# T U M

I N S T I T U T F Ü R I N F O R M A T I K


# From Peer to Service - Object-Oriented Protocol Refinement in Kannel


Kari Grano, Jukka Paakki



TUM-I9623

Mai 96

T E C H N I S C H E U N I V E R S I T Ä T M Ü N C H E N

TUM-INFO-05-96-I9623-200/1.-FI
Alle Rechte vorbehalten
Nachdruck auch auszugsweise verboten

©96 FAKULTÄT FÜR MATHEMATIK UND
INSTITUT FÜR INFORMATIK
TECHNISCHE UNIVERSITÄT MÜNCHEN

Druck: Fakultät für Mathematik und
Institut für Informatik der
Technischen Universität München

# From Peer to Service – Object-Oriented Protocol Refinement in Kannel *

Kari Granö, Jukka Paakki
Department of Computer Science
P.O.Box 26, FIN–00014 University of Helsinki, Finland

**Abstract**

The refinement of communication in protocol engineering is studied by analyzing the relation between a peer-to-peer communication scheme and its service-level counterpart, a characteristic that is well-known in practice but rarely studied in detail. It is shown how an abstract protocol can be developed towards a concrete implementation by gradually refining the abstract messages and the involved state machines, moving systematically from layer to layer over the subject application. The characteristics of the refinement are formalized, and an example is given showing how the method can be applied in practical protocol development. The object-oriented language Kannel is introduced as an advanced tool for protocol engineering providing special support for the refinement technique.

## 1 Introduction

*Protocol engineering* is a versatile discipline with the emphasis on systematically developing distributed communications software of high quality. Having reached a relatively mature status, the protocol engineering field is supported by several special development environments and description languages, the most well-known ones being SDL, Estelle, LOTOS, and ASN.1. The description language employed in this paper is *Kannel* [GHP94] which is based an on object-oriented, visual, and state-based view on protocol engineering. Kannel provides application-oriented support for a number of central aspects in protocol engineering, e.g., for the refinement mechanism which is the topic of this paper.

Protocol engineering is usually founded upon a modularized software architecture. That is, the communicating parties are organized as a stack of *layers*, each having its own special task in the application. The most well-known example is the standardized OSI reference model of seven layers, but similar (though usually more economical) architectures are quite common also in networking and telecommunication applications not strictly following the OSI model.

As in software engineering in general, the hierarchical protocol layers are connected to each others via their *interfaces*. An interface captures the services a layer is externally providing to other layers, making it thus possible to integrate together components that are logically independent in their internal behavior. In protocol engineering, there typically appear two kind of interfaces: (1) between the *peers* of the communicating systems, and (2) between the neighbouring layers within one system. These two basic interface classes serve different purposes: a peer-to-peer interface is needed for specifying the logical communication protocol between the end systems, whereas a layer($n$)-to-layer($n-1$) interface makes the *services* of layer $n-1$ available to its upper layer $n$ for implementing its peer-to-peer protocol.

A communication *protocol* defines how messages are exchanged between two entities through a common interface. Notice that there is always a protocol both between two (distributed) peers

*Part of this work has been carried out while visiting the Department of Computer Science, University of Munich, Germany. The research is part of the projects *Integrated protocol engineering* (Academy of Finland), *The semantics of the protocol engineering language Kannel* (Academy of Finland), and *Object-oriented programming and compiler construction* (Academy of Finland and Deutscher Akademischer Austauschdienst).

and between two adjacent (centralized) layers. There is a close conceptual coupling between these two: A layer($n$)-to-layer($n-1$) protocol can be regarded as an implementation of the peer($n$)-to-peer($n$) communication scheme between the end systems. From the protocol-software engineering point of view, this coupling is most valuable when moving from protocol design into protocol implementation. A peer-to-peer protocol specifies abstractly how the communication between the systems shall behave in general, whereas the corresponding service-level protocol defines concretely how the communication is actually realized. Hence, the step from design into implementation can be taken in a systematic fashion by concentrating on the mapping from peer-level protocols (and interfaces) to the corresponding service-level protocols (and interfaces).

In this paper we present a constructive technique of systematically moving from peer-to-peer communication into layer-to-layer communication. Our approach is based on *refining* the abstract peer($n$)-to-peer($n$) communication protocol by replacing it with the corresponding more concrete layer($n$)-to-layer($n-1$) service protocols and with the (perhaps still abstract) peer($n-1$)-to-peer($n-1$) protocol. A similar scheme of protocol refinement is gradually repeated at the lower layers $n-1$, $n-2$, ..., until a proper level of precision has been reached. The technique is supported by the protocol engineering language Kannel that provides dedicated facilities for protocol implementation in terms of object-oriented inheritance and refinement of communication patterns.

We proceed as follows. The characteristics of protocol refinement are presented in Section 2, followed by a formal study in Section 3. In Section 4 a constructive approach to protocol refinement is presented, using Kannel as the demonstrational case language. Finally, conclusions are drawn in Section 5.

## 2 The notion of protocol refinement

Let us study a typical abstract communication scheme illustrated in Figure 1. Here $A$ and $B$ are (probably distributed) entities (usually processes) that communicate according to their common protocol $AB$ to provide the required services to their clients, *User_A* and *User_B*.

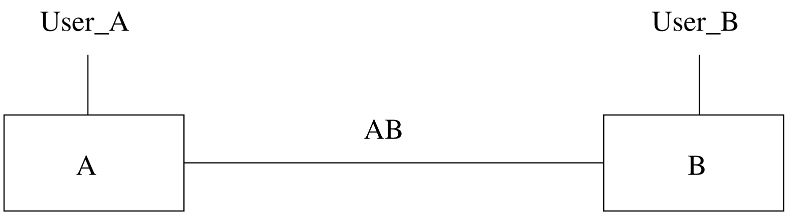


Figure 1: Abstract communication.

In simple applications the communication might be carried out in this straightforward manner, but in realistic cases the functionality of $A$ and $B$ is so complex that some form of modularity is needed, as illustrated in the refined scheme of Figure 2. Now the protocol $AB$ is realized by making use of entities $C$ and $D$ residing at a "lower layer" of functionality. That is, the $AB$ protocol is implemented by using the services of $C$ and $D$ following the protocol $CD$ of their own. From the viewpoint of message-flow, a message from *User_A* to *User_B* does not go directly via $A$ and $B$ as the abstract scheme in Figure 1 suggests, but indirectly along the path $A'$ – $C$ – $D$ – $B'$. Still, from the client's point of view, the external functionality of the system is the same irrespective of the system's architecture (Figure 1 or Figure 2).

Notice that the scheme in Figure 2 is more detailed than that in Figure 1: (1) The peer-to-peer protocol $AB$ (at layer $n$) has been replaced by three new protocols, the service-level protocols $A'C$ and $B'D$ (between layers $n$ and $n-1$) and the peer-to-peer protocol $CD$ (at layer $n-1$); (2) the entities $C$ and $D$ have been introduced (at layer $n-1$); and (3) the entities $A$ and $B$ (at layer $n$) have been replaced by $A'$ and $B'$, respectively, since their direct mutual communication

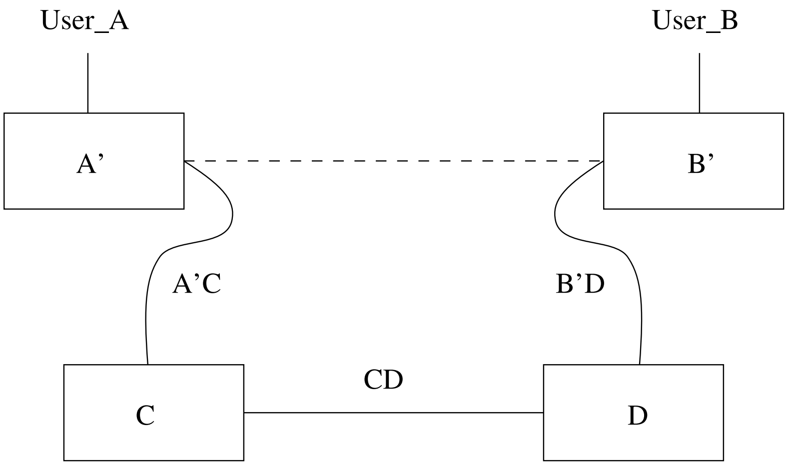


Figure 2: Communication by refinement of $AB$.

scheme has been changed into an indirect one. Thus we can regard the scheme in Figure 2 as an implementation or a *refinement* of that in Figure 1.

The term "refinement" suggests that the two communication schemes shall be semantically related: (1) The external behavior with respect to *User_A* and *User_B* shall remain the same; and (2) the abstract peer-to-peer protocol $AB$ shall be retained, that is, the refinement shall still follow the communication rules captured in protocol $AB$. Intuitively, this means that even under the refinement, (1) *User_A* and *User_B* shall be able to exchange the same set of message sequences as originally; and (2) for each message sequence transmitted between $A$ and $B$, there must be a corresponding (refined) exchange between $A'$ and $B'$. Notice, however, that in addition to the messages covered by the abstract protocol $AB$, $A'$ and $B'$ typically process a number of implementation-oriented messages, at least those captured in protocols $A'C$ and $B'D$ as services provided by the entities $C$ and $D$.

Since the scheme in Figure 2 still applies a peer-to-peer protocol, $CD$, the refinement process can be continued. By iteratively following the same principle, an abstract communication scheme can be developed into a suitable layered architecture and a proper level of precision. The refinement process typically continues until a layer providing direct physical communication services has been encountered (for instance the physical layer in the OSI model).

From a conceptual point of view, the division of protocols into "peer-to-peer" and "layer-to-layer" ones (e.g., $CD$ vs. $A'C$ in Figure 2) is rather arbitrary; after all, in both cases the objective is to specify a communication process between two entities.[1] Therefore, we may in principle refine *any* protocol, not only a "peer-to-peer" one. Moreover, the refinement may introduce more than two new lower-layer processes which is the case in Figure 2 ($C$ and $D$). For instance, the architecture of Figure 2 could be further developed by refining the "vertical" protocol $A'C$ into a chain of, say, four new processes.

## 3 Formal properties of protocol refinement

Let us consider the refining step from Figure 1 to Figure 2 in more detail. What kind of techniques are possible to revise the abstract communication scheme of Figure 1 into the more concrete setting of Figure 2 and still retain the behavior externally the same? At least the following solutions are possible:

[1] Of course, in practice the division may be significant.

1. The whole system is rewritten by the protocol designer, by informally making sure that the refined scheme does not introduce any communication mismatches with respect to the clients *User_A* and *User_B*.

2. A formal *program refinement* strategy is applied, as described e.g. in [Jon80] or in [Bac88]. In this case the refinement step from Figure 1 to Figure 2 is formally proven correct by showing that the latter scheme preserves the behavior and total correctness of the former one. Since the external behavior of a distributed system is usually defined as the set of *traces*, i.e. the set of externally observable message sequences, the special refinement theory of state-based data types [Nip89] can be conveniently applied.

3. A *protocol conversion* methodology can be used. In protocol conversion [Gre86], two different protocols (usually specified with communicating state machines) are merged together by automatically producing a special intermediate machine that "translates" the messages sent by the machine of one protocol into messages accepted by the machine of the other protocol; in other words, the converter makes the two protocols able to interoperate. For instance, when moving in our example from Figure 1 to Figure 2, this technique would introduce a converter both at the connection $A'C$ and at the connection $B'D$. For a more detailed description of different protocol conversion techniques, refer, e.g., to [CaL89] and [PeL93].

4. One can apply the general concept of *software / interface adaptors* [YeS94] to glue together protocols at different layers of the communication architecture. This strategy is rather close to protocol conversion, especially if the adaptation is based on communicating state machines for the specification of software protocols. The main difference between these two is that the software adaptation technique is more general than protocol conversion, due to accepting the distribution of parameters into several different services, and to being based on advanced software engineering principles such as object-orientation (e.g., [Tha94]).

The first alternative is hopeless in nontrivial cases since the matching has to be done totally by human means without any formal support, which sooner or later inevitably leads to a behavioral contradiction between the abstract and the concrete communication scheme. The formal stepwise refinement strategy is rather laborous since then the protocol designer has to (a) formally specify the abstract scheme, (b) formally specify the concrete scheme, and (c) formally prove that the latter correctly implements the former. Also the state-of-the-art in formal program derivation and verification is still too immature for being usable in practical applications. Finally, the related techniques of protocol conversion and software adaptation introduce additional components to the communication architecture, typically one converter/adaptor for each pair of integrated processes. This would soon lead to an exhaustive number of components when stepwise devising a complete layered implementation for a complex communications protocol. Moreover, these techniques need some additional information for generating the converter/adaptor, such as a service specification over the client protocols or a description of the synchronized behavior of the integrated components.

Due to such shortcomings with the conventional solutions addressed above, our approach to protocol refinement is based on *object-oriented* techniques: *incremental modification*, *subtyping*, and *inheritance* (more precisely: *code reuse*). The central idea is to avoid the introduction of new system components upon protocol refinement, and instead reuse as much of the existing framework as possible. With regard to our example transition from the abstract scheme in Figure 1 to the more concrete one in Figure 2, this means the following:

- The process $A'$ is a modification of the process $A$. This means that the program code written for $A$ is reused when producing the code for $A'$. This applies most notably to the (communicating) state machine for $A$ that is incrementally modified to cope with the new protocol scheme. Likewise, the process $B'$ is a modification of the process $B$.

- The structured layer architecture in Figure 2 and the flat layer in Figure 1 (excluding in both cases the external clients *User_A* and *User_B*) are subtypes of the same virtual layer and hence compatible.

- The message flow along the path $A' \to C \to D \to B'$ following the protocols $A'C$, $CD$, and $B'D$ in Figure 2 is a refinement of the flow along the path $A \to B$ by the protocol $AB$ in Figure 1. Likewise, the message flow along the path $B' \to D \to C \to A'$ in Figure 2 is a refinement of the flow along the path $B \to A$ in Figure 1.

The relation of inheritance and refinement has been analyzed, e.g., in [Cus91]. The notion of refinement is more formal than the notion of inheritance: For a component $C$ to be a refinement of a component $D$, $C$ must guarantee the same (correct) behavior whenever substituted for $D$. In [WeZ88] such a property of an incremental modification mechanism is called *behavioral compatibility* which is usually not guaranteed by inheritance in object-oriented languages. This is also the case in Kannel where inheritance satisfies just a weaker property, *signature compatibility* [WeZ88]. Therefore, (protocol) refinement in Kannel does not mean the same as (protocol) inheritance but has stricter behavioral properties, as formalized below. However, these two are still related in Kannel in the sense that the flexible inheritance mechanism is applied for *expressing* and *implementing* the disciplined refinement mechanism: Entities subject to refinement are typically in a subtype relation and/or may share pieces of code. The Kannel approach to refinement is described in more detail in Section 4.

**Definition 1.** We assume conventionally that the communication between processes is specified as communicating finite state automata (machines), one for each process. A *communicating finite state automaton* $\mathcal{A}$ is a 5-tuple: $\mathcal{A} = (S, s_0, F, M, \delta)$, where $S$ is a finite set of *states*, $s_0 (\in S)$ is the *initial state*, $F (\subseteq S)$ is the set of *final states*, $M$ is a finite set of *messages*, and $\delta$ is the *transition function*: $\delta : S \times M \to S$. The message set $M$ is divided into two subsets: $M = M^i \cup M^o$, where $M^i$ denotes the set of *input* messages (events) and $M^o$ denotes the set of *output* messages. $\mathcal{A}_P$ denotes the finite state automaton associated with the process $P$. $M_P$, $M_P^i$, and $M_P^o$ denote respectively the message set, the input message set, and the output message set of the automaton associated with the process $P$.

Since state automata are expressed in Kannel as statecharts [Har87], a transition from a state $S$ to a state $T$ can be associated with an input message $i \in M^i$ and a sequence of output messages $o_1, o_2, \ldots, o_n; \forall i \in [1, n] : o_i \in M^o$, standing for the reception of an incoming event and the immediate sending of the outgoing messages: $\delta(S, (io_1 o_2 \cdots o_n)) = T$. Such a situation is interpreted as introducing intermediate states $S_i$ for splitting the multi-message transition into singletons: $\delta(S, i) = S_1, \delta(S_1, o_1) = S_2, \ldots, \delta(S_n, o_n) = T$.

**Definition 2.** Let $P$ and $Q$ be processes associated with a (communicating) finite state automaton. Then $M_{PQ}^o$ denotes the set of *messages from $P$ to $Q$*, and $M_{PQ}^i$ the set of *messages to $P$ from $Q$*. These define the communicated message set between $P$ and $Q$, that is, the set of output messages of the automaton for $P$ ($Q$) that are also input messages of the automaton for $Q$ ($P$): $M_{PQ}^o = M_{QP}^i = M_P^o \cap M_Q^i$. The total communication between $P$ and $Q$ is denoted by $M_{PQ}^{io} = M_{QP}^{io} = M_{PQ}^o \cup M_{PQ}^i$. To be able to communicate in both directions, $P$ and $Q$ must have the sets $M_{PQ}^o$ and $M_{PQ}^i$ ($M_{QP}^i$ and $M_{QP}^o$) nonempty. For notational simplicity, we assume that the communicated message set is different for each different pair of target processes: $M_{PQ}^{io} \cap M_{PR}^{io} = \emptyset$ whenever $Q \neq R$.

Consider the example scheme in Figure 1. The communication between the processes $A$ and $B$ is specified by two finite state automata (statecharts), $\mathcal{A}_A$ for $A$ and $\mathcal{A}_B$ for $B$. Since the process $A$ is communicating with its client *User_A* (denoted U) and with its peer $B$, the set of messages $A$ is processing is divided into the following subsets: $M_A = (M_A^i \cup M_A^o) = (M_{AU}^{io} \cup M_{AB}^{io}) = (M_{AU}^o \cup M_{AU}^i \cup M_{AB}^o \cup M_{AB}^i)$. Likewise, $M_B = (M_B^i \cup M_B^o) = (M_{BV}^{io} \cup M_{BA}^{io}) = (M_{BV}^o \cup M_{BV}^i \cup M_{BA}^o \cup M_{BA}^i)$, where $V$ denotes *User_B*. Furthermore, $M_{AB}^o = M_{BA}^i$, and $M_{BA}^o = M_{AB}^i$.

As usual, we model the behavior of a distributed system as sequences of messages, or traces, between the processes in the system (see e.g. [Jon89]). Since the functionality of communicating processes is defined in our approach as finite state automata, we can apply the standard concepts and techniques of automata theory for specifying and analyzing the trace-behavior of protocols.

**Definition 3.** Let $\mathcal{A} = (S, s_0, F, M, \delta)$ be a communicating finite state automaton. A *trace* in $\mathcal{A}$, denoted $\mathcal{T}_{\mathcal{A}}$, is a sequence of (input or output) messages associated with a path from the initial state of $\mathcal{A}$ to a final state of $\mathcal{A}$. That is, a message sequence $m_1 m_2 \cdots m_n$ $(\forall i \in [1, n] : m_i \in M)$ is a trace, if $\delta(s_0, m_1) = s_1, \delta(s_1, m_2) = s_2, \ldots, \delta(s_{n-1}, m_n) = s_n$ such that $s_n \in F$. The *language* of $\mathcal{A}$, denoted $\mathcal{L}_{\mathcal{A}}$, is the set of traces in $\mathcal{A}$.[2] If the automaton $\mathcal{A}$ is associated with the process $P$, we denote by $\mathcal{T}_P$ a *trace* in $P$ and by $\mathcal{L}_P$ the *language* of $P$.

**Definition 4.** Let $\mathcal{A} = (S, s_0, F, M, \delta)$ be a communicating finite state automaton, and let $\mathcal{T}_{\mathcal{A}} = m_1 m_2 \cdots m_n$ be a trace in $\mathcal{A}$. Let $N \subseteq M$ be a set of messages. The *projection* of $\mathcal{T}_{\mathcal{A}}$ with respect to $N$, denoted $\mathcal{T}_A/N$, is a subsequence of $\mathcal{T}_{\mathcal{A}}$ consisting of just the messages in $N$. That is, $m_i (i \in [1, n])$ is in $\mathcal{T}_A/N$ only if $m_i \in N$.

The refinement of a protocol between processes $P$ and $Q$ is achieved by refining the messages exchanged between $P$ and $Q$ into more concrete ones, by introducing a new protocol layer to implement the abstract protocol, and by modifying the communicating automata associated with $P$ and $Q$ to cope with the new architecture.

To specify this, we give the necessary definitions below. Intuitively, mapping functions are needed for translation between an abstract message and a more concrete one (usually achieved in practice by composing a protocol data unit from a service data unit and embedded local control information, and by decomposing it later on), for splitting a concrete message into several abstract ones (in practice by segmenting a service data unit into a set of protocol data units), and for joining several messages into a single one (in practice by concatenating several protocol data units into a single service data unit).

**Definition 5.** Let $P$, $Q$, and $R$ be processes, such that there is a communication protocol between (the state automata for) $P$ and $Q$ and between (the state automata for) $R$ and $Q$. $P$ and $R$ are *trace-equivalent* with respect to $Q$, if the following conditions hold: (1) $M^o_{PQ} = M^o_{RQ}$; (2) $M^i_{PQ} = M^i_{RQ}$; (3) $\{\mathcal{T}_P/M^{io}_{PQ} \mid \mathcal{T}_P \in \mathcal{L}_P\} = \{\mathcal{T}_R/M^{io}_{RQ} \mid \mathcal{T}_R \in \mathcal{L}_R\}$.

This definition stands for the fact that the refinement of one communication protocol shall not affect the system's behavior with respect to the other protocols. When considering the situation in Figures 1 and 2, the processes $A$ and $A'$ must be trace-equivalent with respect to *User_A*, and the processes $B$ and $B'$ must be trace-equivalent with respect to *User_B*.

**Definition 6.** Let $P_1, P_2, \ldots, P_n, n \geq 2$, be processes such that $P_i$ communicates directly with $P_{i+1}$ $(i = 1, 2, \ldots, n-1)$ via state automata. Then $(P_1 P_2 \cdots P_n)$ is called a *configuration*.

For instance, (*User_A* $A$ $B$ *User_B*) and $(AB)$ are configurations in Figure 1, and (*User_A* $A'$ $C$ $D$ $B'$ *User_B*) and $(A'CDB')$ are configurations in Figure 2.

**Definition 7.** Let $\mathcal{T}_P = m_1 m_2 \cdots m_n$ be a trace, let $f : M \rightarrow N$ be a function where $M$ and $N$ are sets of messages, and let $S \subseteq M$. Then the *transformation* of $\mathcal{T}_P$ by $f$ and $S$, denoted $t(\mathcal{T}_P, f, S)$, is the trace $p_1 p_2 \cdots p_n$ where $p_i = f(m_i)$ if $m_i \in S$, and $p_i = m_i$ otherwise; $i = 1, 2, \ldots, n$.

**Definition 8.** (See Figures 1 and 2). Let $(AB)$ be a configuration, and let $M$ be a message set. Let $\mathcal{S}(M)$ denote the set of all the sequences of messages in $M$. Let $Com(P)$ denote the set of processes with which process $P$ has a communication protocol. The configuration $(A'CDB')$ is a *refinement* of $(AB)$ if there exist *mapping functions* $f_1$ (total), $f_2$, $f_3$, $f_4$, $g_1$ (total), $g_2$, $g_3$, and $g_4$:[3]

[2] This corresponds to the concept of language in automata theory. Therefore it is necessary for our automata to have final (accepting) states. Notice that while reactive systems usually do not have a fixed final state from where no progress is possible, even they always have "logical" final states closing a main event loop. Typically a reactive automaton contains a cycle with the initial state as entry; in that case the initial state must be regarded as a final state as well.

[3] For more extensive configurations, the definition is similar but involves a larger number of mapping functions.

$f_1 : M^o_{AB} \rightarrow M^o_{A'C}$ $(= M^i_{CA'})$
$f_2 : M^i_{CA'} \rightarrow \mathcal{S}(M^o_{CD})$ $(= \mathcal{S}(M^i_{DC}))$
$f_3 : \mathcal{S}(M^i_{DC}) \rightarrow M^o_{DB'}$ $(= M^i_{B'D})$
$f_4 : M^i_{B'D} \rightarrow M^i_{BA}$ $(= M^o_{AB})$
$g_1 : M^o_{BA} \rightarrow M^o_{B'D}$ $(= M^i_{DB'})$
$g_2 : M^i_{DB'} \rightarrow \mathcal{S}(M^o_{DC})$ $(= \mathcal{S}(M^i_{CD}))$
$g_3 : \mathcal{S}(M^i_{CD}) \rightarrow M^o_{CA'}$ $(= M^i_{A'C})$
$g_4 : M^i_{A'C} \rightarrow M^i_{AB}$ $(= M^o_{BA})$

such that the following conditions hold:

1. $\forall m \in M^o_{AB} : m = f_4(f_3(f_2(f_1(m))))$
2. $\forall n \in M^o_{BA} : n = g_4(g_3(g_2(g_1(n))))$
3. $Com(A') \setminus \{C\} = Com(A) \setminus \{B\}$, and $A'$ and $A$ are trace-equivalent with respect to every process $P \in Com(A) \setminus \{B\}$
4. $Com(B') \setminus \{D\} = Com(B) \setminus \{A\}$, and $B'$ and $B$ are trace-equivalent with respect to every process $P \in Com(B) \setminus \{A\}$
5. $\{\mathcal{T}_{A'}/M^{io}_{A'C}\} = \{t(T, g_3 \circ g_2 \circ g_1, M^i_{AB}) \mid T = t(\mathcal{T}_A/M^{io}_{AB}, f_1, M^o_{AB})\}$
6. $\{\mathcal{T}_{B'}/M^{io}_{B'D}\} = \{t(T, f_3 \circ f_2 \circ f_1, M^i_{BA}) \mid T = t(\mathcal{T}_B/M^{io}_{BA}, g_1, M^o_{BA})\}$

The definition of refinement captures the fact that the abstract communication pattern shall remain the same, even when the concrete message path changes. Conditions 1 and 2 above guarantee that each message in the original protocol and process configuration reaches its destination in the corresponding format in the refined protocol/configuration even when having different intermediate representations during the transmission. Conditions 3 and 4 state that the refinement of a protocol shall not affect the external client-wise behavior of the end processes. Finally, conditions 5 and 6 guarantee that the state automata of the refined processes ($A'$ and $B'$ in Figure 2) manifest the same abstract communication protocol with respect to the original processes ($A$ and $B$ in Figure 1) even when introducing concrete representations for the messages as well as additional communication with the new lower-layer processes ($C$ and $D$ in Figure 2). In other words, the extended communication must preserve the original traces by the following mapping of messages:

- message $m$ sent from $\mathcal{A}_A$ to $\mathcal{A}_B$ is represented as $f_1(m)$ in $\mathcal{A}_{A'}$;
- message $n$ received from $\mathcal{A}_B$ in $\mathcal{A}_A$ is represented as $g_3(g_2(g_1(n)))$ in $\mathcal{A}_{A'}$;
- message $o$ sent from $\mathcal{A}_B$ to $\mathcal{A}_A$ is represented as $g_1(o)$ in $\mathcal{A}_{B'}$;
- message $p$ received from $\mathcal{A}_A$ in $\mathcal{A}_B$ is represented as $f_3(f_2(f_1(p)))$ in $\mathcal{A}_{B'}$.

Finding the mapping functions $f_i$ and $g_i$ is, of course, very hard in a general case without any discipline on the structure of the refinement. That is why the refinement mechanism in Kannel involves certain syntactic and semantic restrictions to make it possible to *automatically* find the mapping functions and to verify the refinement conditions, as illustrated in the next section.

# 4 Object-oriented protocol refinement in Kannel

The refinement concept is realized in Kannel with mechanisms for grouping and subtyping processes, combined with constructs for layer refinement and event mapping within state machines. The latter two mechanisms aim at maximal reuse of existing code. Subtyping and code reuse are normally distinct mechanisms in Kannel but refinement combines them, yielding a construct that resembles the traditional concept of inheritance. We illustrate these mechanisms with a Kannel model of a generic weather reporting system and its disciplined refinement over an alternating bit transport service.

**Overview of the weather reporting system** The weather system consists of a set of sensors connected to a control terminal, which provides information about mean temperature changes and fault conditions to the user of the system. The control terminal receives temperature reports from the sensors and computes their average. Should the average change too radically, the terminal will inform the user of the system about it. Likewise, fault indications from the sensors are reported. In addition, the user may pose explicit report requests which are immediately answered. Each sensor contains a computing unit and a timer which is used to control the interval at which the sensor sends probes. The control terminal and the sensors are abstracted into a `WeatherSystem` interface process (see below) which is used as the basis for subtyping. Full details of the example are included in the appendix.

**Brief introduction to Kannel** A Kannel program consists of a set of communicating objects. Kannel divides these objects into two categories—there are *local* objects that can exploit identity and create other objects, something that *distributable* objects cannot do. The distributable Kannel objects are called *processes*; they are used to model the combined state and behavior of a protocol. The communication and creation capabilities of processes are provided with mechanisms such as channels, ports, messages, routers, transfer syntaxes, and statecharts.

Distributable objects are superior to local objects. They are used to specify the system on a larger level of granularity, whereas the local objects serve to provide traditional computing capabilities. Recognizing this, Kannel provides a visual syntax for the distributable parts of the language. The visual syntax describes *both* the structure of processes and the multiplicity of their instances; hence it aims at being simultaneously a static and a dynamic model of the system. This is in contrast with approaches that use separate models for these aspects.

The intention of Kannel is to describe the communicating system as a whole by including in the specification all the involved components. This deviates from the traditional approach of describing only a single communicating (although maybe layered) entity at a time. Indeed, a Kannel compilation can result in the creation of several such entities. The designer controls this separation by tagging some associations between processes as `separate`.

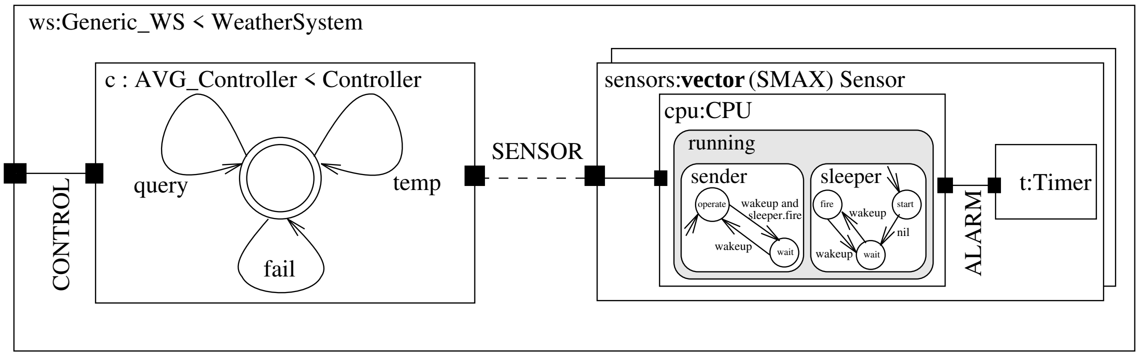


Figure 3: The `Generic_WS` process.

## 4.1 A high-level specification of `WeatherSystem`

**Channels and process interfaces** On the top level, the weather system consists of processes `User` and `WeatherSystem` that are connected by the channel `CONTROL`. The processes communicate by sending *messages*, that is, local objects whose types are specified within the channel. Kannel channels may be segmented into sets of unidirectional messages with *views* as shown in the definition of `CONTROL` below:

**channel** CONTROL **is**

```
  view Requests is query : event end Requests;
  view Results is
      report : real;
      faulty : integer
  end Results
end CONTROL
```

For example, message `query` always travels from the user to the weather system, never vice versa (the flow directions for the views are stated within process type definitions). Thus, a channel defines one service interface for a process. The combined set of these interfaces combined with a protocol assertion forms the type of a (branch) process:

```
process interface WeatherSystem is
  service : CONTROL(in Requests, out Results);
  protocol SENSOR separate
end
```

Local classes in Kannel use *interfaces* (collections of method signatures) as the basis for subtyping and for the dynamic binding of method calls. In a similar fashion, processes use *process interfaces* to specify their externally visible properties. The interface for `WeatherSystem` states that it is prepared to process any incoming message within the `Requests` view and that it may generate any message within the `Results` view[4]. However, the specification of the legal temporal orderings of these messages is left unspecified: rather, they are specified with a statechart within the concrete (leaf) processes that are subtypes of `WeatherSystem`.

Kannel requires that each subtype exhibits the equivalent set of traces on its service interfaces. There are two reasons for dropping the state automata from the interface specification: First, there may exist several structurally differing automata that exhibit equivalent behavior; and second, the automaton generally has to access internal details of the process in order to handle events and these details should not belong to the interface. Instead of an automaton, the interface may specify a protocol assertion (see below) that abstracts the kind of service it internally provides. The assertions are also crucial for process refinement, as we shall see: any (protocol) association to be refined must be specified with a protocol assertion.

The process structure in Kannel is given statically, and since instance identities and creation are not applicable to them (due to the strong distribution semantics that Kannel imposes on processes) the idea of process subtyping with interfaces may seem unnecessary. However, during system initialization and in the restricted context of *routers* the subtyping may be exploited, opening interesting possibilities such as the dynamic selection of process stacks.

**Grouping of processes** Kannel processes fall into two categories: *leaf processes* are used to express the actual behavior of a protocol element. They contain a statechart that declares the legal temporal orderings of events together with (private) methods and attributes that are used in computation. It is worth pointing out that a leaf process in itself does not implement a protocol—a protocol is a *mutual* agreement between two or more communicating parties and thus necessarily involves several entities. This leads us to *branch processes* that are used exactly for this purpose: to group together entities that form a protocol. Only branch processes may aggregate other processes.

Figure 3 shows the branch process `Generic_WS` as an architectural description of `WeatherSystem`. The channels for communication are represented in Kannel as *associations* (depicted as a line; *separate* associations are depicted as a dotted line), here `CONTROL`, `SENSOR` (separate), and `ALARM`. `Generic_WS` groups together a `Controller` process together with several `Sensor` processes. Each `Sensor` in turn is a branch process containing a `Timer` and a `CPU` process. In addition to grouping, the branch processes can perform initialization of their (local) component processes by invoking their methods—this is the only context in Kannel where processes may interact with method calls; thus there is quite a strong form of aggregation between a branch process and its components.

[4]Each service interface is assigned a port identifier (e.g., `service`) that is used within associations and within the statechart.

**Type relations for leaf processes** A process interface for a leaf process enumerates a set of service interfaces that the leaf is prepared to serve. Kannel requires that the visible behavior of the concrete subtypes be equivalent. For example, when a `Controller` receives a `query` message (see Figure 3), it will respond with a `report`: this behavior is required to remain the same for all its concrete subtypes (in the example just `AVG_Controller`), as stated in Definition 8 of Section 3. Of course, this is not full behavioral compatibility, since the actual *content* of the messages is not required to be the same. It appears that full behavioral compatibility is often too restrictive, since one usually wants to model a slight semantic change while preserving substitutability (consider, for example, a `Log_controller` process that records sensor reports within persistent storage).

**Type relations for branch processes** The previous discussion on behavior also applies to branches—except that in this case the services are not implemented by the branch itself but rather by some component therein. In addition to service interfaces, a branch process may also have a *protocol assertion* that indirectly states the protocol implemented by the component processes. The assertion identifies an association within a subtype of the interface containing the association; for example, the `SENSOR` assertion within the interface for `WeatherSystem` must appear (in one way or another, see 4.2) within every concrete subtype.

Protocol assertions are not just labels; they carry significant semantic weight by imposing the requirement that the endpoints of the association identified by the assertion *become part of the service interface* of the branch for the purposes of type checking. Of course, the endpoints are not visible to the clients of the branch—rather, the behavior exhibited at the endpoints becomes part of the branches' type and must be the same for every subtype. This is a subtle way to integrate the behavior of the most important component processes into the type definition. Protocol assertions are reminiscent to the concept of *structural conformity* in [HaG96]; however, they are more flexible by allowing the designer to leave out the components that are "uninteresting" with regard to layer behavior (e.g., endpoints of the `ALARM` association within `Sensor`).

The protocol assertion must also state (with the keyword `separate`) whether the subparts using the protocol are located within separate address spaces, e.g., in different machines. This is required in order for the subtypes to be meaningfully substitutable—were it not so, the Kannel compiler would not always be able to partition the processes deterministically in a context where process interface attributes are used.

**The structure of `Generic_WS`** Figure 3 illustrates the `Generic_WS` process. Since it is a branch process, there is no controlling statechart. Three component processes (`AVG_Controller`, `CPU` and `Timer`) are leaf processes and thus have a controlling statechart[5]. As an example, we describe the structure of the `CPU` statechart—see the appendix for full details.

The `CPU` consists of a single hierarchical, concurrent state `running` (departing from the traditional notation of [Har87], concurrent states are shown with shaded background) that has two submachines: a `sleeper` that communicates with the timer process to obtain periodic alarms and a `sender` that synchronizes with the state `sleeper.fire` in order to send a probe to the controller. Kannel statecharts do not support message broadcasting, but have a few useful additional features such as the internal `nil` message that has an infinitely low priority and is fired whenever there are no other available messages to process (for example, the `sleeper.start` state uses a `nil` transition to request the initial `wakeup` message from the timer).

```
process Generic_WS < WeatherSystem is
  c : AVG_Controller;
  sensors : vector (SMAX) Sensor;
    ...
assoc
  SENSOR: c.peer and separate sensors.all.peer;
  c.up and service
end Generic_WS
```

[5] The `Timer` process is part of the standard library and its structure is thus omitted.

The condenced textual definition for `Generic_WS` shown above lists its components and their associations over process ports (e.g., `c.peer`). The subtype relation is specified with a `<` symbol. Note the required assertion label `SENSOR` before the association between the sensors and the controller.

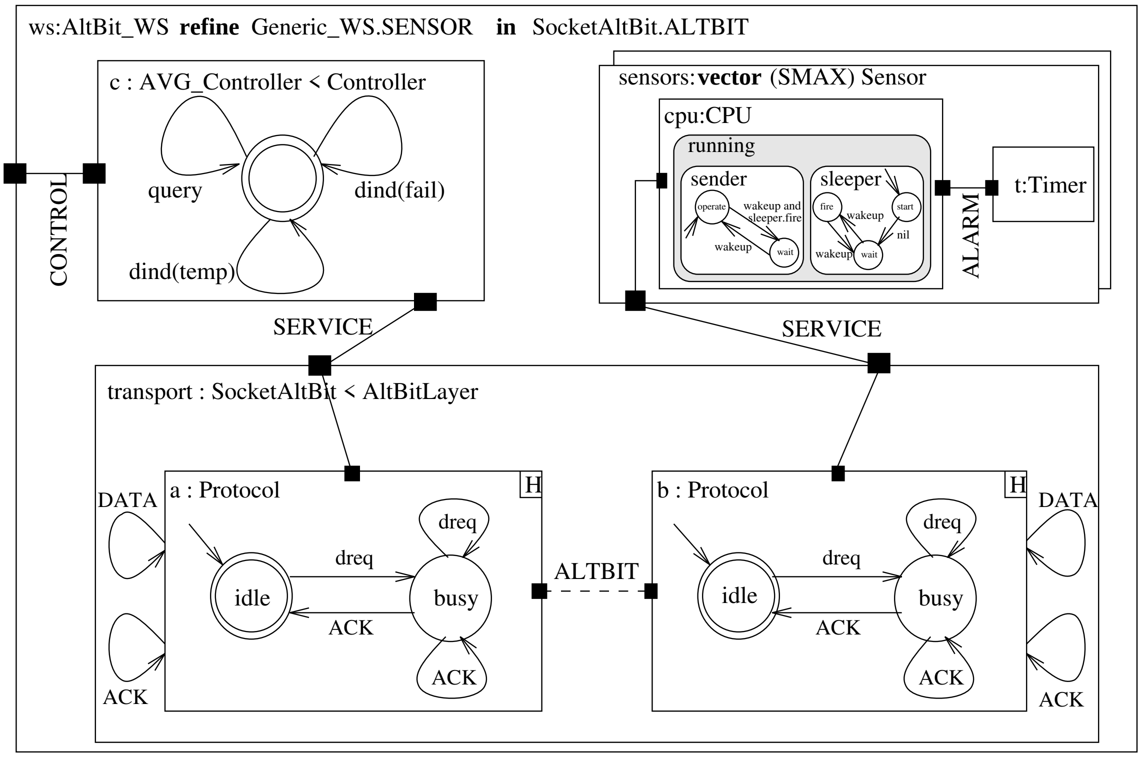


Figure 4: The refined `AltBit_WS` process.

## 4.2 The refinement of `Generic_WS`

Now we have our framework ready for the refinement of the `SENSOR` association over an alternating bit transport service layer. The mechanisms we are going to present are based on the observations (see Section 3) that (a) the behavior of the processes to be refined does not change at all on service interfaces except for the one being refined and that (b) the actual structure of the statecharts within the original and refined processes does not really matter if the *language* of the processes (the set of accepted traces) remains the same. The former point is important since we want to capture the refinements within the Kannel type system in a flexible manner. The latter point arose as the result of investigating mechanisms for reusing the synchronization behavior of a process (sometimes called the "inheritance anomaly" problem).

Figure 4 illustrates the result of refining the `Generic_WS` process with respect to protocol `SENSOR` within an alternating bit transfer service (cf. the corresponding abstraction in Figure 2). We omit its description in order to keep the presentation compact; see the appendix for details. However, two things are worth pointing out. First, note the use of *history* information (depicted with a capital H in the upper right hand corner of the process) within the statechart for `Protocol` to capture the fact that the processing of a `DATA` message always ends up in the same substate where it was received. Second, note having `idle` as a *final* state (depicted in Figure 4 with an additional circle) to mark the logical endpoint of a message delivery[6]. We proceed by presenting

[6] In other statecharts the implicit final state rule of Kannel is applied. The rule says that every state without any (outgoing) transitions is implicitly considered final.

the textual form of the mechanisms and by discussing their semantics and effects on code reuse.

#### 4.2.1 Branch process refinement

Refinement can only be done based on a protocol assertion that identifies an association within a branch process. Thus, the assertion enables stepwise extension of a process' components. The structure of the refined `AltBit_WS` process is shown below:

```
process AltBit_WS
  refine Generic_WS.SENSOR
      in SocketAltBit.ALTBIT
is
  transport : SocketAltBit
assoc
  transport.up1 and c.peer;
  transport.up2 and sensors.all.peer
end AltBit_WS
```

The subtyping section of the refined process is replaced with a *refine* clause that names two protocol assertions, the original and its replacement. Note that both assertions are on peer level—we thus establish a binding between two adjacent layers of which the latter is less abstract than the former. In addition, the (concrete) process to be refined is also specified. The effects of the mechanism are as follows:

- The `AltBit_WS` process becomes a subtype of the process interface to which the `SENSOR` assertion belongs (and for which `Generic_WS` must be a subtype).
- The associations and other code (private methods and routers) are reused as is within the `AltBit_WS` process except for the association to be refined.
- The leaf processes residing at the endpoints of the refined association become undefined and must be superimposed into `AltBit_WS` as explained in Section 4.2.2.
- Any branch processes between the leaf processes are locally redeclared with a new service interface for the refined association. In our example, the `Sensor` process gets locally redeclared within `AltBit_WS`; its old `peer:SENSOR` interface changes into `peer:SERVICE`.

Note that the transformed association endpoints remain unassociated and must be explicitly given. In the above code fragment a new process component `transport` is plugged into the now unassociated ports. This also resolves any potential ambiguity about the refining process component (in general, there might be more than a single `SocketAltBit` component).

As mentioned in the very beginning of this section, the refinement resembles inheritance in the sense that it combines code reuse with subtyping. However, here reuse has semantics that differ from the "simple textual copy" semantics used in the reuse mechanism for local Kannel classes: it results in a set of (compiler-generated) new types for the branch processes that are part of the refined association.

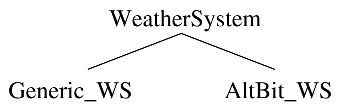


Figure 5: Subtypes of `WeatherSystem`.

The resulting type relations are shown in Figure 5. Note that `AltBit_WS` is *not* a subtype of `Generic_WS` but rather a subtype of its interface—this is in harmony with the whole Kannel type system which is based on the idea that all type relations are abstract and should not be confused with code reuse (for which there is a separate mechanism).

The leaf processes at the endpoints of the refined association remain to be respecified. Here the situation is more complex, since their statecharts are populated with receptions and transmissions of messages that are part of the abstract protocol.

### 4.2.2 Leaf process refinement

The types of the original leaf processes and their refinements do not remain compatible, since one service interface gets changed in the processes. However, there still exists significant similarity that we wish to exploit. A further consideration is that both process definitions must coexist within the source code. Generally, this can be tackled with scoping or renaming. We have chosen the former approach, since renaming tends to be messy and since coming up with meaningful names for the entities is a major burden in itself. The refined processes `CPU` and `AVG_Controller` are shown below:

```
process CPU in AltBit_WS map
  s1 : peer ! dreq(temp.create(unit, t.read));
  s2 : peer ! dreq(fail.create(unit))
is action
  running
end CPU;

process AVG_Controller in AltBit_WS map
  temp,fail in dind
is action
  AVG_Controller
end
```

The subtyping section of the refined process is replaced with a *superimposition* clause **in** ($a$) **map** ($b$), where ($a$) names the refined branch process into which the refined type is to be superimposed and ($b$) provides a transformation mapping for all messages travelling in the abstract association which is being refined.

**The transformation mapping** The statecharts in Kannel are granularized on the level of message *receptions*: one reception may result in several transmissions (depending on the transition action). This is a convenient notation for specifying the behavior since one does not have to clutter the state space with states which immediately fire by transmitting a message.

In the context of transformation mappings this extra convenience has a price: the designer must give explicit labels for all message transmissions into the abstract peer association; these labels are then used within the mapping to provide the refined transmission statement. For example, the message transmissions into the `SENSOR` association within the `CPU` process are wrapped as `dreq` messages into the `SERVICE` association within the refinement; the original `CPU` definition contains the labels `s1` and `s2` (see the appendix) referred to in the mapping.

For message receptions the situation is simpler, since they can use the originating state as a natural label. In our example the receptions within `AVG_Controller` are all wrapped into `dind` messages—this is a degenerate case, since there is only a single state. In the general case, however, the receptions of, say, message `M` may be refined into *distinct* messages `N1` and `N2` depending on the current state. The mechanism extends to situations like this by using the originating state name as the mapping label.

The reception mapping must specify the exact entity into which the abstract message is transformed in order to enable the Kannel compiler to instrument the receptions with the necessary disambiguations. For example, since both `temp` and `fail` messages are received within a `dind` message, the compiler must instrument the refined `AVG_Controller` process with a dynamic type check which determines the exact type[7].

[7] All Kannel objects carry run-time type information.

### 4.2.3 Reuse of leaf processes

As we have seen, the distinction between subtyping and code reuse extends quite naturally in Kannel from local classes to processes. With processes, subtyping considers the language induced by their state automata (see Section 3).

Since the language has no forced relation with any given automaton (a given language may be accepted by several structurally different automata), subprocesses of a given interface may use any means whatsoever to implement the language. This is reflected in the reuse mechanism which is quite liberal, allowing practically any modifications to a reused state machine. This is in contrast with approaches that use inheritance to reuse a given state automaton and hence have to force severe restrictions on the set of allowed modifications (e.g., [HaG96, CHB92]). The Kannel leaf process reuse mechanism comes in two flavors which are augmented with special syntax:

- The standard liberal code reuse mechanism that can be used for arbitrary leaf processes.
- The disciplined superimposition mechanism (Section 4.2.2) that is used solely in the context of refinement.

## 4.3 Discussion

**Ensuring the preservation of behavior** The notion of branch processes allows one to have type relations between refinements, which is useful in practice. However, a more fundamental issue concerns the amount of checking a compiler can do when confronted with the refinement mechanisms. The statechart model augmented with the notion of final states allows us to speak about the language of a process, and subsequently we have more freedom in modifying the statecharts. The transformation mapping provides the compiler a rough estimate of the total functions $f_1$ and $g_3$ (and, symmetrically, $g_1$ and $f_3$); see Definition 8 in Section 3. The remaining two mappings are defined implicitly by the constraints set upon refinement. They do require, however, that the compiler can perform (some fairly unsophisticated) statechart slicing in order to reveal the mappings from the message flow.

**Varying-height protocol stacks** The refinement mechanism enables interesting variations within the peer branches of a process interface. Since refinement effectively increases the number of subcomponents contained within parent by one, the peer branches may represent protocol stacks of differing "heights" and still be type-compatible. This is illustrated in Figure 5 where the left subtype has height 1 and the right subtype has height 2.

**Further work** The presented formalism works nicely in a context where the leaf processes residing at the endpoints of the abstract association are concrete. If they are replaced with interfaces, the situation gets more complex, since several structurally differing statecharts may now exist within the eventual leaves. Our current solution to this is to require that superimpositions be provided for all the concrete subtypes of a leaf interface, but alternative solutions, such as selective pruning of the type tree at the point of refinement are worth considering. A related problem arises in a situation where one or more *branch* processes that are part of the abstract association are represented by a process interface. Clearly the requirement that all final leaves within subtypes must then be superimposed is too stringent one.

The possibility to refine several abstract associations with differing protocol assertions needs more consideration. Currently, only a single association may be refined at a time.

# 5 Conclusions

We have presented a systematic methodology for developing communications software by stepwise refinement of protocols. The characteristics of the problem of protocol refinement have been analyzed and other potential solutions have been outlined. As a practical approach to protocol refinement, we have shown how the mechanism can be expressed and implemented using the object-oriented protocol engineering language Kannel.

With regard to other suggested approaches, the main novelty of ours is being *constructive* rather than theoretical. Unlike the approaches of program refinement [Bac88], data type refinement [Nip89], protocol conversion [PeL93], and top-down protocol specification refinement [LiM88], whose main objective is to employ formal specifications to formally prove the correctness of system evolution, we have developed a programming language by which a protocol designer can express the system evolution on a proper level of preciseness. The task of verifying the central formal properties of refinement is laid on the Kannel system, not on the designer. Of course, this kind of "automatic verification" done by the Kannel compiler is less complete than a formal proof but still powerful enough to capture the most fundamental errors.

Another language-based approach to stepwise refinement of communicating systems is presented in [SLR95]. As Kannel, the *RL* language makes it possible to gradually evolve distributed systems by incrementally modifying the system's architecture, its state-behavior, and the types of communicated messages. The refinements are validated with an analysis over the derived message types and the state space of the underlying state automata. The main difference to Kannel is that RL does not rely on object-oriented features, whereas these are the key factor in Kannel for achieving the mechanism of refinement.

Being object-oriented in general, Kannel shares some features with general-purpose object-oriented programming languages, such as Eiffel [Mey92] and Sather [SOM93]. The key difference to these is that Kannel is a special-purpose language with a number of central facilities tuned especially towards protocol engineering. For instance, the refinement mechanism presented in this paper is not intended for applications of arbitrary kind but just for the development of distributed systems with a communication protocol in the core. By focusing refinement on one ("peer") side of a process interface at a time, our approach shares some ideas with dividing the interface of a class into two distinct categories, a client interface and a specialization interface [Lam93].

From the applications' point of view, Kannel is closely related to the formal language family of telecommunications, in particular to the object-oriented variant of SDL, *SDL-92* (OSDL) [FæO92]. With respect to the theme of this paper, these two languages differ in that Kannel considers incremental refinement as a semantic mechanism of special kind, whereas a similar effect has to be simulated in SDL-92 using conventional inheritance and virtuals without formal support.

The basic implementation of Kannel is complete, and the refinement features described in this paper are currently under implementation.

# References


[Bac88] R.J.R. Back: A Calculus of Refinements for Program Derivations. *Acta Informatica* 25, 1988, 593–624.

[CaL89] K.L. Calvert, S.S. Lam: Deriving a Protocol Converter: a Top-Down Method. In: Proc. ACM SIGCOMM'89 Symp. on Communications Architectures & Protocols, Austin, Texas, 1989. *ACM Computer Communications Review* 19, 4, 1989, 247–258.

[CHB92] D. Coleman, F. Hayes, S. Bear: Introducing Objectcharts or How to Use Statecharts in Object-Oriented Design. *IEEE Transactions on Software Engineering* 18, 1, 1992, 9–18.

[Cus91] E. Cusack: Refinement, Conformance and Inheritance. *Formal Aspects of Computing* 3, 1991, 129–141.

[FæO92] O. Færgemand, A. Olsen: Introduction to SDL-92. *Computer Networks and ISDN Systems* 26, 1994, 1143–1167.

[Gre86] P.E. Green, Jr.: Protocol Conversion. *IEEE Transactions on Communications* 34, 3, 1986, 257–268.

[GHP94] K. Granö, J. Harju, J. Paakki, T. Järvinen: Proposal for a Protocol Engineering Language. Technical Reports TR-6, Department of Computer Science and Information Systems, University of Jyväskylä, 1994.

[HaG96] D. Harel, E. Gery: Executable Object Modeling with Statecharts. In: *Proc. 18th International Conf. on Software Engineering* (ICSE-18), Berlin, Germany, 1996. IEEE Computer Society Press, 1996, 246–257.

[Har87] D. Harel: Statecharts: A Visual Approach to Complex Systems. *Science of Computer Programming* 8, 1987, 231–274.

[Jon80] C.B. Jones: *Software Development — A Rigorous Approach.* Prentice-Hall, 1980.

[Jon89] B. Jonsson: On Decomposing and Refining Specifications of Distributed Systems. In: *Stepwise Refinement of Distributed Systems.* Lecture Notes in Computer Science 430, Springer-Verlag, 1989, 361–385.

[Lam93] J. Lamping: Typing the Specialization Interface. In: Proc. OOPSLA'93, Washington, DC. *ACM SIGPLAN Notices* 28, 10, 1993, 201–214.

[LiM88] D.-H. Lim, T.S.E. Maibaum: A Top-Down Step-Wise Refinement Methodology for Protocol Specification. In: *Proc. Int. Conf. on Concurrency* (Concurrency'88), Hamburg, FRG, 1988. Lecture Notes in Computer Science 335, Springer-Verlag, 1988, 197–221.

[Mey92] B. Meyer: *Eiffel — The Language.* Prentice-Hall, 1992.

[Nip89] T. Nipkow: Formal Verification of Data Type Refinement — Theory and Practice. In: *Stepwise Refinement of Distributed Systems.* Lecture Notes in Computer Science 430, Springer-Verlag, 1989, 561–591.

[PeL93] M. Peyravian, C.-T. Lea: Construction of Protocol Converters Using Formal Methods. *Computer Communications* 16, 4, 1993, 215–228.

[SLR95] R. Sekar, Y.-J. Lin, C.R. Ramakrishnan: Modelling Techniques for Evolving Distributed Applications. In: D. Hogrefe, S. Leue (eds.): *Formal Description Techniques VII.* Chapman & Hall, 1995, 461–476.

[SOM93] C. Szyperski, S. Omohundro, S. Murer: Engineering a Programming Language: The Type and Class System of Sather. In: J. Gutknecht (ed.): *Programming Languages and System Architectures.* Lecture Notes in Computer Science 782, Springer-Verlag, 1993, 208–227.

[Tha94] S.R. Thatté: Automated Synthesis of Interface Adapters for Reusable Classes. In: *Conf. Record 21st ACM SIGACT-SIGPLAN Symp. on Principles of Programming Languages* (POPL'94), Portland, Oregon, 1994, 174–187.

[WeZ88] P. Wegner, S.B. Zdonik: Inheritance as an Incremental Modification Mechanism or What Like Is and Isn't Like. In: *Proc. Second European Conf. on Object-Oriented Programming* (ECOOP'88), Oslo, Norway, 1988. Lecture Notes in Computer Science 322, Springer-Verlag, 1988, 55–77.

[YeS94] D.M. Yellin, R.E. Strom: Interfaces, Protocols, and the Semi-Automatic Construction of Software Adaptors. In: Proc. OOPSLA'94, Portland, Oregon. *ACM SIGPLAN Notices* 29, 10, 1994, 176–190.

# Source code for the weather system specification

```
const SMAX ::= 5;
const TIMEOUT ::= 10;

class Temperature is
  unit, val : integer
end Temperature;

channel CONTROL is
  view Requests is query : event end;
  view Results is
      report : real;
      faulty : integer
  end Results
end CONTROL;

channel SENSOR is
  temp : Temperature;
  fail : integer
end SENSOR;

process interface Controller is
  peer : SENSOR (in);
  up : CONTROL (in Requests, out Results)
end;

process AVG_controller < Controller is
  const THRESHOLD ::= 2;
  buf : list [Temperature];
  old : real;
  average (list [Temperature]) : real is
    sum::= 0;
    iter : list_iter[Temperature];
    iter.reset(arg);
    loop iter.done.until;
      sum:= sum + iter.next
    end loop;
    return sum / buf.length
  end average
final arcs
  temp -> {
     buf.add(temp);
     if buf.length = SMAX then
        res::= average(buf);
        if (old-res).abs > THRESHOLD then
           up ! report.create(res)
        end if;
        old:= res; buf.clear
     end if }
  query -> { up ! report.create(old) }
  fail -> { up ! fail }
end AVG_controller;

process CPU ports
  peer  : SENSOR (out);
  clock : ALARM (in Notices, out Settings)
is
  initialize(u,t:integer) is
     unit:= u; delta:= d end;
  unit, delta : integer;
  t : SensorIO
action
  and state running is
    state sender is
      state operate arcs
        wakeup and sleeper.fire -> {
          if t.ok then
             s1: peer ! temp(unit, t.read)
          else
             s2: peer ! fail(unit)
          end if;
          go wait }
      end;
      state wait arcs wakeup -> { go operate } end;
    end sender;
    state sleeper is
      state start arcs
        nil -> { clock ! set(delta); go wait } end;
      state wait arcs
        wakeup -> { clock ! set(delta / 2); go fire } end;
      state fire arcs
        wakeup -> { clock ! set(delta / 2); go wait } end
    end sleeper
  end
end CPU;

process Sensor ports
  peer : SENSOR (out)
is
  initialize(integer) is cpu.initialize(arg,TIMEOUT) end;
  t : Timer;
  cpu : CPU
assoc
  cpu.peer and peer;
  cpu.clock and t.service
end Sensor;

process interface WeatherSystem is
  service : CONTROL(in Requests, out Results);
  protocol SENSOR separate
end;

process Generic_WS < WeatherSystem is
  c : AVG_controller;
  sensors : vector (SMAX) Sensor;
  main[sensors] is
    i::=0;
    loop (i < SMAX).while;
      sensors(i).initialize(i); inc(i)
    end
  end
assoc
  SENSOR: c.peer and separate sensors.all.peer;
  c.up and service
end Generic_WS;
```

```
interface UserData > Temperature,integer
is end;

process AltBit_WS
  refine Generic_WS.SENSOR in
         SocketAltBit.ALTBIT
is
  transport : SocketAltBit
assoc
  transport.up1 and c.peer;
  transport.up2 and sensors.all.peer
end AltBit_WS;

process CPU in AltBit_WS map
  s1 : peer ! dreq(temp.create(unit, t.read));
  s2 : peer ! dreq(fail.create(unit))
is action
  running
end CPU;

process AVG_controller in AltBit_WS map
  temp,fail in dind
is action
  AVG_controller
end;

process System is
  u : User;
  ws : WeatherSystem;

  process User ports
    control:CONTROL(out Requests; in Results)
  is arcs
    userinput -> { control ! query.create }
    report -> {
      stdout.print("Temperature: %d\n", report) }
    faulty -> {
      stdout.print("Error in unit %d\n", faulty) }
  end User;

  main(vector string) is
    if arg(0) = "test" then
       ws:= new Generic_WS
    else
       ws:= new AltBit_WS
    end if
  end main
assoc
  u.control and ws.service
end System;

channel SERVICE is
  view requests is dreq : UserData end;
  view results  is dind : UserData end
end SERVICE;
```

```
process interface AltBitLayer is
  up1, up2 : SERVICE (in requests, out results)
  protocol ALTBIT separate
end AltBitLayer;

process SocketAltBit < AltBitLayer is
  a,b : Protocol
assoc
  ALTBIT: a.peer and separate(Socket) b.peer;
  a.up and up1;
  b.up and up2
end SocketAltBit;

class Packet is
  contents : UserData;
  seq : boolean
end Packet;

channel ALTBIT is
  DATA : Packet;
  ACK  : boolean
end ALTBIT;

process Protocol ports
  up : SERVICE (in requests, out results);
  peer : ALTBIT (in out)
is
  todo : list [DataReq];
  next::= false;              -- next bit to send
  received::= true            -- most recent bit received
action
  init idle traced;           -- history & initial state
  final state idle arcs
    dreq -> { todo.add(Dreq); peer ! DATA(dreq,next); go busy }
  end idle;
  state busy arcs
    dreq -> { todo.add(Dreq) }
    ACK ->  { if ACK = next then
                 todo.remove_first;
                 next:= not next;
                 if todo.empty then
                    go idle
                 else
                    peer ! DATA(todo.first, next)
                 end if
              else              -- retransmit
                 peer ! DATA(todo.first, next)
              end if }
  end busy
arcs
   ACK  -> {}                 -- ignored
   DATA -> { if DATA.seq <> received then
                received:= DATA.seq;
             up ! Dind(DATA.contents)
          end if;
          peer ! ACK(received) }
end Protocol
```