standard library and timer signals defined by the data type TimerSignal.

```
library NaturalNumber endlib

type TimerSignal is
sorts TimerSignal
opns set, cancel, expired : -> TimerSignal
endtype

type ExtendedNat is NaturalNumber
opns _Mod_ : Nat, Nat -> Nat;
     NumberOfTimers -> Nat
eqns forall x, y : Nat
     ofsort Nat
     x lt y => x Mod y = x;
     x Mod y = (x + y) Mod y
endtype
```

The sort TimerSignal has three elements, namely set, cancel, and expired, represented as constants (nullary operators in LOTOS jargon). The type ExtendedNat extends Integer with the operation Mod, which is not presented in the SPECS library. The two "_"s indicate Mod is an infix operator. The equations given under eqns define the semantics of Mod, the remainder of integer division. NumberOfTimers is a constant whose value is not defined since there are no equations for this operator.

The behavior is specified by processes. The process TimerBank [t] (0) forks NumberOfTimers number of Timer processes that are in parallel but do not communicate with each other. This is expressed by

```
process TimerBank [t] (TimerId : Nat) : noexit :=
  [TimerId lt NumberOfTimers] ->
    ( i; TimerBank [t] ( Succ (TimerId))
     |||
      Timer [t] ( TimerId) )
endproc
```

"i" is an internal event that cannot be observed. The *action prefix* operator ";" specifies that only after the event has been executed the process may behave as TimerBank.

New instances of process Timer are forked as long as the condition TimerId lt NumberOfTimers holds. Thus the process TimerBank [t] (0) will behave as the following process:

```
Timer[t](0) |||
 (i; Timer[t](1) |||
          ...                |||
   (i; Timer[t](NumberOfTimers-1) ) . . .)
```

Process Timer is composed of process IdleTimer in parallel with process Identification, both synchronizing at gate t. Process Identification uniquely identifies each timer, and process IdleTimer describes that a timer can be canceled or may expire after it has been set.

```
process Timer [t] (TimerId : Nat)
             : noexit :=
   IdleTimer [t]
 |[t]|
   Identification [t] (TimerId)
endproc
```

A process may offer events at a gate possibly associated with data values, the notation "?" meaning any value of the given sort, and "!" only the given value. The "?" values may be constrained by a *selection predicate*. The *enabling* operator "> >" composes two behavior expressions in sequence; the second behavior is entered only after the first behavior has successfully terminated. Successful termination is denoted by exit. "[]" is a choice between two behaviors.

```
process Identification [t] (MyId : Nat)
                        : noexit :=
  t ? Id : Nat ? AnySignal : TimerSignal
       [MyId = (Id Mod NumberOfTimers)];
  Identification [t] (MyId)
endproc (* Identification *)

process IdleTimer [t] : noexit :=
  t ? AnyId : Nat ! set;
  RunningTimer [t] (AnyId)
 > >
  IdleTimer [t]
where

  process RunningTimer [t] (AnyId : Nat) : exit :=
    t ! AnyId ! cancel; exit
   []
    i; t ! AnyId ! expired; exit
  endproc (* RunningTimer *)
endproc (* IdleTimer *)
```

In the parallel composition IdleTimer [t] |[t]| Identification [t] (TimerId), both processes must synchronize at gate t. A synchronization takes place if both processes offer the same event for that gate, satisfying the associated selection predicate [TimerId = (Identifier Mod NumberOfTimers)].

Process RunningTimer is the capability to execute action t ! AnyId ! cancel that cancels the timer as well as to execute an internal action. However, once an action is chosen, either by the environment that is not specified here or by the process itself, in case of the internal action, the successor behavior is determined. The internal action here can be interpreted as "after some delay." LOTOS has no notion of real time; it is therefore not possible to specify a delay in seconds. Therefore, SPECS has introduced a notation to associate a duration with an internal action.

SDL

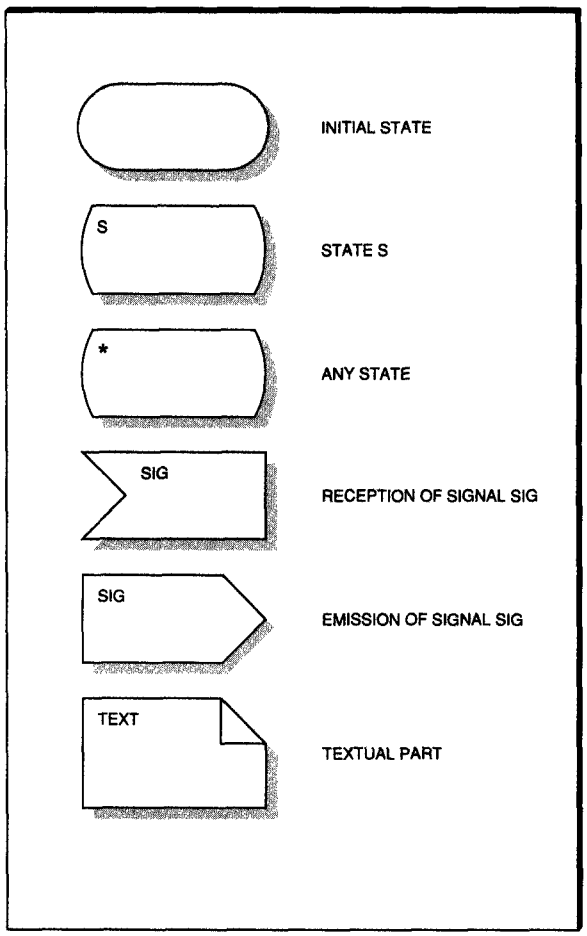
SDL is the specification language standardized by the International Telegraph and Telephone Consultative Committee (CCITT).[10] It has both a textual and a graphical syntax. The language is subject to a four-year revision cycle. The update in 1992 will, among others, contain facilities for object-oriented specifications that have been partially developed by the SPECS project.

Data descriptions are, as in the case of LOTOS, algebraic specifications based on ACT ONE. Several language constructs have been introduced to simplify the writing of data descriptions of which the most notable ones are the so-called *literal classes*, allowing a user to define (infinite) classes of constants like the usual notation for integers, strings, and *error* values.

Processes are the basic building blocks of SDL. They are made up of input and output statements and state elements. Process descriptions are extended finite state machines, extended while the transition of a state to a next state is not only determined by input signals. There is also the notion of a set of (program-like) variables internal to a process, and decisions that distinguish between next states can be taken based upon expressions made up out of these internal values.

With each process is associated an unbounded queue that receives incoming *signals*. Signals are messages that are sent between processes along

Figure 2 Some SDL symbols



signal routes or *channels* in the case of processes living in different blocks. They carry a source and destination address and possibly data values. Signals are consumed by input statements in the order in which they arrive. This implies an asynchronous communication model between SDL processes: a process deals with a signal as soon as it can find the time to do so, without setting up synchronization with other processes.

Using the graphical syntax explained in Figure 2, Figure 3 shows an example of a simple SDL process. When this process ACCUMULATOR comes into existence, the variable total is initialized and the process enters the state IDLE. An add signal instructs it to add any further values sent by the signal value to its total, while the subtract signal makes it subtract such values. Upon reception of a query_reset signal, the process sends the accumulated value by a sum signal and enters again the IDLE state.

The static structuring mechanism in SDL is the *block* concept. A division in blocks represents the static structure of an SDL system. Blocks, processes, and channels can be subdivided into subcomponents. These structuring concepts support the hierarchical decomposition of an SDL system.

An example of a block diagram is shown in Figure 4. The block a is composed of two processes, ACCUMULATOR and CLIENT, connected by the signal routes R1 and R2. The process CLIENT is connected via the signal routes R3 and R4 to the channels C1 and C2.

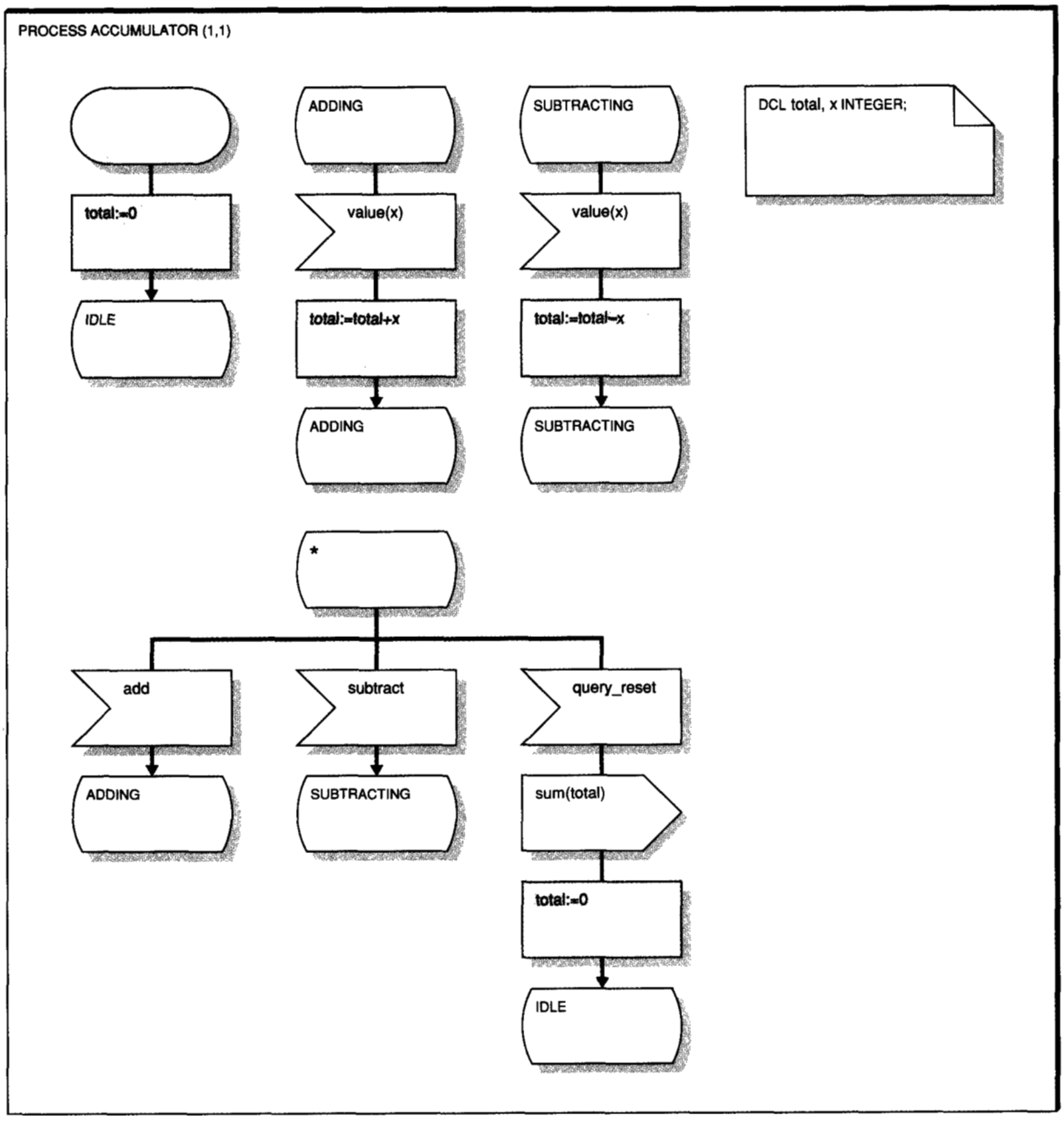
SDL also provides a procedure concept, comparable to the classical procedure concept: when a process invokes a procedure, it is itself suspended, and control returns to the process only when there is a return from the procedure call. Furthermore, there is some concept of time in SDL. A special variable provides access to system time; it is possible to set timers and thus model a certain duration of time.

A formal semantics for SDL has been given in Annex F of Recommendation Z.100[10] in the denotational style, i.e., by interpreting SDL constructs in an abstract SDL machine. SPECS has given an operational semantics to part of SDL in the same style as the official LOTOS semantics: for an SDL specification a labeled transition system is constructed. Execution of the specification is then performed by stepping through this transition system, as for LOTOS.

The SPECS support for data in LOTOS and SDL

Although it is easy to specify simple data types in the algebraic abstract-data-type style, they remain tedious to write and require great care to provide a complete set of equations. Therefore, the task of giving algebraic specifications of data is considered to be a serious problem in the use of LOTOS and SDL and even tends to be avoided. For this reason and to have a uniform set of data types, it was decided that all the languages occurring in the SPECS architecture should implement the same fixed set of data types and type generators, each with a fixed set of predefined operators. This set is directly modeled on the set

Figure 3 A simple SDL process

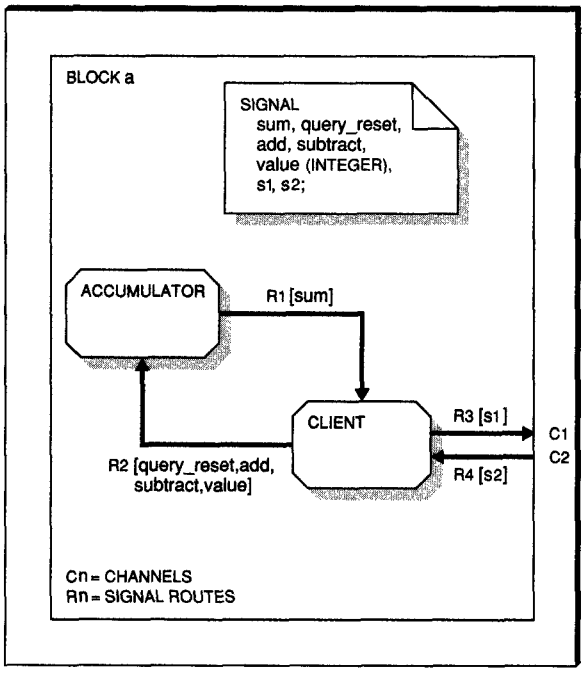


of data types available in imperative languages. It is comprised of:

- A set of *basic* types, e.g., Boolean, integer, rational, and character
- The parameterized types list, set, and map
- The type schemes enum, array, record, and variant record
- Facilities to allow user-defined functions

Details on the implementation of these SPECS data types are given in later sections of this paper on

Figure 4 An SDL block diagram



I-CRL data part, translation of data, and translation of the data part.

Translation to the common intermediate representation

As stated in the introduction, the Implementation Common Representation Language, or I-CRL, serves as an intermediate layer in the compiler architecture. It serves two goals.

1. It supports the decomposition of the two translations, LOTOS to C with the run-time environment (RTE) and SDL to C with the RTE, into two complementary phases.
   - A first phase transforms the communication and concurrency semantics of the source language into an execution-oriented model, but stays within the semantic world of labeled transition systems. It is thus possible to define and check the formal correctness of this translation phase.
   - A second phase maps the execution-oriented model onto a specific programming language (C) and the supporting run-time functions. It handles the problems introduced by the limitations of the programming language and the finiteness of the machines on which the generated code will be eventually executed.
2. It factors out the part that is common to these two translations, so that it needs to be defined and implemented only once. Specifically, the second translation phase described above is common to the LOTOS and SDL translators.

The design of the I-CRL has been based on previous work of one of the authors[11] on the compilation of LOTOS specifications. The model has been extended and adapted to also cover semantic aspects of SDL and the data part of both languages.

The I-CRL is composed of two independent, but interrelated parts, the data part and the control part. These are presented in the next two sections.

The I-CRL data part. The I-CRL data part is a simple version of a functional language, conceptually located between the algebraic abstract data types of LOTOS and SDL, and imperative programming languages. The main features are:

- Referential transparency—a function that has no side-effects
- Support for basic data types and static structures—Boolean, integer, rational, character, array, record, and variant record
- Support for dynamic data structures—list, set, map—that come with a rich set of predefined operators

Unlike most functional programming languages, I-CRL does not provide higher-order functions nor pattern matching facilities, as these were not necessary for the building of the compiler.

The two main concepts in the data part are *sorts* and *operators*. Sorts denote sets of values. The sorts BOOL, INTEGER, RATIONAL, and CHAR (the ASCII character set) are predefined, together with a set of operators on them denoting the usual functions, such as addition and subtraction. Sort constructors provide the means to define new sorts. The enumeration sort constructor ENUM provides the means to define a sort with a finite set of values. The sort constructors ARRAY, RECORD, and VARRECORD (variant-record) have the same meaning as in conventional programming lan-

guages such as PASCAL. The sort constructors LIST, SET, and MAP provide support for frequently used dynamic data structures. Even though I-CRL does not contain the notion of pointer, any kind of dynamic data structure, e.g., binary trees or cyclic data structures, can be represented using variant records together with the possibility of recursive sort definitions. Sorts defined with these sort constructors are equipped with a set of predefined operators on them. The sorts of all the expressions occurring in an I-CRL specification must be declared, e.g., the following declares `intlist` as a list of integers:

```
(SORT-DCL intlist (LIST INTEGER))
```

Operators denote a mapping from tuples of input values to an output value. Operators take a fixed number of arguments and are strongly typed; each operator has a list of input sorts and one output sort. Semantically, operators denote either a constant value (in case the input sort list is empty), or a mapping from one sort (in case the input sort list has one element) or the Cartesian product of two or more sorts (in case the input sort list has two or more elements) to the output sort.

In the following simple example, the operator `square` is defined on the rationals by `square` (x) = x · x. The `OP-DCL` clause defines the input sort list and the output sort. Next, `x` is defined as the formal parameter, and then the result of the operator is given as the application of the predefined function `multiply`.

```
( OP-DEF
  ( OP-DCL square ( RATIONAL ) RATIONAL )
  ( x )
  ( OP-APP multiply ( x x ) ) )
```

An operator can be defined recursively, i.e., the expression defining an operator may contain an application of the operator itself. Recursion is frequently used, especially since the I-CRL data part contains no loop construct that is essentially nonfunctional, because it relies on destructive assignment.

Expressions can be constructed in the usual way, using literals, operator applications, and simple control structures, such as *if ... then ... else*. Expressions occur not only in operator definitions, but also in the control part of I-CRL, described in the next section.

The I-CRL control part. The I-CRL control part defines a dynamic network of communicating state machines, where state machines serve as processes as well as procedures. In the way processes are composed and communicate, I-CRL follows an algebraic approach, as first proposed by Milner for CCS,[6] and more specifically the LOTOS model. The LOTOS concept of gates has been generalized, however, to include queued asynchronous communication and shared variables. Also, a supplementary process creation mechanism has been added that reflects the way processes are created in SDL.

State machines. State machines, more precisely extended finite state machines, are the basic components of I-CRL. An automaton that embodies the imperative programming concept of assignment statements gives them an operational structure.[12] State machines have the combined functionality of *regular* LOTOS processes, i.e., those that consist of a finite number of control states, and of SDL processes.

A state machine can be viewed as a collection of *rules* $R = \{r_1, \ldots, r_n\}$ on some set of *state variables* $V = \{v_1, \ldots, v_m\}$. Each variable is typed, i.e., it has a specific sort. The set of possible assignments of values to the state variables constitutes the set of *data states*. Each rule r_i is of the form $\langle j,p \rangle \xrightarrow{a} \langle j',c \rangle$ and defines that from control state j the state machine can perform an action a and thereby enter control state j' provided the enabling predicate p is true under the current assignment of the state variables. Furthermore, c, a statement on the state variables, is executed and may change the values of some state variables. Thereby, a rule defines a class of transitions. In addition, a state machine has an *initialization statement* c_0 that assigns initial values to the local variables in V, and an *initial control state* j_0.

Associated with a stack, state machines may be invoked as procedures and return a vector of values on successful termination. A procedure is a state machine that has variables as parameters. The variables may be passed either by value or by reference. When a state machine performs a *call* command, its behavior becomes the behavior of the state machine of the procedure, where the formal parameters have been replaced by the actual parameters passed with the call command, until the state machine procedure terminates.

Processes. A *sequential process* is a state machine that is augmented with a (possible empty) set of input signals and a (possible empty) set of timers. Its behavior is determined by the associated state machine. When created, a process has an implicit queue attached to it that accepts signals of the types listed in the *input signals* list. The timers behave as in SDL.

There are three constructors to build complex systems from sequential processes. State machines can be composed in parallel to form a network of processes. Such networks are dynamic, as state machines may create new processes. A special form of parallel composition is supported by the disabling operator. Further, gates can be introduced opening a new scope. Communication over these gates is only possible within this scope.

Translation to I-CRL is different for data types and control parts.

Processes can communicate synchronously, asynchronously, or through shared variables. Synchronous communication actions consist of multiple and "bidirectional" experiment offers at interaction points called *gates*. An associated selection predicate may impose constraints on the values to be received in the input offers. More than two parties may engage in this communication scheme. Asynchronous communication, designated by *send* and *receive*, does not cause the sending process to wait. The *receive* operation dequeues a message (called signal) from the input queue of the process, if one is present; otherwise it causes the receiving process to wait until a message arrives or until some Boolean expression, possibly involving global variables, becomes true.

There is also an *internal action* that allows a process to proceed on its own without synchronization with other processes. As a parameter it takes a time value that specifies a delay and thus provides a time-out facility. The process interacts with a timer by executing *set-timer* and *reset-timer* commands, as in SDL.

Commands. In addition, a number of special actions allow a state machine to modify its environment. The *call* command pushes the state machine's current state on the stack and executes a procedure as a subroutine. The *fork* and *create* commands allow a state machine to create new processes, with and without an input queue.

From LOTOS to I-CRL

The translation from LOTOS to I-CRL is composed of two parts: the first maps LOTOS data specifications onto the I-CRL data part, exploiting specific compiler directives; the second maps LOTOS behavior expressions onto extended finite state machines, the basic units of I-CRL.

Translation of data. In LOTOS, it is not possible to support some of the SPECS data types directly, specifically type schemes like records and arrays. Therefore, it was decided to implement the SPECS data types by means of special compiler directives, called *pragmas*. We will introduce these pragmas through examples.

LOTOS pragmas. LOTOS pragmas have been implemented as special comments denoted by "(*$" and "$*)" bracket pairs. They are divided into *sort* pragmas and *operation* pragmas. For basic data types, no pragma is needed; they are implemented as a special library of basic types. We describe the data types introduced in the LOTOS example in the earlier LOTOS section again, this time using pragmas.

First, we recognize that the type `TimerSignal` can be implemented as a simple enumerated type, and use the *enum* pragma, expressing this.

```
type TimerSignal is
  sorts TimerSignal (*$ enum 3 $*)
        (* TimerSignal is an enumerated type *)
        (* with three values *)
opns set, cancel, expired : -> TimerSignal
        (* These are the values of the type *)
endtype
```

To implement the type `ExtendedNat`, we use the library type `Integer`. In fact the integers contain more data values, but have the advantage that

they are a standard type in all imperative languages. Moreover, the SPECS implementation of the integers already contains the functions `lt` and `mod`. So, `lt` does not have to be declared at all, while the pragma for `Mod` indicates that this is a user-defined renaming of the standard library function *mod* (in fact, this pragma could also be empty, because LOTOS is not case-sensitive).

```
library Integer endlib

type ExtendedNat is Integer
  opns _Mod_ (*$ mod $*) : Nat, Nat -> Nat;
  (* user renaming of the standard *)
  (* function mod *)
       NumberOfTimers : -> Nat
endtype
```

In our example we assume that `NumberOfTimers` has been given a value elsewhere.

Note that we no longer need the equations defining the semantics of the data, because the semantics are now defined by mapping these types to the same types at the I-CRL level (using the pragmas). The function declarations remain necessary, however, to preserve the static semantic correctness of the LOTOS specification.

On the I-CRL level, the same set of data types and type schemes has been implemented, so pragmas are used to generate the corresponding type on the I-CRL level. In our first example this leads to the sort declaration:

```
(SORT-DCL
   TimerSignal
   (ENUM set cancel expired))
```

The second example is even simpler, because no new sorts are introduced by this type declaration, and only operators that are predefined, or a user renaming of a predefined operator are allowed. Predefined operators need not be explicitly declared at the I-CRL level, and the user renaming is resolved during translation, i.e., every use of the function `Mod` at the LOTOS level is translated to the use of the predefined function *mod* at the I-CRL level. So, this declaration does not generate any I-CRL code.

A more elaborate example declares a variant record with two variants. Types `Calling`, `Called`, and `Module` are supposed to have been defined previously. The type `Boolean` is predefined.

```
type Primitive is
     Calling, Called, Module, Boolean

   sorts
     Primitive (*$ rec vars 2 $*)

   opns
     CON_RQ (*$ mkrec 3 var 1 $*) :
          Calling, Called, Module -> Primitive
     CON_RP (*$ mkrec 1 var 2 $*) :
          Module                  -> Primitive

     calling_of (*$ sel 1 var 1 $*) :
          Primitive -> Calling
     module_of (*$ sel 3 var 1, sel 1 var 2 $*) :
          Primitive -> Module

     is_con_rq  (*$ isvar 1 $*),
     is_con_rp  (*$ isvar 2 $*) :
          Primitive -> bool

endtype
```

The compilation of the sort pragma (*$ rec vars 2 $*) leads to the following sort declaration on the I-CRL level:

```
( SORT-DCL Primitive
 ( VARRECORD
   ( VARIANT-DEF CON_RQ
     (
      ( FIELD-DEF
          XS_v1_f1XPrimitive_op Calling )
      ( FIELD-DEF
          XS_v1_f2XPrimitive_op Called )
      ( FIELD-DEF
          XS_v1_f3XPrimitive_op Module )))
   ( VARIANT-DEF CON_RP
     (
      ( FIELD-DEF
          XS_v2_f1XPrimitive_op Module )))))
```

The translation of operator pragmas is rather straightforward, with the exception of the multiple selector pragma (the `module_of` operator in the above example). This is translated to a case statement (McCarthy expression, in I-CRL terminology).

Similarly there are pragmas to describe list, set, map, and array declarations on the LOTOS level, as well pragmas to describe the predefined func-

tions on these types. Finally, a special pragma allows the user to construct user-defined functions with equations to describe the semantics of these functions.

Translation of control. In order to achieve an efficient implementation of LOTOS specifications, inherent parallelism involving complex multiway synchronization is reduced. For this purpose, closely connected parallel processes of finite control state space are merged into a single state machine. Loosely connected processes can be mapped on a network of state machines, thus avoiding interleaving expansion.[13] The compiler detects processes that are dynamically created and compiles them into separate state machines. However, the user can also identify additional processes that should be implemented as single state machines.

The translation proceeds as follows: After checking syntax and static semantics, the LOTOS specification is further simplified—either to replace run-time expensive LOTOS operators by simpler ones or to ease the translation into state machines, e.g., by reducing the number of different operators. Further, all occurrences of dynamic process creation are replaced by a *fork* command. The next pass transforms the LOTOS processes into a network of state machines, expressed in I-CRL. Using a chart construction technique, a state machine is built directly from the syntax tree of the corresponding LOTOS process. Optimizations on the state machines are performed in the fourth pass: removing unused variables and useless assignments, replacing variables by constants, internal step removal, etc.

State machine construction. Milner introduced extended automata called charts.[14] In a chart, a state may be labeled by zero or more process identifiers; a process identifier X indicates states at which the behavior of the chart may be extended by substitution of another chart for X. Based on this model, we have developed a translation method for LOTOS processes.[11] Milner's labels have been extended to carry not only process identifiers, but also information on the update of data variables and a predicate making the continuation conditional.

For each LOTOS operator, there is a corresponding operation on charts. The final state machine is obtained in an "inside-out" way, building the machine bottom-up from the leaves of the LOTOS syntax tree. The LOTOS process *stop*, which is incapable of communicating, becomes a state machine with one state but without any transition. Similarly, the translation of a LOTOS process instantiation $X[g_1, \ldots, g_m](t_1, \ldots, t_n)$ results in the same chart except that an extension stores the effect of parameter passing in the form of a parallel assignment statement.

There are operations to prefix a transition to a state machine or to glue together the roots of state machines. However, the most interesting operation is recursion resolving. When the body of process X has been completely built, then each extension with process name X in the chart will be replaced by the derivations and extensions (except X) of the root of the chart.

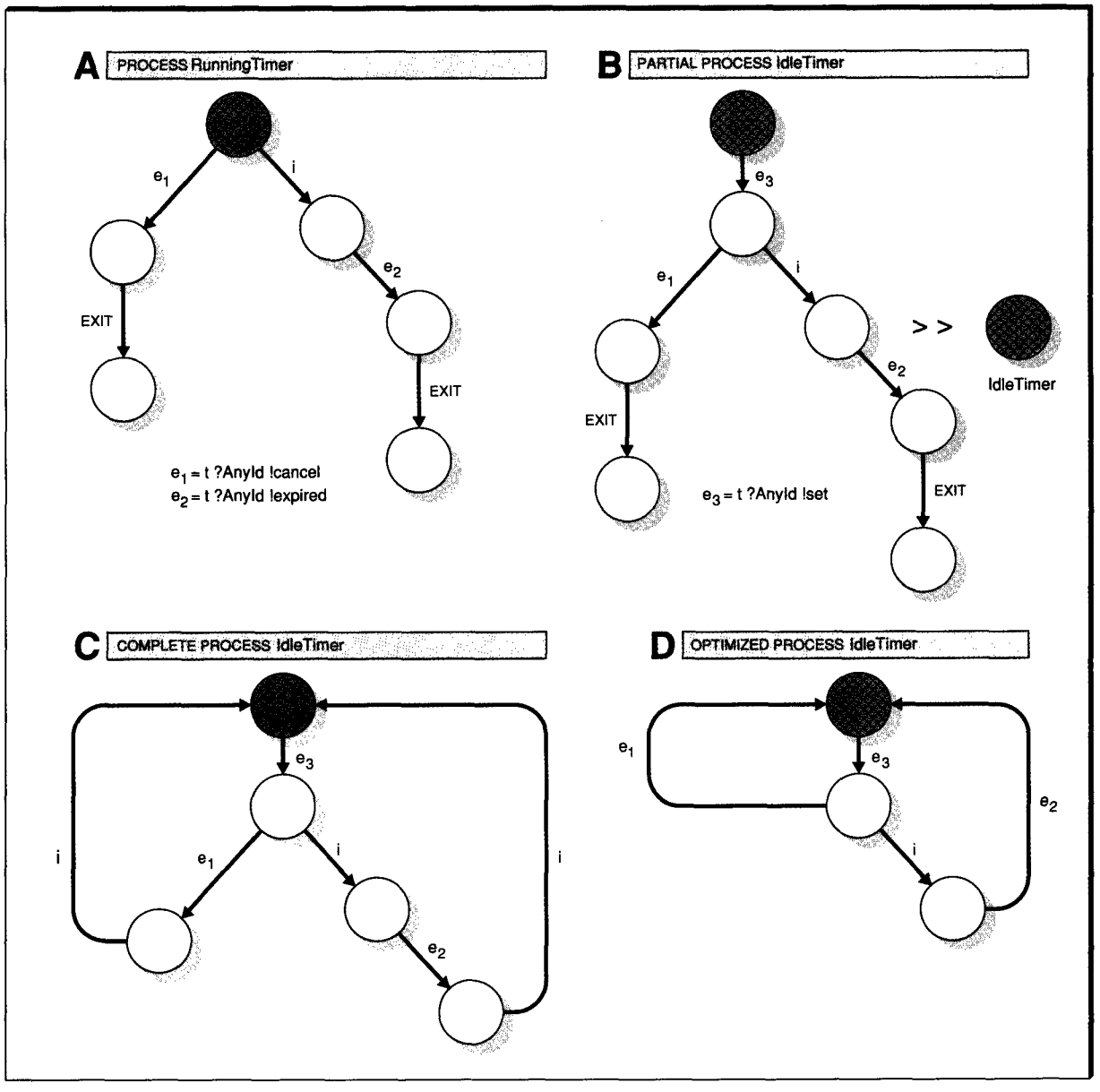
Example. Translating the LOTOS specification from the section on LOTOS, the compiler recognizes that the process `TimerBank` recursively instantiates itself on the left of the parallel operator and replaces the parallel operator in `TimerBank` by a `fork` command. Then the corresponding state machines are separately constructed for processes `TimerBank` and `Timer`.

I-CRL process `TimerBank` is a state machine with only one state. It executes the action *create* (`Timer(TimerId)`) together with the statement `TimerId := Succ(TimerId)` as long as condition `TimerId lt NumberOfTimers` is true.

The process `Timer` is defined as the parallel composition of the processes `IdleTimer` and `Identification`. Its translation requires, therefore, that the processes `IdleTimer` and `Identification` be translated first. We describe the translation of `IdleTimer`, as it illustrates several issues of our state machine construction algorithm. First, a machine for process `RunningTimer` has to be constructed. This state machine can easily be derived from the LOTOS specification and is shown in Figure 5A with the following notational conventions. The nodes of the graph represent states of the state machine; the shaded circle indicates the initial state. The edges represent transitions and are labeled with the corresponding action.

Then the compiler builds a machine for `IdleTimer`. First, a new initial state is created with transition `t ?AnyId !set` leading to the former initial state of `RunningTimer`. This leads to the graph shown in Figure 5B. Next, the sequential composition of

Figure 5 An example of the construction of a state machine



the two processes, expressed by the enabling operator ">>", is resolved by redirecting the two exit transitions to the state labeled `IdleTimer`. The edge label `exit` is replaced by `i`. Then, the compiler resolves the recursive call of `IdleTimer` by merging the node labeled `IdleTimer` with the initial state. This leads to the graph shown in Figure 5C. Finally, after collapsing states that are connected by internal actions only and where the originating state has no other transitions, we get for process `IdleTimer` the state machine shown in Figure 5D.

Once the state machines for IdleTimer and Identification have been built, the process Timer can be translated. Its normal translation would consist in two parallel processes. However, to improve the efficiency of the code to be generated, we could add a pragma (see the section on translation of data in "From LOTOS to I-CRL") to request the compiler to further merge the state machines for process IdleTimer and Identification. Because the state machine for process Identification has only one transition, the size of the resulting state machine would be the same. By this means, the translation would remove the composition operators "||" and ">>" used to express the decomposition of constraints.

From SDL to I-CRL

From a semantic point of view, there is no real gap between SDL and I-CRL; the basic I-CRL processes are extended finite state machines like SDL processes, and I-CRL supports the asynchronous communication paradigm of SDL. Thus, there is no conceptual difficulty to translate the basic concepts of SDL to I-CRL. However, I-CRL is simpler than SDL in several domains, e.g., hierarchical structuring mechanisms. The translator has to take care of these differences. It turns out that the combination of these many small differences makes the definition of the translator a nontrivial task.

Since its first versions in the 1970s, SDL has gradually been enriched with a certain number of constructs that make the language easier to use in various situations, but that makes the tooling of the language expensive. Therefore the full SDL Recommendation[10] is not supported by the compiler; a few nonessential language extensions are not covered. As explained earlier in the section on the SPECS support for data, only a restricted form of the SDL data part is supported. The subset of SDL that is supported is quite large and covers all the essential points of the language.

The translation from SDL to I-CRL is done in two steps. First, the SDL specification is *flattened* or transformed into more basic constructs, then the flattened SDL specification is translated into I-CRL.

The SDL recommendation defines a certain number of constructs of the language (called *additional concepts* in the recommendation) in terms of the primitive constructs of the language. Some of these additional constructs have a direct counterpart in I-CRL, and are therefore not flattened. The other *additional concepts* are flattened, following the rules defined in the recommendation. In order to simplify the symbol tables later on in the translation process, names are made unique within each scope unit.

Translation of the data part. SDL provides direct support for most of the SPECS data types and constructors, either through its library of predefined data or through specific language constructs. For the support of the definition of enumerated sorts and variant records, the syntax of SDL had to be extended. The following example shows how a variant record can be expressed using the SPECS extensions to SDL:

```
NEWTYPE Primitive
  VARSTRUCT
    VARIANT CON_RQ;
      calling_of Calling;
      called_of  Called;
      module_of  Module;
    VARIANT CON_RP;
      module_of  Module;
ENDNEWTYPE;
```

Most SDL type definitions, introduced by the NEWTYPE clause, map straightforwardly into I-CRL sort definitions. For the definition of a new operator f, the following restricted format of SDL equations is to be used:

$$y_1 == e_1, \ldots, y_m == e_m,\ c_1, \ldots, c_p\ ==>\ f(x_1, \ldots, x_n) == e;$$

where: $x_1, \ldots, x_n$ are the formal parameters of the operator f and must be pairwise distinct; e is the resulting expression. $c_1, \ldots, c_p$ $(p \geq 0)$ is a list of conditions. $y_1, \ldots, y_m$ $(m \geq 0)$ are auxiliary variables that may not be circularly defined.

The same restrictions on the format of axioms apply to LOTOS.

For the translation to I-CRL, all the equations defining one operator are grouped together into a case expression, with one case per equation. The conjunction of the translation of the conditions $c_1, \ldots, c_p$ makes up the condition of the corresponding case. The auxiliary variables $y_1, \ldots,$

y_m are rendered through a *LET* construct, a specific I-CRL construct for the definition of local synonyms.

Translation of the control part. Two points of the translation algorithm are highlighted in this section: the representation of the static SDL system structure in I-CRL, and the translation of the SDL state machine.

The static hierarchical structuring mechanisms of SDL have no equivalent at the I-CRL level. In I-CRL, a process can send a message to any other process, provided it knows its process identifier. In SDL, a process can send a signal only to those processes to which it is connected via a signal path. SDL provides several levels of *anonymous* addressing; thus the knowledge of the destination process is not required. To cope with these differences between SDL and I-CRL, a static connectivity table is computed during the compilation process. For each process type P, each signal s and each signal route r connected to P, it indicates the possible destination process types for the signal s sent by P on r. For the example in Figure 4, this table would contain the following entries:

```
...
(accumulator, sum,      R1) -> client
(client,      add,      R2) -> accumulator
(client,      subtract, R2) -> accumulator
...
```

At execution time, a dynamic table that contains—for each process type—the set of process identifiers of all the active processes of that type, is maintained. Thus the SDL signal routes and channels are not explicitly modeled by active components in I-CRL. The effect is better execution performance, at the price of more expensive compilations.

Most SDL behavior concepts (such as process, procedure, and variable) map directly to equivalent I-CRL concepts. Due to the complexity of the state transitions in SDL, states cannot be mapped one-to-one from SDL to I-CRL. Auxiliary states have to be introduced in I-CRL in various places, which correspond to the points between the actions in an SDL transition. All the SDL *INPUT* statements leaving a state are grouped together in a single I-CRL *RECEIVE* command. The SDL process creation command *CREATE* is translated into the I-CRL *create* command and an update is performed on the global dynamic process identifier table mentioned in the previous paragraph.

From I-CRL to C

This translation is described in two parts: the data part and the control part.

Translation of data. In I-CRL, data are described in a functional way, whereas C is a procedural programming language. Thus, the problem addressed is in the area of implementation of functional programming languages, a domain that received much attention in the last few years (see e.g., Reference 15). As this area was not our main domain of interest, a simple and straightforward approach was taken for the compilation of the I-CRL data part. We assume that a product version of the compiler would implement the various optimization strategies developed elsewhere and documented in the literature.

Sorts. The basic I-CRL data sorts are mapped to basic C types. User-defined I-CRL sorts are represented as dynamically allocated data structures and are declared as pointers to an element of this structure. Simple data structures have been used to represent all the sort constructors. Lists, sets, and maps are represented as simply linked lists. Variant records are represented as a *union* of all the variants, grouped together with an *enum* field to indicate which variant is represented; arrays are represented as arrays. A possible enhancement would be to provide a choice between several implementations and to let the user select one, via design annotations in SDL or LOTOS, depending on time and space performance criteria.

Operators. I-CRL operators are mapped to C functions. For each user-defined sort, an implementation of the predefined operators is generated by adapting C templates. The I-CRL constructs for expressions map straightforwardly to C, using the C conditional operator *exp ? exp : exp*, and the C comma operator.

In the setting of dynamically allocated structures, an important issue is memory management. Care has to be taken that allocated memory is freed when no longer in use. A simple approach was chosen: a function is responsible for freeing the memory allocated to its arguments. Common subexpressions are not shared between data structures; thus, an operator can modify its ar-

guments and reuse their allocated memory if useful.

Translation of control. The state machine concept of I-CRL allows a straightforward implementation. Processes and procedures are implemented by C functions. Therefore, a process instantiation as well as a procedure call becomes a function call. Gate names and data values of the process instantiation form the parameters of the C function.

The C function body declares variables for the internal gates and the local state variables of the process, followed by statements realizing the *initialization statement* and a jump to the *initial state* label.

Each *control state*, represented by a label, is a block of statements computing the *rules* that may lead from that state. For each rule, the generated code first checks the local constraints by evaluating the *provided clause*. All enabled actions are passed as event offers to the scheduler. Depending on the action chosen, their effects will be computed, followed by a jump to the successor control state.

An excerpt of a translation of an I-CRL specification to C with the run-time environment is shown below.

```
static void Timer( t, TimerId)
RT_Gate t;
int TimerId;
{ int _s, _last, _cur, _noia;
  boolean _ug;
  RT_Param _pars_0[ 2],  _pars_1[ 2];
  long     _args_0[ 3],  _args_1[ 3];
  RT_EventStruct _evt[ 2];
  int AnyId;

S1: /* State S1 */
  RT_SetParam( _pars_0[ 0], 1, RT_Read, &AnyId);
  RT_SetParam( _pars_0[ 1], 16, RT_Write, set());
  _args_0[ 0] = (long) &TimerId;
  _args_0[ 1] = (long) &NumberOfTimers;
  _args_0[ 2] = (long) &AnyId;
  RT_SetExtEv( _evt[ 0], t, "t", 1, 2, _pars_0,
     _sp_1, 3, _args_0, 0, 0, 0);
  if ( RT_Synch( 1, _evt) == RT_Aborted)
     RT_Terminate( 0, RT_Zombie);
  goto S3;

...
```

Run-time environment

The problem of implementing run-time environments for parallel programming languages in general[16–19] and LOTOS or SDL in particular[20–24] has already been discussed in the literature. In all these efforts, the language compiler assumes an underlying execution environment that serves as an intermediary between the compiler-generated code and the underlying operating system.

Since none of the above environments was available to us or exactly fitted the I-CRL execution model, we decided to design and implement our own LOTOS and SDL specific run-time environment.

Goals of the run-time environment. The goals of the run-time environment were the following:

- To provide a concrete implementation of the execution model advocated in the section about I-CRL. To that effect, active abstractions performing the state machine transitions and the appropriate communication mechanisms had to be provided via a programming interface to the rest of the compilation environment.
- To execute on a concrete, physical computing platform. For reasons of availability and suitability as a prototyping environment, we have chosen Advanced Interactive Executive* (AIX*), IBM's implementation of the UNIX operating system, to provide the underlying execution environment. The run-time environment interfaces, however, are largely independent from the target platform and could be implemented on top of other execution environments.

The next two sections describe the overall functionality of the run-time environment, as well as the implementation of the LOTOS multiway synchronization algorithm and the SDL style message passing.

Run-time environment functionality. The execution paradigm for LOTOS and SDL is focused on a few, fundamental abstractions. Both languages use *processes* as active entities that sequentially execute a piece of program. Processes communicate *asynchronously* by sending messages in SDL, whereas LOTOS advocates a multiway *synchronous* communication model. Quite naturally, the following abstractions appear at the run-time environment interface:

- *Threads* are independently scheduled, quasi-parallel executing units of control where each thread executes one extended finite state machine. LOTOS or SDL processes can thus be mapped to run-time environment threads.
- *Gates* model LOTOS's multiway, synchronous communication paradigm. They provide synchronization abstractions to which individual threads can submit event offers. These are unified according to the LOTOS rules for process synchronization.
- *Messages* support SDL-style asynchronous communication between individual threads.

Using these abstractions, we engineered a programming interface that is used by the code generation phase of the compiler: The I-CRL to C compiler generates C code that invokes the run-time-environment-provided operations to implement the operational semantics of the specification languages. The generated C code is then compiled and linked with the run-time environment C library. Although we only provided one concrete implementation of this library, it is hoped that the interface is sufficiently implementation-independent and could also be supported on other platforms.

Before presenting the run-time environment programming interface in some more detail, a few additional observations explain the need for hierarchical data structures to implement both threads and gates.

Since in LOTOS and SDL, processes can dynamically spawn child processes, the internal organization of the run-time environment organizes threads in a parent-child hierarchy. This allows for certain functions not to affect just one thread, but the entire thread hierarchy for which a given thread is the root. In particular, this can be used to implement LOTOS *disabling* and the SDL *offspring* functionality.

The need for hierarchical data structures also holds for the implementation of gates: the execution of LOTOS processes and their subprocesses is either *interleaved* or *synchronized*, thus creating an arbitrary hierarchy of interleaved and synchronized subprocesses. The following example clarifies the behavior of LOTOS multiway synchronization and indicates how such synchronization can be implemented using the well-known *and-or* tree data structure.

Consider the following LOTOS expression:

$$P := ((Q_0 \ |[g]| \ P_0) \ ||| \ (Q_1 \ |[g]| \ P_1)) \ |[g]| \ ((Q_2 \ |[g]| \ P_2) \ ||| \ (Q_3 \ |[g]| \ P_3))$$

The expression above yields the synchronization topology of Figure 6, assuming that each LOTOS process is mapped onto one run-time environment thread. In Figure 6, we have labeled synchronized processes as *and* nodes and interleaved processes as *or* nodes. The terminal nodes corresponding to the LOTOS processes Q_0, P_0, P_1, Q_2, P_2, Q_3, P_3 are shown as rectangular boxes. An intuitively straightforward interpretation of the synchronization is based on an *and-or* tree behavior of a gate hierarchy. That is, for an *or* or interleave node, any of the present subprocesses might provide the matching event. For a synchronization or *and* node in contrary, all subprocesses must provide a matching event for synchronization to occur.

In the above expression, four processes must thus communicate over gate g for the synchronization to take place. The synchronization event can be established by the following groupings: $\{Q_0, P_0, Q_2, P_2\}$, $\{Q_0, P_0, Q_3, P_3\}$, $\{Q_1, P_1, Q_2, P_2\}$, or $\{Q_1, P_1, Q_3, P_3\}$. All other groupings of processes do not result in a synchronization.

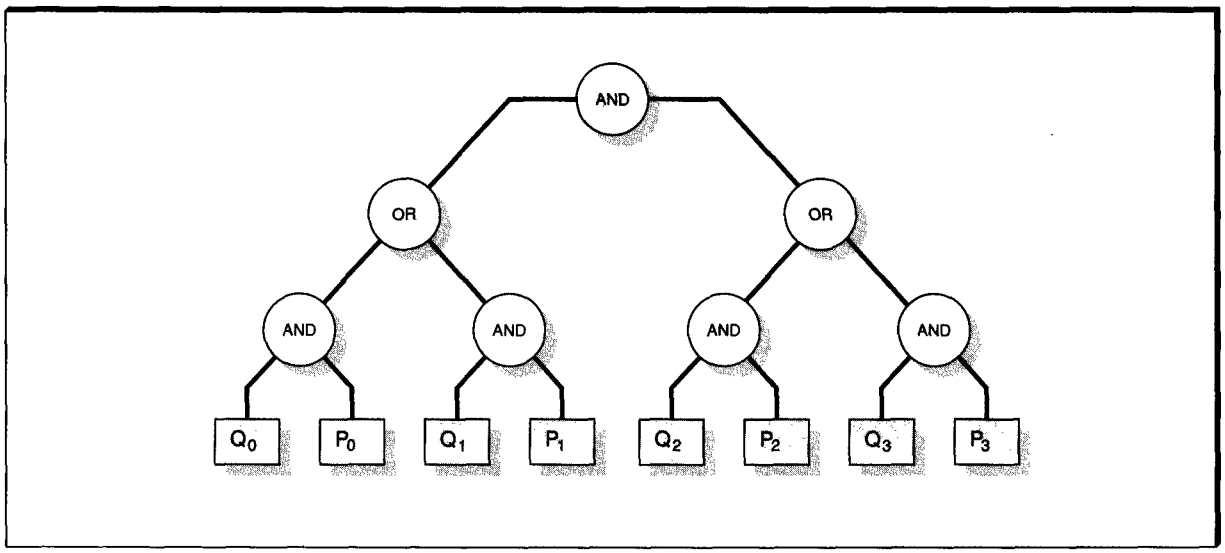
The main functionality of this run-time environment library can now be summarized by the following key operations.

Thread creation. The *create* entry point takes arguments that determine the thread's stack size, its entry point, and its initial arguments. An additional argument indicates whether the thread and any of its child threads can be disabled through the *disable* call (see below). The call returns a handle to the newly created thread. The newly created thread becomes the child of its creator thread, thus creating a thread hierarchy.

Thread termination. A thread and all of its children are terminated via a call to the *terminate* entry. Thread descriptors are deallocated through use of the run-time environment's *dispose* call. A parent thread can wait for the termination of one or all of its children through use of the *join* call.

Thread disabling. Through a call to the *disable* entry, a thread can terminate an entire subhier-

Figure 6 Synchronization topology for sample LOTOS expression



archy of the thread hierarchy. This call specifically implements the semantics of the LOTOS "[>" operator by identifying the subtree of threads to be terminated. (The marking of such subtrees occurs during creation of threads.)

Gate creation. To add a new gate node to a (possibly empty) gate hierarchy, the client may use the *new_gate* call. Its arguments include the type of parallel composition for the processes bound to that gate node, e.g., interleaved or synchronized, the parent gate node, and the expected number of threads to be bound to that gate node. Note that a new gate node must be created only if the synchronization mode differs from the parent node. Otherwise, the old and new node have the same synchronization type and can be combined.

Thread-gate association and disassociation. To bind one or several threads to a gate, the client uses the *bind* call after the gate has been created. To break the association between a thread and a given gate, the run-time environment supports the *unbind* call. As a side effect of the *unbind* call, the gate hierarchy may shrink if an internal gate hierarchy node becomes childless.

Thread synchronization. When a thread is ready to submit one or several synchronization offers at various gates, it uses the *synch* call. The call takes an array of event offer descriptors as its argument. Each offer is recorded at its gate and the pending threads are then tested for possible synchronizations. If no gate is ready to synchronize, the calling thread is suspended until some other thread offers one of the missing synchronization opportunities or until a time out associated with a LOTOS internal event expires. The call's return value indicates which event took place.

Message passing. Threads can *send* messages to any other thread, where these messages are buffered in a queue associated with the receiving thread. To receive a message, the receiving thread invokes the *receive* operation. Additional arguments to the receive operation support encoding of enabling conditions and the SDL SAVE concept.

Timer support. SDL support requires the provision of timers that can be started, tested, and canceled.

In addition to the above operations, which are visible to the compilers, the run-time environment also provides internal functionality for thread scheduling and thread synchronization on condition variables. These are used to implement the exported functions described above.

Within the SPECS project, the run-time environment also supports the mixing of specifications, i.e., interspecification communication.[4] This is implemented by allowing the generation of an SDL type message (or signal) upon occurrence of a LOTOS event and by mapping specific SDL messages into LOTOS event offers. The required extensions to the base run-time environment were straightforward after the desired semantics had become clear.

A last class of operations handles UNIX-like input/output, thus providing means to communicate between a specification and its environment.

Compiler implementation

We first present the platform on which the compiler has been implemented, then highlight specific aspects of each translation step.

Implementation platform. The LOTOS/SDL to C compiler is implemented on the development platform CONCERTO.[25] To introduce CONCERTO, we need some terminology.

The *concrete syntax* of a formal language defines exactly what character sequences are correct expressions of the language, and associates a concrete syntax tree, containing all the terminals, to them. An *abstract syntax tree* is a simplified version of the concrete syntax tree that contains no keywords, and abstracts from certain details that are necessary at the concrete syntax level to guarantee the uniqueness of the parsing. To define certain operations on a language, e.g., translation, it is easier to operate at the abstract syntax level than at the concrete one. A *formalism* is a language whose concrete and abstract syntaxes are formally defined using a grammar-like notation. A *parser* breaks up the source program according to the grammar into constituent parts and represents it by its abstract syntax tree. A *tree transformer* maps an abstract syntax tree of a source formalism to an abstract syntax tree of a target formalism; both formalisms may be the same. A *pretty printer* generates the concrete syntax of an abstract syntax tree.

CONCERTO has three important components to support the development of a compiler:

- A metalanguage (METAL)[26] and pretty printing metalanguage environment (METAL-PPML),[27] to define a formalism
- A structure processor, also called virtual tree processor (VTP), to operate on an abstract syntax tree
- A tree transformer (TRANS), to define recursive tree transformations

A METAL specification defines the grammar of a formalism of interest, e.g., LOTOS, in a declarative fashion. From a METAL specification, CONCERTO produces a parser that builds an abstract syntax tree from any correct input program of the specified formalism. The user can also specify the printing rules, also called unparsing rules, in a PPML specification for each node of the tree. Based on these rules, CONCERTO generates a pretty printer.

The VTP component provides various kinds of operations that can manipulate the tree. A VTP tree always belongs to a certain formalism. The organization of VTP is based on the concepts of objects and classes. An object of one class can be converted to a compatible object of another class. The browsing functions enable walks through a tree without modifying it. Of course, there are tree construction and modification functions. There are also pattern matching facilities. A major feature is that CONCERTO can provide simultaneous support for several formalisms; this is necessary for the building of a compiler.

TRANS allows the user to concisely specify a mapping from one formalism to another one as a set of abstract syntax tree transformation rules. A transformation rule consists of a source pattern and a target pattern. Both patterns may be arbitrarily deep and contain tree variables. Variables in the source pattern are unified with elements of the input tree that are to be transformed; the transformation rules are recursively applied to these variables, and the result is substituted for occurrences of the variables in the target pattern. Moreover, it is possible to call arbitrary VTP functions for complex configurations.

LOTOS to I-CRL implementation. The LOTOS to I-CRL compiler is implemented as a four-pass compiler. The first pass performs the flattening function as defined by the ISO standard.[8] The static semantics check is also done here. In the second pass, all parallel expressions that spawn off a new process are replaced by semantically equivalent *fork* commands; the result can then be mapped directly to I-CRL. The third pass trans-

forms LOTOS abstract syntax trees into I-CRL abstract syntax trees. To implement the recursion solving, an extra argument—a list of process identifiers and actual gates—is added to the translation function. This argument records the history of process instantiations; thus the compiler can recognize when the chain of process dependencies is completed. In the fourth pass, some optimizations on the state machines, e.g., copy propagation, are performed.

SDL to I-CRL implementation. The SDL to I-CRL compiler is implemented in two passes. The first pass does the flattening as described earlier in the section "From SDL to I-CRL"; it transforms SDL into SDL. The second pass transforms SDL into I-CRL. It has been implemented using the TRANS tool. The transformation rule of the SDL root node calls a number of VTP functions that scan the SDL tree to compute the SDL connectivity table also described earlier, before yielding an I-CRL tree. This table is stored in a global variable, which is referred to in other TRANS rules, e.g., the one defining the translation of the SDL OUTPUT statement.

I-CRL to C with the run-time environment. This translation is implemented in a single pass. Its implementation is rather straightforward. The C formalism provided by CONCERTO is used as target formalism. A global variable *sel-preds* is used to collect all the selection predicates that occurred during the translation. At the end, for every element in *sel-preds*, a function definition is generated. These definitions are inserted in the declaration part of the C-tree.

Run-time environment implementation

The run-time environment provides the concrete execution environment for the compiled LOTOS and SDL code with IBM's AIX operating system environment. In our implementation, *threads* are implemented as independently scheduled coroutines. Each thread executes on its private stack, but all threads share the same AIX address space. One rationale for this implementation choice was efficiency in thread-related operations: since all operations are within one AIX address space, thread creation, scheduling, and destruction only necessitate a few procedure calls and no trap into the operating system.

The reason for not using one UNIX process per thread was efficiency. We could have used UNIX's shared-memory primitives to implement message passing and multiway synchronous communication, but the costs of these operations, as well as process management in general, would have been considerably higher. Another unexplored design dimension would have been a distributed implementation of the execution environment. This would have introduced not only further inefficiencies, but also considerably increased development complexity. Particularly, the sharing of SDL process variables and the LOTOS multiway synchronization are hard to implement in a truly distributed environment. Unless a centralized scheduler operating in a distributed environment is used, the multiway synchronization would indeed require a two-phase commit protocol—the cost of which appeared prohibitive.

The scheduling of run-time-provided threads is explicit, i.e., we do not implement preemptive context switching as this would have required a reentrant implementation of the standard C libraries. Therefore, threads are only rescheduled at well-defined entry points to the run-time environment.

Most aspects of the thread-related functionality are straightforward to implement and do not differ notably from other UNIX implementations of lightweight threads.[28–30]

SDL-style message passing can be implemented in a straightforward way: threads have an associated message queue into which a sender deposits a message. The receiving process simply dequeues that message and processes it as indicated by the SDL state machine. Since all threads operate in one AIX address space, message passing is simply implemented by copying in shared memory. The support for the SDL *SAVE* construct is provided by having the compiler generate a bit pattern that indicates the types of messages that shall be saved or received.

The implementation of LOTOS-style multiway synchronization with possibilities of value matching, value generation, or value passing, however, was of greater challenge.

As shown in the section on run-time environment functionality, LOTOS event unification can be mapped onto an *and-or* tree data structure. Individual threads propose a set of events, which are described as structures containing the necessary

information for value matching, value passing, or value generation. After a thread proposes a new set of events, the run-time environment attempts to solve the *and-or* tree in a bottom-up fashion. At each layer of the tree, event unification is attempted. This involves matching the data parameters of individual events according to the LOTOS rules for value matching, value generation, and value passing.

Once a set of events with matching data values has been found within a subtree, the run-time environment invokes possible selection predicates for which the compiler generated the corresponding C functions. It is only when the events satisfy these selection predicates that events are propagated upward in the tree, otherwise a new combination of event offers is explored. If no combination of event offers succeeds, the algorithm fails and no synchronization of threads occurs.

Value generation during the event unification is based on random value generation. We have not implemented a more general constraint-solving algorithm that would interpret the selection predicates as a set of constraints to be solved and assign suitable values to unbound variables during event unification. If event unification fails because of unsatisfied selection predicates, a warning message is generated.

The interpretation of the gate hierarchy as an *and-or* tree determines the inherent complexity of the event unification algorithm. Without taking into account selection predicates, complexity of the algorithm is then nondeterministically polynomial (NP-complete).[31] Selection predicate evaluation makes the event unification undecidable for the most general case.

The run-time environment only provides minimal support for data. It assumes that all data values are represented as 32-bit entities, i.e., either as 32-bit values or as pointers to multiword, heap-allocated structures. This allows for uniform treatment of data within the run-time environment. The data operations are implemented through regular C macros or functions, generated by the data part of the I-CRL to C compiler.

Event unification for LOTOS and SDL message passing requires that client-program-defined, sort-specific functions be installed in the run-time environment during initialization. Thus, the compiler generates some code to install such sort-specific routines for equality testing, random value generation, copying, and deallocation in internal look-up tables. These are then invoked when necessary. This approach represents another case in which tight coupling between the run-time environment and the code-generation phase was necessary.

The run-time environment as described has been implemented on IBM's RISC System/6000* family of processors, on IBM RT* (RISC technology) workstations, as well as on the SUN-3** architecture. With the exception of three assembler routines needed for thread context switching, all the code is written in C. The overall library is approximately 10 000 lines of moderately commented C code.

Conclusion

We have designed translations from SDL and LOTOS to a procedural programming language, augmented with specific run-time support. We have shown that it is feasible to map the abstract language LOTOS to an execution-oriented model consisting of communicating state machines, which is also a natural target for SDL. This intermediate representation allows the sharing of code generation for LOTOS and SDL and allows the integration of mixed specifications. This approach has already been applied to programming languages, e.g., the IBM PL.8 compiler,[32] but is new for specification languages.

LOTOS specifications are quite often written in the constraint-oriented specification style. This style structures the system specification into a set of parallel processes, each expressing an independent system property. However, existing LOTOS compilers[20,21] map LOTOS processes on C functions, which are executed as coroutines. Accordingly, the structure of the generated code closely resembles the architecture of the LOTOS specification. In particular, the process structure as given by the parallel, enabling, and disabling operators is not changed. We have achieved an efficient implementation of these specifications by reducing the inherent parallelism.

As there are no efficient implementation techniques for general algebraic data specifications, we introduced a set of basic types and type constructors into the two specification languages.

This contrasts with other LOTOS and SDL compilers (e.g., References 21 and 23), which oblige the user to implement the data functions. It turned out that our approach also simplifies the task of specifying data algebraically.

The use of a high-level language-engineering platform has several advantages over the more classical approach that uses lexical analyzer and parser generators such as LEX and YACC (available on most UNIX platforms). Parsers, unparsers and syntax-directed editors were generated automatically from the grammar of the specification languages. The internal representation of the abstract syntax trees was predefined and encapsulated by a rich set of access functions; this eased the distributed development of the compiler over three sites. Before being implemented, the compilers had been specified in a VDM-like notation;[33] the derivation of the implementation was simplified by the use of a high-level transformation language and support tool (TRANS). The drawback was a rather long learning period, due to the complexity of the platform.

The compiler described in this paper can already be used for the validation of a formal specification in a rapid prototyping approach, as advocated for example in Boehm's spiral model for software development,[34] which includes prototyping as a means of risk reduction. To increase the adequacy of the compiler for the generation of product code, the following items require further work. For the data part, the user should be allowed to choose between several implementations of the dynamic data structures (list, map, set), depending on the nonfunctional constraints, such as performance and reliability. It should also be possible to refer to existing libraries of C type and function definitions. For the behavior part, pragmas defining communication with the environment should be introduced. Thus, the sending of an SDL signal to the environment or the synchronization on a LOTOS gate could trigger the call of a specific C function, realizing, for example, an input/output operation. In the current version of the compiler, the user has to write C functions realizing this mapping and to integrate them manually into the generated code in order to obtain this effect.

Acknowledgments

The work presented in this paper was carried out as part of the SPECS project, to which many people from all over Europe contributed. This large cooperative effort was made possible by the support from the RACE program of the European Communities. Georg Karner, Heinz Saria, and Sylvia Suchanek from Alcatel Austria—ELIN Research Center, Yves Trémolet from GSI-TECSI (France), Palle Christensen, Bo Bichel Nørbæk, and Anders Olsen from TFL (Denmark), and Didier Dupuy d'Angeac, Bertrand Gruson, and Richard Ngo-si-xuyen from IBM France participated with the authors in the development of the prototype described in this paper.

The work in this paper was supported by the European Community's SPECS RACE R1046 project. The paper reflects the views of the authors.

Cited references

1. M. Dauphin, M. M. Marques, A. P. Mullery, and P. Rodier, "The European Telecommunications Research and Development Program RACE and Its Software Project SPECS," *IBM Systems Journal* **31**, No. 4, 649–667 (1992, this issue).
2. R. Reed, W. Bouma, M. M. Marques, and J. Evans, "Methods for Service Software Design," *Proceedings of the Eighth International Conference on Software Engineering for Telecommunications Systems and Services*, London (1992), pp. 127–134.
3. B. Blanchard, M. Dauphin, B. Gruson, and S. Simon, "SPECS: un Environnement de Spécification et de Programmation pour les Systèmes de Télécommunications," *Proceedings of the Fourth International Conference on Software Engineering and Its Applications*, EC2, Toulouse, France (December 1991), pp. 88–112 (in French).
4. H. Saria, H. Nirschl, and C. Binding, "Mixing LOTOS and SDL Specifications," K. R. Parker and G. A. Rose, Editors, *Proceedings of the Fourth International Conference on Formal Description Techniques*, Elsevier Science Publishers B.V., Amsterdam (1992), pp. 425–439.
5. H. Ehrig and B. Mahr, *Fundamentals of Algebraic Specification I: Equations and Initial Semantics, II: Module Specifications and Constraints,* Springer-Verlag, NY, EATCS Monograph (1985, 1990).
6. R. Milner, *Communication and Concurrency*, Prentice-Hall, Inc., Englewood Cliffs, NJ (1989).
7. C. A. R. Hoare, *Communicating Sequential Processes*, Prentice-Hall, Inc., Englewood Cliffs, NJ (1985).
8. *Information Processing Systems—Open Systems Interconnection—LOTOS—A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour,* International Standard 8807, ISO, Geneva (February 1989).
9. A. S. Tanenbaum, *Computer Networks*, second edition, Prentice-Hall International Editions, Prentice-Hall, Inc., Englewood Cliffs, NJ (1989).

10. *Functional Specification and Description Language, (SDL)*, CCITT Blue Books Volume X—Fascicle X., Recommendation Z.100 and Annexes A, B, C, D, and E, Geneva (1989).
11. G. Karjoth, "Implementing LOTOS Specifications by Communicating State Machines," W. R. Cleveland, Editor, *Proceedings of the Third International Conference on Concurrency Theory*, CONCUR '92, Springer-Verlag, Berlin/Heidelberg (1992), pp. 386–400. Lecture Notes in Computer Science No. 630.
12. G. V. Bochmann and J. Gecsei, "A Unified Model for the Specification and Verification of Protocols," *Proceedings of IFIP Congress* (1977), pp. 229–234.
13. H. Garavel and J. Sifakis, "Compilation and Verification of LOTOS Specifications," *Protocol Specification, Testing, and Verification X*, Elsevier Science Publishers B.V., Amsterdam (1990), pp. 359–376.
14. R. Milner, "A Complete Inference System for a Class of Regular Behaviours," *Journal of Computer and System Sciences* **28**, 439–466 (1984).
15. S. L. Peyton Jones, *The Implementation of Functional Programming Languages*, Prentice-Hall, Inc., Englewood Cliffs, NJ (1987).
16. T. P. Baker, *An Ada Run-Time System Interface*, Technical Report 85-06-05, Department of Computer Science, University of Washington, Seattle (June 1985).
17. E. W. Olsen and S. B. Whitehill, *Ada for Programmers*, Reston Publishing Company, Reston, VA (1983).
18. R. C. Holt, *Concurrent Euclid, the UNIX System and TUNIS*, Addison-Wesley Publishing Co., New York (1983).
19. M. Weiser, A. Demers, and C. Hauser, "The Portable Common Runtime Approach to Interoperability," *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, Litchfield Park, AZ, ACM SigOps (December 1989), pp. 114–122.
20. J. A. Mañas and T. de Miguel, "From LOTOS to C," *Formal Description Techniques I*, K. Turner, Editor, Elsevier Science Publishers B.V., Amsterdam (1988), pp. 79–84.
21. S. Nomura, T. Hasegawa, and T. Takizuka, "A LOTOS Compiler and Process Synchronization Manager," *Protocol Specification, Testing, and Verification X*, Elsevier Sciences Publishers B.V., Amsterdam (1990), pp. 165–184.
22. J. Mañas and T. de Miguel, "The Implementation of a Specification Language for OSI Systems," *1988 International Zürich Symposium on Digital Communications*, B. Plattner and P. Günzburger, Editors, IEEE (March 1988), pp. C3.1–C3.5.
23. V. Encontre, "GEODE: An Industrial Environment for Designing Real Time Distributed Systems in SDL," *SDL '89: The Language at Work*, M. M. Marques and O. Faergemand, Editors, Elsevier Science Publishers B.V., Amsterdam (June 1989), pp. 105–115.
24. K. Miyake, Y. Shigeta, W. Tanaka, and H. Hasegawa, "Automatic Code Generation from SDL to C++ for an Integrated Software Development Support System," *Formal Description Techniques III*, E. Vazquez, J. Quemada, and J. Mañas, Editors (1991), pp. 555–558.
25. E. André, A. Conchon, and P. Andrieu, "Concerto: Balancing Functionality with Technology," *Proceedings of Software Tools Conference*, Wembley, June 1990, Blenheim, London (1990).
26. G. Kahn, B. Lang, B. Mélèse, and E. Morcos, "METAL: A Formalism to Specify Formalisms," *Proceedings of INRIA Seminar*, Aussois (April 1983), pp. 169–204.
27. A. Conchon and E. Morcos, "PPML (Pretty Printing MetaLanguage): A General Formalism to Specify Pretty Printing," *Proceedings of IFIP '86*, Tenth World Computer Congress, Dublin (1986).
28. C. Binding, "Cheap Concurrency for C," *ACM SIGPLAN Notices* **20**, No. 9, 21–26 (1985).
29. E. C. Cooper and R. P. Draves, *C Threads*, Technical Report CMU-CS-88-154, Computer Science Department, Carnegie Melon University, Pittsburgh (June 1988).
30. *Lightweight Process Library, SUN OS Release 4.0 Manual*, SUN Microsystems Inc., Mountain View, CA (1987).
31. E. Horowitz and S. Sahni, *Fundamentals of Computer Algorithms*, Computer Software Engineering Series, Computer Science Press, Inc., Rockville, MD (1978).
32. M. Auslander and M. E. Hopkins, "An Overview of the PL.8 Compiler," *Proceedings of the Sigplan '82 Symposium on Compiler Writing*, ACM (June 23–25, 1982).
33. C. Jones, *Systematic Software Development Using VDM*, Prentice-Hall, Inc., Englewood Cliffs, NJ (1986).
34. B. W. Boehm, "A Spiral Model of Software Development and Enhancement," *IEEE Computer* **21**, No. 5, 61–72 (1988).

Accepted for publication June 18, 1992.

Carl Binding *Union Bank of Switzerland, LHIS/LHDD-BSZ, Bahnhofstrasse 45, 8021 Zürich, Switzerland (electronic mail: bsz%ubszh.uu.ch@chsun.chuug.ch).* Dr. Binding received the degree of Dipl. El. Ing. ETH from the Swiss Institute of Technology in Zurich, Switzerland, in 1981, and the degrees of M.S. (1985) and Ph.D. (1987) in computer science from the University of Washington, Seattle. From 1989 to 1992 he was with the IBM Zurich Research Laboratory where he worked on the compilation and execution of the LOTOS specification language. Before that he held a position at the Olivetti Research Center, Menlo Park, California, where he was involved with multimedia user interface system architecture and implementation.

Wiet Bouma *PTT Research, Neher Laboratories, P.O. Box 421, 2260 AK Leidschendam, the Netherlands (electronic mail: L.G.Bouma@research.ptt.nl).* Mr. Bouma is a senior project leader at the research laboratory of the Dutch PTT. After graduating from the University of Amsterdam, he worked there for several years in research and education, initially in pure mathematics (differential topology); in 1981 he switched to a focus on the application of formal logic to computer science, more specifically: Floyd-Hoare Logic, foundations of logic programming, and algebraic methods in semantics. He left the university in 1988 to join PTT Research, working since then on the SPECS project, first as a project member and later as a partner leader. Since January 1992 he has led the PTT team for the RACE-II project SCORE (Service Creation in an Object-Oriented Reuse Environment). SCORE is concerned with the development of an environment to implement the Service Creation Function in the CCITT and ETSI IN models. His current interests are in making formal methods in software engineering amenable to human use, and in the adaptation of temporal logic to the verification of concurrent software.

Michel Dauphin *Compagnie IBM France, IBM Centre d'Etudes et Recherches, 06610 La Gaude, France (electronic mail: dauphin@vnet.ibm.com).* Mr. Dauphin is a consultant on software development processes and tools at the IBM La Gaude laboratory. He joined IBM in 1984 in the Advanced Technology department, where he contributed to a development and execution environment for fault-tolerant telecommunication software. In 1987, he joined the IBM RACE team, where he participated in the SPECS project in the areas of semantics, analysis, and implementation of specifications of telecommunications systems. Since 1991, he has been the technical coordinator of the SPECS project. He has published several papers on the application of formal methods and languages to telecommunications software development. Mr. Dauphin received his engineering degree from the Ecole Polytechnique, Paris, in 1984. He won a First Prize in the International Mathematical Olympiad in Washington D.C. in 1981. His main research interests are in discrete mathematics, theoretical computer science (especially concurrency), logic programming, and formal methods.

Günter Karjoth *IBM Research Division, Zurich Research Laboratory, Säumerstrasse 4, 8803 Rüschlikon, Switzerland (electronic mail: gka@zurich.ibm.com).* Dr. Karjoth is a research staff member in the Communications & Computer Science department, IBM Research Division, IBM Zurich Research Laboratory. He received his Diplom in computer science in 1980 and his Ph.D. in 1987 from the University of Stuttgart, Germany, where he worked as a research scientist at the Institute of Computer Science from 1980 to 1986. Since 1986, he has been working at the IBM Zurich Research Laboratory on the application of formal methods in the development of communication protocols. Dr. Karjoth was actively involved in the development of the formal description technique LOTOS in ISO between 1981 and 1985. In 1986, he was a visiting scientist at the Swedish Institute of Computer Science, Kista, Sweden. His interest is in the modeling of distributed systems, their validation and implementation.

Yan Yang *PTT Research, Neher Laboratories, P.O. Box 421, 2260 AK Leidschendam, the Netherlands (electronic mail: Y.Yang@research. ptt.nl).* Ms. Yang is a research staff member in the Computer Science department of Tele-Informatics Division of PTT Research, the Netherlands. She received her master's degree in mathematics at the Technical University of Delft, the Netherlands, in 1985. She joined PTT Research in the same year and has worked on the application of formal methods, verification and implementation of algebraic specifications, temporal logic, and conformance testing.

Reprint Order No. G321-5492.