Annales Univ. Sci. Budapest., Sect. Comp. 17 (1998) 137-154

KANNEL – A LANGUAGE FOR TUNING PROTOCOLS

K. Granö (Jyväskylä, Finland)
J. Harju (Lappeenranta, Finland)
T. Järvinen (Jyväskylä, Finland)
T. Larikka (Lappeenranta, Finland)
J. Paakki (Helsinki, Finland)

Abstract. Modern communication protocols are rather complex and large software products. To manage the inherent complexity of protocol engineering a number of application-orientated methods and software tools have been developed for this discipline. This paper presents Kannel, a language for protocol engineering. Also an application to ISDN protocol LAPD using Kannel is presented.

1. Introduction

The key aspect in building distributed applications or computer networks is to specify and implement communication protocols between the involved objects. *Protocol engineering* is a term often used for the general discipline of developing communication software. The foundation of protocol engineering is the use of methods and tools that have a rigoruos, formal basis.

The communication software products resulting from the modern multi-layer protocol specifications are large and complex. They contain different types of tasks, some of which appear to be easy to generate automatically, while some are better suited for hand-coding. In protocol engineering an abstract specification is used to derive an executable program in the target environment. This higher level specification should be strictly formal to avoid ambiguities and to allow computerized processing. Due to the distributed, open environment the specifications must be so unambiguous and complete that

two implementations, perhaps produced by different vendors, can communicate with each other without extensive preliminary tests.

Recent active research on application-orientated languages promotes the idea that application areas with specialized nature, such as protocol engineering, should be supported with dedicated tools founded on high-level concepts that match with the central characteristics of the applications. This view is in sharp contrast with the simplistic approach of engineering every application using general-purpose languages, such as C.

The Kannel language provides explicit support for the specialized area of protocol engineering. It aims at covering the whole discipline in a uniform manner thus eliminating the need for using several different languages and several unrelated tools to implement the protocol. The most notable features of Kannel are hierarchical finite state machines, distribution, interfaced layering and facilities for encoding/decoding the data contents of protocol data units. In the following sections we give a short description of the Kannel language and apply it to specify a complex protocol (LAPD). Then we compare Kannel with languages having similar target area and draw a number of conclusions.

2. General overview of Kannel

The core of protocol engineering is the use of formal *specification languages*. These languages must be abstract enough to hide those implementation details that are environment specific. There are four well-known formal specification languages: SDL [2] which is based on the extended state transition model, Estelle [5] which also is based on the state transition model, LOTOS [4] which is based on the CCS process algebra, and ASN.1 [14] which is used to describe the format of data units.

While there has been extensive research and development efforts on protocol design, the proposed languages usually have only a restricted focus of application. For instance, Estelle and SDL can be used for defining the internal control behavior of a protocol entity (as an extended finite state machine) with only a very abstract notion of data, and ASN.1 is intended solely for describing the conveyed data units without specifying any operational characteristics of the protocols. LOTOS in turn is too abstract for being implemented efficiently with current technology.

The Kannel language is an evolving attempt to overcome the shortcomings mentioned above. In protocol engineering the usual approach has been to separate the design of a protocol from its implementation issues and to consider functional logic and concrete data units as orthogonal issues. We believe that

a unified scheme which would provide tools for *all phases* in protocol design is a desirable goal. To reach this goal Kannel integrates full-fledged protocol engineering facilities with general software engineering principles:

object-orientation: Kannel supports the *class* and *object* concepts of object-orientated languages. There are two fundamental object categories in Kannel: *value objects* are always represented by value, while *reference objects* are represented by reference. This division is a compromise between efficiency and orthogonality - representing all objects via references would be inefficient, but it still is desirable to have the primitive types such as *integer* part of the class system. In addition, objects may be *local* or *distributed*; this distinction affects the way the object can be communicated with. Local objects are flexible to use, but do not fulfill strong distribution semantics. Kannel has adopted its notion of classes most notably from the Sather language [13], in particular the facility of incremental superclassing and the explicit separation of (multiple) inheritance from code reuse, as proposed in [1].

visuality: The visual components of Kannel support the design of behavioural components of a protocol. It can be argued that visual notation can greatly enhance the understandability of complex constructs [11]. This seems to be especially true for state machines.

distribution: In Kannel the service interfaces provided by a distributed class are separated from their implementation into a *channel*. This decision has the following advantages: the unidirectional nature of messages is supported better. By giving different *views* into the service related messages going in different directions may be syntactically attached together (see below). This leads to a more concise representation of the *mutual* relationship between a client and its server. In addition, when channel is made an explicit language construct, we may better grasp the notion of *transfer syntax*. Lastly, the view concept provides static security and more readable specifications.

communication: In Kannel a transfer syntax is a class definable by the protocol writer. Each transfer syntax specifies a method of encoding/decoding all values of both basic and compound types. Functions for encoding and decoding data types are automatically created and applied when needed. The idea of transfer syntax in Kannel is similar to that in ASN.1. However, the implementation of this idea is more high-level and advanced as a language.

concurrency: Kannel supports concurrency in terms of *active objects*. At most one thread of execution is active within each active object at a time. This approach, although excluding e.g. concurrent read access, provides a simple and safe solution to mutual exclusion problems. Also, the concept of protocol layers are supported in terms of active objects.

functional logic: Kannel adopts the statechart formalism [10] with some modifications as the means to describe the concurrent semantics of an active object. A statechart specifies the exact set of messages the object will respond to in a given state and thus maintains its integrity - this can be viewed as stating as a precondition those messages that are valid for a given state. The communication model is asynchronous so as to offer greater efficiency and flexibility in communication operations.

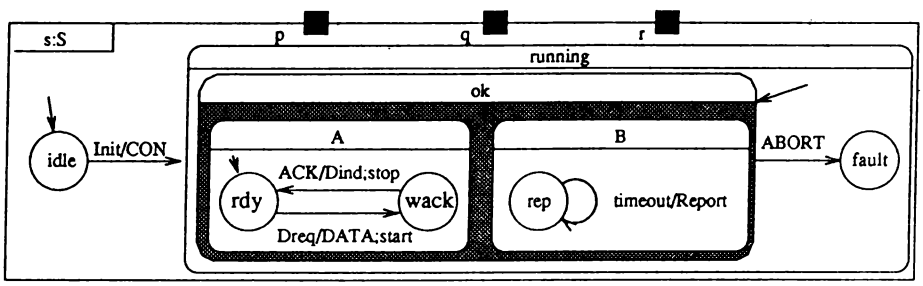

Fig.1. A statechart

As an example of Kannel let us introduce a simple statechart. Statecharts are a visual formalism for specifying the structure and behavior of reactive systems, such as protocols. They differ from conventional finite state machines in the following respects: *hierarchical structure* - a state may be decomposed into a set of refined constituent states and provide abstraction by forming "black boxes"; *XOR-decomposition* - a group of transitions from constituent states may be combined to a single transition in a superstate; *AND-decomposition* - the exponential blow-up in the number of states in large state machines is solved by dividing the states into *concurrent* state machines that act in parallel. This is sometimes referred to as *orthogonality*.

An example of the statechart formalism is given in Fig. 1 which describes a simple transfer protocol with acknowledged packets. The protocol starts in the *idle* state and proceeds to state *running* upon receiving the *Init* message (also causing emittance of the message *CON*). As Fig. 1 shows state *running* is structured, consisting of two internal states. The choice between these states is shown by the *default arrow* inside state *running*. In this case the default state is *ok* which itself appears to be structured, this time using AND decomposition (note the gray background) into two orthogonal substates A and B. The orthogonality implies that A and B act together upon receiving a message.[1] Returning to the example, the entered compound pair is (*rdy*, *rep*). In this

[1] In the general case two orthogonal states with n and m substates require $n * m$ states when expressed as a conventional state machine.

state the chart accepts the message *Dreq* requesting data transfer. (Technically, the message *timeout* may also be accepted in state (*rdy*, *rep*), although in this context it is unlikely since it cannot arrive before the message *start* has been sent to the timer entity.) Transfers are acknowledged (message *ACK*) and the acknowledgements are guarded with a timer (messages *start*, *stop* and *timeout*). Note how the transition for the *ABORT* message is associated with all states within *ok*: this provides a concise way to group together transitions shared by several substates. The corresponding textual Kannel code looks as follows:[2]

```
process S
ports p:...; q:...; r:... is
action state (idle, running) is
state idle arcs
  Init - > { p.send(CON); running } end
state running(ok, fault) is
  and state ok(A, B) is
    state A(rdy, wack) is
      state rdy arcs
        Dreq - > {p.send(DATA);
          r.send(start);
          wack
        }
      end;
      state wack arcs
        ACK - > {q.send(Dind);
          r.send(stop);
          rdy
        }
      end
    end;
    state B(rep) is
      state rep arcs
        timeout - > q.send(Report) end
    end
```

[2] While providing a number of visual facilities, Kannel also has a purely textual syntax.

```
      arcs
        ABORT - > fault
      end;
      state fault end
    end
end
```

Statecharts in Kannel provide for other expressive features as well, such as incrementality with new states and transitions. For reasons of space we omit these advances as well as a more complete description of the Kannel language. The current form of Kannel is reported in [9].

3. LAPD application in Kannel

The LAPD protocol [6] defines the data link layer protocol used in the ISDN protocol suite. It was chosen as an example protocol to be specified in Kannel for several reasons: firstly, the compact representation of LAPD frames requires a mapping from the high-level protocol data structures into a sequence of bits, that is a transfer syntax. Secondly, flexible properties of LAPD such as support for several logical links between the terminal equipment and the network call for the dynamic treatment of LAPD connections. This is accomplished with the routing method mechanism of Kannel. Thirdly, the logical structure of LAPD is relatively independent of a particular physical layer implementation and is thus a potential reusable component. Kannel supports refinement of communication channels which is the key mechanism for protocol reuse.

Due to space constraints the following example omits the actual LAPD statechart which, although structured, is quite formidable in size. Instead we highlight the Kannel mechanisms for the above aspects of LAPD.

3.1. General framework

A Kannel specification is *closed* in the sense that it specifies all the communicating entities - if one of these entities is external, it is reflected in the specification by omitting its internal details. This contrasts conventional protocol techniques which emphasize a single-entity (usually a stack) view of the communication framework. This is reflected in Fig. 2 which depicts the

ISDN layer 2 setup.[3] The peer entities *a* and *b* communicate via the abstract LAPD_APDUs channel and provide service to their respective users via the LAPD_SERVICE channel. The special nature of the peer channel is exposed in the textual Kannel equivalent of Fig. 2.

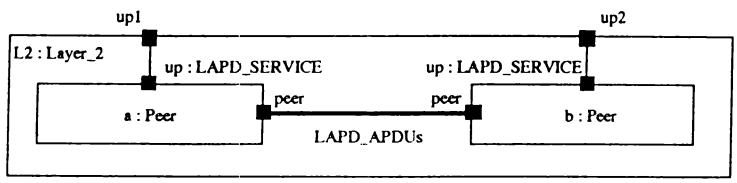

Fig.2. Top level LAPD specification

```
process Layer_2
ports
  up1,
  up2: LAPD_SERVICE(in user; out provider)
is
  process Peer...end Peer; – details omitted
assoc a,b; Peer is
  a.up and up1;
  b.up and up2;
  a.peer and separate b.peer
end Layer_2
```

The **assoc** mechanism relates the ports of different communicating processes together and also provides special information about the exact nature of the relationship, this leads to strict typing between the distributed components. By specifying that the association is *separate* we indicate that the processes cannot communicate via shared memory, instead some other mechanism must be employed (see Section 3.4).

The ports convey messages that are regarded as instances of their respective types which in turn are normal Kannel classes. The messages are grouped into unidirectional *views* that are the building blocks of a *channel* such as LAPD_SERVICE. In the port declaration the role of the process is fixed, thus we state that ports *up*1 and *up*2 will receive messages in view *user* and send

[3] As we are not specifying the user side of layer 2 the *up1* and *up2* ports in Fig. 2 are unconnected. However, deriving an implementation requires that also these ports be connected.

messages in view *provider* of the LAPD_SERVICE channel. Syntactically the channel definition looks the following:

```
channel LAPD_SERVICE is
  view user is
    estreq: dynamic DL_EstablishRequest;
    udtreq: DL_UnitDataRequest - etc.
  end user;
  view provider is
    estind: DL_EstablishIndication;
    udtind: DL_UnitDataIndication - etc.
  end provider
end LAPD_SERVICE
```

3.2. Transfer syntax

For messages conveyed via a local (as opposed to separate) association their internal representation is not an issue. This is not the case for separate associations. For example the *b* peer above might be a product of a different vendor. Simply using the exact bit representation for peer messages as specified in [6], however, is unsatisfactory since it tends to lower the abstraction level of the specification. In Kannel a special transfer syntax class can be employed to map abstract messages such as *info* into a protocol-conforming bit sequence. This notion is probably even more relevant in the higher protocol layers. The special transfer syntax used is specified in the **transfer** clause of a channel definition:

```
channel LAPD_APDUs is
  transfer LAPD_syntax;
  info: Information;
  sabme: SABME - etc.
end LAPD_APDUs

class Information key 1 is
  addr: address; - (SAPI, TEI) information
  poll: boolean; - P/F bit
  data: Packet; - user data
  send_packet,
  recv_packet: integer - sequence numbers
end Information
```

The LAPD_syntax class processes all messages that travel through a separate association of LAPD_APDUs. For example, when an *info* message is sent, the encoding process prepares the outgoing frame by inserting the address information, poll bit, user data (from layer 3) and the packet sequence numbers into correct positions in the frame and finally wraps it with the delimiter octets (01111110 binary). It also takes care of such low-level tasks as zero-bit insertion and FCS calculation. In the receiving end the LAPD_syntax decoder maps the incoming bit stream into corresponding abstract LAPD peer messages.

The process of data transfer for an *Information* message, such as *info* in LAPD_APDUs, is sketched below. Class *Information* is associated with the tag **(key)** 1 which identifies the specific messages for encoding/decoding. The LAPD_syntax class makes use of a buffer *buf* to store the raw binary representation of a message. A message is encoded using the procedure *encode* and decoded using the function *receive*. There are a number of predefined buffering operations available, e.g. *putbits* for filling the buffer upon sending and *getbits* for accessing the buffer upon receiving a message.

The transfer syntax class has access to the internal representation of message objects and also to dynamic type information. However it is cleanly separated from the rest of the specification. We also expect that support for generic transfer syntaxes will be part of the Kannel protocol support library. In the example Transfer_syntax is a library class providing general standard encoding and decoding services to LAPD_syntax. This is indicated in Kannel by specifying LAPD_syntax to be a *subtype* (subclass) of Transfer_syntax, as expressed with the "<" symbol.

```
class LAPD_syntax < Transfer_syntax is
  buf: Packet;
  k: integer;
    - etc.
  address_field is - encoding of address PDU
    - etc.
  end address_field;

  encode(object) is
    k:=arg.keyseq.first; - fetch PDU key
    if k=0 then
      - etc.
    elsif k=1 then - info: Information
      address_field; - arg.addr
      buf.putbits(1,0b0);
```

```
        buf.putbits(7,arg.send_packet);
        buf.putbits(1,arg.poll);
        buf.putbits(7,arg.recv_packet);
        buf.putbits(arg.length-6,arg.getbits(7,arg.length)); - arg.data
      elsif k=2 then ... - etc.
      end
    end endcode;

    receive(p:channel): object is
      if buf.getbits(25,25)=0b0 then - info: Information
        - decode the packet
      elsif ... - etc.; other possible PDUs
      end
    end receive;
  end LAPD_syntax
```

3.3. Dynamic processes

The ability of LAPD to support several simultaneous links calls for dynamic processes. This for the process declarations has been static, but the *Link* process depicted in Fig. 3 is different in that it must be dynamically created (note dashed borders). The responsibility for creating new *Link* instances is in the *routing methods* (RM for short) associated with the ports in the *Peer* process. The RM acts as packet redirector and creator for its direct child process, should they be dynamic. The textual equivalent of Fig. 3 looks following:

```
process Peer
ports
  up: LAPD_SERVICE(in user; out provider);
  peer: LAPD_APDUs
is
  dynamic process Link ... end Link;
  process Management ... end Management;

  mgmt_route(MGMT): Link is

  end mgmt_route;
  up_route(LAPD_SERVICE): Link is
```

```
    ...
  end up_route;
  peer_route(LAPD_APDUs): Link is
    ...
  end peer_route;
  const CMAX::=8
assoc m: Management; l: vector [Link, CMAX] is
  m.mgmt and l().mgmt in mgmt_route;
  up and l().up in up_route;
  peer and l().peer in peer_route
end Peer
```

Note first the keyword **dynamic** in front of the process declaration for *Link*: it specifies that *Link* instances must be dynamically created. In the association clause we specify that a maximum of *CMAX* links may be simultaneously active and that the *Link* instances are held in a generic *vector* class whose indices are used as local connection endpoint suffixes (CES) in connection management.

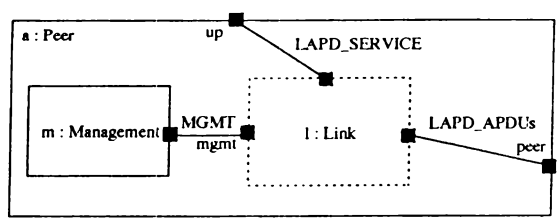

Fig.3. LAPD peer structure

The RMs are also given in the *assoc* clause. Their actual body is omitted for brevity; basically, each method checks for the special primitive(s) that require the creation of a new *Link* (such as the *estreq* primitive in the *up_route*) and maintain a mapping of (TEI[4], CES) values which enable the routing of packets to those *Link* instances already active.

There are restrictions to the use of dynamicity. Informally speaking, the use of dynamic processes is limited to local entities, that is a dynamic process may not have direct or indirect separate associations in its **assoc** clause. However this is not a severe limitation for protocol systems which Kannel is targeted at.

[4] Terminal Endpoint Identifier that identifies the connection.

The *Link* process consists of three subprocesses - two timers and the actual LAPD protocol machine - that form the leaf level of the specification (not shown).

3.4. Channel refinement

So far we have described the generic LAPD aspects: the service interface and the peer interface. But since the peer association is separate, we must somehow refine it to achieve a concrete implementation. In Kannel this refinement is separate from the "abstract" specification which helps in keeping it on a high level (without proper care the peer channel is easily neglected altogether resulting in a monolithic, low-level implementation which cannot be systematically reused).

Kannel has special mechanisms that allow the controlled refinement of a separate peer association into a process. These mechanisms are *process subtyping, implementation inheritance* and *statechart refinement.*

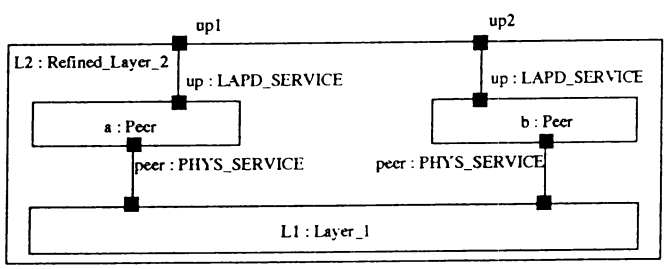

Fig.4. LAPD peer channel refinement

The visual form of the LAPD refinement is shown in Figure 4. It can be regarded as a more concrete implementation of Figure 2. (The graphical tool *ked* supports flexible transition between these two levels of abstraction.) Note that the old LAPD_APDUs channel has disappeared into the PHYS_SERVICE channel. The textual representation of the refinement gives more detail:

```
process Layer_1 is external "layer1.k";
process Refined_Layer_2 < Layer_2 is
    include Layer_2(Peer - > Old_Peer);
    process Peer is
      include Old_Peer(Link - > Old_Link);
      dynamic process Link is
        include Old_Link;
```

```
        refined
          ports
          peer: PHYS_SERVICE(in provider, out user);
          action ...
        end
      end Link
    end Peer
assoc L1: Layer_1 is
    L1.up1 and a.peer;
    L1.up2 and b.peer
end Refined_Layer_2
```

The process *Layer_1* describes a particular layer 1 implementation that the LAPD is being adapted to. Its implementation resides in a separate compilation unit as indicated by the **is external** *f* mechanism. Note that the external interface of the new *Refined_Layer_2* is identical to that of *Layer_2* which means it will accept exactly the same set of messages. This is specified by making *Refined_Layer_2* a subtype of *Layer_2*. Being a subtype it is potentially useful in RMs: one could imagine e.g. specifying the lower protocol (say, *P*) layer as dynamic and instantiating it with any subtype of *P* during execution.

Subtyping in Kannel is separated from reuse, and thus we need the `include` mechanism to textually include the *Layer_2* definitions into the refinement. During inclusion some included components are renamed in order to allow them to be used later. For example *Peer* is renamed as *Old_Peer* to allow it to be used in the redefinition of *Peer*. This renaming must be done for all paths leading to a leaf protocol that must be refined. Note that those parts not related to the peer channel being refined can be directly reused without any modification; this applies e.g. to the *Management* process (see Fig. 3).

The refinement of peer channel inherently changes the type of the process(es) associated with it. In LAPD, for example, the protocol data units such as *sabme* and *info* are carried *within* layer 1 service primitive user data. Thus the new type of the peer channel for processes *Link* and *Peer* will be PHYS_SERVICE. Unfortunately this also implies some changes are necessary in the protocol statechart[5]; in Kannel these changes are given in the **refined** clause in the redeclaration of any affected leaf process (in the case of LAPD, the *Link* process).

[5] Usually only the peeer channel communication must be refined with changes like *peer.send(sabme(..))* becoming *peer.send(phys_datareq(.., sabme(..),..))*.

A similar pattern of channel refinement may be repeated for *Layer_1* as well. Finally, however, layering reaches the bottom level (e.g. the physical layer of the OSI reference model), where the concrete communication takes place. If the communicating parties reside in the same address space there is actually no need for bitwise data transformations. Usually, however, the parties are physically separated into different machines. This holds for our LAPD example as well; recall that the *peer* channel was specified as **separate** (see Fig. 2). In such case the lowest protocol layer has to take care of the transmission over some physical medium. Kannel provides predefined low-level facilities for a physical communication. The low-level operations are largely machine-dependent and attach the externally relevant **separate** channels into some hard-wired facilities, such as Unix sockets. We omit the details in this paper.

4. Related work

Kannel provides a language-centered environment for protocol engineering. Thus, while the emphasis in the development of Kannel is on the language side, we do not underestimate the significance of the tool environment in practical protocol development. So far, we have concentreted on implementing an editor for graphical Kannel and a back-end translator for textual Kannel, but there are plans to integrate a number of useful support tools into the environment.

Kannel can be characterized as an object-orientated protocol engineering language. From this point of view the nearest relative of Kannel is SDL-92 (OSDL) [8, 12]. The main principal difference between the languages is that SDL-92 is an object-oriented compatible extension of an existing language (SDL), while Kannel has been designed with no explicit language as a model in the background. SDL-92 has introduced object-oriented facilities for protocol specification directly on the top of central features of SDL which, besides being a controversial approach of language design in general, inherits some of the obvious practical problems from SDL to SDL-92 as well. Most notably, the data types of SDL-92 are still founded on the algebraic specification language ACT ONE [7] with no special support for defining the encoding and decoding routines on (abstract) messages. Also the graphical state machine notation of SDL-92 is somewhat unconventional and clumsy. Kannel is more advanced in these aspects by providing dedicated ASN.1-like support for data transfer operations, and by basing the state machines on the hierarchic statechart formalism. Further development of SDL(-92) is currently addressing the former problem by integrating ASN.1 directly into the language.

With respect to object-orientated facilities, both Kannel and SDL-92 emphasize specialization and refinement as the key for reuse of protocol elements, mostly state machines and the involved channel machinery. In Kannel this is achieved mainly by implementation (code) inheritance and disciplined transformations, whereas SDL-92 employs virtual transitions and procedures and flexibly generic protocol units. It can be argued that the Kannel approach may be more explicit and thus closer to implementation issues, while the more general SDL-92 approach stresses design and specification aspects. One notable difference between the two languages is that SDL-92, being based on basic SDL with the ACT ONE flavor, does not provide "ordinary" data-orientated classes and objects at all, whereas these are the cornerstones of the type system in Kannel. Hence, it can be argued that Kannel is more object-orientated than SDL-92 with respect to conventional object-orientated programming. On the other hand, SDL(-92) is more graphical than Kannel due to having a visual representation even for algorithmic behavior ("procedures"). It has often been argued that it is not sensible to visually represent low-level operational details and that is why we have deliberately left them out from Kannel.

MONDEL [3] is another object-oriented language for specifying communication protocols. The idea behind MONDEL is roughly the same as that behind Kannel: to integrate protocol conventions and description techniques into a single object-orientated methodology and language. MONDEL is compatible with several standard OSI notations, most notably with ASN.1. The main difference to Kannel is that MONDEL as a formal language is mainly directed to validation and verification, whereas Kannel is more implementation oriented. Thus the most natural operational environment of MONDEL is simulation instead of a full-fledged execution.

The Kannel environment under development shares the same central tools as a number of protocol engineering environments based on SDL, Estelle or LOTOS. One notable difference is that the conventional environments are usually not seamlessly based on a single language, but include at least an ASN.1 tool (or similar) as an external entity. One systematically designed multi-language and multi-tool environment is described in [15]. The environment has a layered architecture consisting of a low-level kernel, general support shells and the necessary application tools. Being founded on a standard kernel and a consistent internal protocol information the architecture provides an extensible environment and flexible interworking for all the tools. Typically the tools are based on different protocol languages and, therefore, the approach to integration is quite different from the Kannel approach where language is the unifying factor.

5. Conclusions

Protocol engineering is a versatile discipline with a number of important aspects to take care of. We have presented Kannel, an application-oriented language designed especially for engineering and implementing communication protocols. While the principle behind the most conventional languages in the area has been providing explicit support for solving some narrow problems within protocol engineering, Kannel aims to cover the whole discipline in a uniform manner under a single language and environment. The most notable features of Kannel are object-oriented development and reuse of protocol elements, a visual syntax especially for Harel's statecharts, strong distribution and a mechanism for data transfer.

The main components of the Kannel environment, a graphical editor and a compiler to C++ are currently under implementation. A third Kannel tool currently under development is a graphical class browser. The first prototype of the system will be available by the end of 1995. The environment will be gradually complemented with support tools that are useful in protocol engineering, e.g. a simulator and a translator between Kannel and ASN.1.

Acknowledgements. The Kannel project is financed by the Technology Development Centre of Finland (TEKES) and by an industrial steering group. The industrial steering group has also greatly affected the design of the language. The working environment has been provided by Nokia Research Center.

References

[1] **America P.,** Issues in the design of a parallel object-orientated language, *Formal Aspects of Computing,* **1** (1989), 366-411.

[2] **Belina F. and Hogrefe D.,** The CCITT specification and description language SDL, *Computer Networks and ISDN Systems,* **16** (1989), 311-341.

[3] **v.Bochmann G., Poirier S. and Mondain-Monval P.,** Object-orientated design for distributed systems: The OSI directory example, *Computer Networks and ISDN Systems,* **27** (1995), 571-590.

[4] **Bolognesi T. and Brinksma E.,** Introduction to the ISO specification language LOTOS, *Computer Networks and ISDN Systems,* **14** (1987), 25-59.

[5] **Budkowski S. and Dembinski P.,** An introduction to ESTELLE: A specification language for distributed systems, *Computer Networks and ISDN Systems,* **14** (1987), 3-23.

[6] *Digital Subscriber Signalling System No. 1 (DSS 1), Data Link Layer,* Recommendations Q.920-Q.921, CCITT, 1992.

[7] **Ehrig H. and Mahr B.,** *Fundamentals of algebraic specification 1.,* Springer, 1985.

[8] **Faergemand O. and Olsen A.,** Introduction to SDL-92, *Computer Networks and ISDN Systems,* **26** (1994), 1143-1167.

[9] **Granö K., Harju J., Paakki J. and Järvinen T.,** *Proposal for a protocol engineering language,* Technical Reports TR-6, Department of Computer Science and Information Systems, University of Jyväskylä, 1994.

[10] **Harel D.,** Statecharts: A visual approach to complex systems, *Science of Computer Programming,* **8** (1987), 231-274.

[11] **Harel D.,** On visual formalisms, *Communications of the ACM,* **31** (1988), 514-530.

[12] **Møller-Pedersen B.,** Rationale on object-oriented SDL, *Object-oriented environments: The Mjølner approach,* eds. J.L.Knudsen, M.Löfgren, O.Lehrmann-Madsen and B.Magnusson, Prentice-Hall, 1994, 136-152.

[13] **Omohundro S.,** *The Sather 1.0 Specification,* unpublished, available electronically as file `manual-1.0v5.ps.Z` in directory `/pub/sather` via anonymous ftp to `ftp.icsi.berkeley.edu`.

[14] **Steedman D.,** *ASN.1 - Tutorial & Reference,* Technology Appraisals Ltd., 1990.

[15] **Schneider J.M., Mackert L.F., Zörntlein G., Velthuys R.J. and Bär U.,** An integrated environment for developing communication protocols, *Computer Networks and ISDN Systems,* **25** (1992), 43-61.

K. Granö
Department of Computer Science
and Information Systems
University of Jyväskylä
P.O.B. 35
FIN-40351 Jyväskylä, Finland
grano@research.nokia.com

J. Harju
Department of
Information Technology
University of Technology
P.O.B. 20
FIN-53850 Lappeenranta, Finland
harju@lut.fi

T. Järvinen
Department of Computer Science
and Information Systems
University of Jyväskylä
P.O.B. 35
FIN-40351 Jyväskylä, Finland
jarvinet@research.nokia.com

T. Larikka
Department of
Information Technology
University of Technology
P.O.B. 20
FIN-53850 Lappeenranta, Finland
larikka@lut.fi

J. Paakki
Department of Computer Science
University of Helsinki
P.O.B. 26
FIN-00014 Helsinki, Finland
paakki@cs.helsinki.fi