The Sophtalk reference manual

Ian Jacobs, Francis Montagnac, Janet Bertot, Dominique Clement, Vincent Prunet

UNITÉ DE RECHERCHE
INRIA-SOPHIA ANTIPOLIS

Rapports Techniques

N°150

Programme 2

Calcul symbolique, Programmation et Génie logiciel

THE SOPHTALK REFERENCE MANUAL

Ian JACOBS
Francis MONTAGNAC
Janet BERTOT
Dominique CLEMENT
Vincent PRUNET

Institut National
de Recherche
en Informatique
et en Automatique

2004 route des Lucioles
B.P. 93
06902 Sophia-Antipolis
France

Février 1993

The Sophtalk Reference Manual

Ian Jacobs Francis Montagnac Janet Bertot
Dominique Clément Vincent Prunet

March 2, 1993

Sophtalk

Abstract

This is the reference manual for the LeLisp implementation of the Sophtalk system. Sophtalk is a set of tools that enable one to program the interaction between objects following an event model of communication. System objects, called *stnodes*, emit messages when significant events occur, such as the termination of a computation, a request for a service from another object, error conditions, etc. Messages circulate asynchronously in a network of stnodes. An stnode's type determines which messages it will receive; upon reception, an action corresponding to the message and stnode instance is triggered. Messages may also circulate between processes.

The manual describes the three modules of Sophtalk: stnode, a multicast communication mechanism; stio, an extension of the standard LeLisp asynchronous and synchronous i/o mechanisms; and stservice, which offers interprocess communication at the shell and LeLisp levels.

Manuel de Référence de Sophtalk

Résumé

Ceci est le manuel de référence de l'implantation en LeLisp du système Sophtalk. Sophtalk est un ensemble d'outils pour décrire l'interaction d'objets selon un modèle de communication basé sur des événements. Les objets primaires du système, dits *stnodes*, émettent des messages lors d'événements significatifs, tels que la terminaison d'un calcul, une requête pour un service fourni par un autre objet, ou encore une erreur, etc. Les messages circulent de façon asynchrone dans un "réseau" de stnodes. Le type d'un stnode détermine l'ensemble des messages qu'il écoute; lors de la reception, une action qui est fonction du message et du stnode est déclenchée. Les messages peuvent également circuler entre des processus distincts.

Ce manuel décrit les trois modules de Sophtalk: stnode, un mécanisme de communication par envoi de messages; stio, une extension des mécanismes d'entrée/sortie synchrones et asynchrones fournis en standard par LeLisp; enfin stservice, qui permet d'établir la communication entre processus au niveau du shell et de LeLisp.

Contents

Chapter 1

Stnode reference manual

1.1 Stnode reference manual

In this section we present the functions that create and manipulate stnodes. For details about stnode types, please consult appendix A.

1.1.1 Type declaration

(st-declare type private-inputs^{list} **private-outputs**^{list} [**field**$_1$] ... [**field**$_n$]) $\rightarrow$ *stnode-type*

Declares a new stnode-type. The **type** may be an unpackaged or packaged Le-Lisp symbol. In the latter case 1) the type must be preceded by a sharp symbol (#) and 2) the type is the whole **type** name, but private ports are only declared for the last portion of the name. All ancestor types of this last portion *must* be previouly declared, otherwise an error is provoked. Redeclaring an existing type produces a warning message and erases the previous declaration. When redeclaring a type, subtypes previously defined may function improperly under the new declaration. The result type inherits its input and output ports from parent types, and possesses its own **private-inputs** and **private-outputs**. In this version of Sophtalk, multiple declarations of port names are forbidden, so you may neither give the same name to an input and output port nor redeclare an inherited port name. The methods private-inputs, private-outputs, inputs, and outputs are all defined automatically to access the port names of a type instance. The latter two return lists containing both the inherited and private ports. Note that none of the arguments should be quoted. When present, instances of the new **type** will also have additional fields, also inherited by subtypes. These fields are accessible through the call `(send '<fieldname> <instance>)`

Example:

```
? ; No quote before mytype or the port lists!
? (st-declare mytype (in1 in2) (out1))
= #:stnode:mytype

? ; Define subtype "sub" once supertype "mytype" declared.
? ; The "#" is necessary before mytype.
```

```
? (st-declare #:mytype:sub (subin1) (subout1 subout2))
= #:stnode:mytype:sub

? ; Error if supertype not previously declared.
? (st-declare #:a:b:c (in) (out))
** st-declare : undefined-node-type : #:stnode:a:b

? ;Inherited ports can't be redefined.
? (st-declare #:mytype:sub:sub2 (subin1) ())
** st-declare : port multiply defined :
              (#:stnode:mytype:sub:sub2 subin1)
```

See the section below on instance creation for examples using the private-inputs, private-outputs, inputs and outputs methods.

1.1.2 Instance creation

(st-make-opaque-node) → *stnode*

Creates an instance of the class #:stnode. The stnode has no input or output ports. One usually creates an opaque node to encapsulate other stnodes.

(st-create type [a-list]) → *stnode*

Creates an instance of the indicated stnode **type** (quoted symbol). Triggers an error if **type** has not been previously declared. The resulting stnode encapsulates nothing. When present, the **a-list** should contain `(fieldname . value)` pairs where `fieldname` is one of additional structural names provided in the type declaration or a parent type declaration.

(st-create-named type name [a-list]) → *stnode*

Creates an instance of the indicated stnode **type** (quoted symbol). Triggers an error if **type** has not been previously declared. The resulting stnode encapsulates nothing. The name of the instance is given by **name**. When present, the **a-list** should contain `(fieldname . value)` pairs where `fieldname` is one of structural names provided to the function `st-declare`.

(st-name stnode) → *stnode*

Get the name of the **stnode**.

(st-set-name stnode name) → *stnode*

Set the name of the **stnode** to **name**.

(st-map-inputs stnode lambda [arg$_1$] ... [arg$_n$]) → *nil*

Applies **lambda** to the input ports (inherited and private) of **stnode**, with any additional arguments. The first argument of **lambda** is the stnode, the second the input port name. When present, the optional arguments are evaluated in parallel, and only once.

(st-map-outputs stnode lambda [arg$_1$] ... [arg$_n$]) → *nil*

Applies **lambda** to the ouput ports (inherited and private) of **stnode**, with any additional arguments. The first argument of **lambda** is the stnode, the second the output port name. When present, the optional arguments are evaluated in parallel, and only once.

(#:stnode:prin stnode [depth]^{integer}**)** → *nil*

Pretty prints the **stnode** on the current output channel. The port names are displayed if the variable ***st-print-ports*** is non nil. By default, the value is nil. If stnode is non-atomic, the contained stnodes are recursively displayed up to a depth specified via the **depth** argument when present, or the value of the variable ***st-print-network***. When **depth** is 0, all descendents of **stnode** are displayed. The default value for ***st-print-network*** is 1.

Atomic stnodes

An atomic stnode may encapsulate any Le-Lisp object except nil. An object may be encapsulated by several stnodes, but an stnode may only encapsulate one object at a time.

In what follows, the term *empty* refers to an stnode that encapsulates nothing, *atomic* refers to an stnode that encapsulates an object, and *non-atomic* refers to an stnode that encapsulates other stnodes. These states are mutually exclusive.

(st-atomp stnode) → *lisp*

Returns the Le-Lisp object encapsulated by **stnode**. Returns nil if **stnode** is empty or non-atomic.

(st-object stnode) → *lisp*

Returns the Le-Lisp object encapsulated by **stnode**. Triggers an error if **stnode** is empty or non-atomic.

(st-store stnode object) → *stnode*

Makes **stnode** an atomic stnode that encapsulates **object**. **object** may not be nil. After storage, no other object may be encapsulated by **stnode** unless **st-unstore** is used. Similarly, **stnode** may not encapsulate other stnodes unless **st-unstore** is used. Triggers an error if **stnode** is already atomic or non-atomic.

(st-store-and-set-action stnode object [package]) → *stnode*

Like **st-store**, but additionally defines the input callback methods. Each method name is the **package** prefix followed by the port name. If **package** is absent, the type of **object** is used as the callback prefix. **N.B.** This function differs slightly from `st-action` defined in the section on callbacks. The function `st-action` associates an input port with a complete function name whereas the present function declares a common prefix from which function names will be constructed.

(st-unstore stnode) → *stnode*

Disassociates **stnode** from the object it encapsulates. Does nothing if **stnode** is empty or non-atomic.

Stored object primitives

(stobject-atomp object type) → *stnode-list*

Returns the list of stnodes that 1) encapsulate **object** and 2) are instances of a subtype of **type**. When **type** is nil, returns the entire list of encapsulating stnodes. Returns nil if **object** is not encapsulated.

(stobject-nodes object type) → *stnode-list*

Like `stobject-atomp`, but triggers an error if **object** is not encapsulated.

(stobject-unstore object type) → *object*

Unstores **object** from all encapsulating stnodes that are instances of a subtype of **type**. When **type** is nil, **object** is unstored from all encapsulating **stnodes**. Does nothing if **object** is not encapsulated.

Example:

```
? (st-declare foo (in) (out))
= #:stnode:foo

? (setq son1 (st-create 'foo))
= foo:[net<0>]

? (defstruct obj field1)
= obj

? (setq obj (#:obj:make))
= #:obj:#[()]

? (st-store-and-set-action son1 obj)
= foo:[atom<obj>]  ;; <obj> is the stored object type.

? (st-atomp son1)
= #:obj:#[()]

? (stobject-atomp obj 'foo)
= (foo:[atom<obj>]) ;;a list!

? (stobject-atomp obj 'bar)
= ()

? (st-store son1 2)
** st-store : atomic node : foo:[atom<obj>]

? (st-unstore son1)
= foo:[net<0>]

? (st-store son1 2)
= foo:[atom<fix>]
```

```
? (st-map-inputs son1 (lambda (node port) (print port)))
in
= ()
```

1.1.3 Non-atomic stnodes

Non-atomic stnodes contain other stnodes which may send messages to one another over bus lines.

(st-up stnode) → *stnode*

Returns the stnode that encapsulates **stnode**, or nil if **stnode** is not encapsulated.

(st-include parent^{stnode} **descendent**^{stnode} **[alias-list])** → *stnode*

Encapsulates **descendent** with **parent**. Each port is plugged into bus lines that link the ports of the other descendents of **parent**. It is possible to connect a **descendent** port to several bus lines. The **alias-list**, if present, is a Le-Lisp A-list where the key of each pair is a port name and the value is a bus name to be connected. Returns **parent**.

(st-include* parent^{stnode} **[descendent**^{stnode} **| (cons descendent alias-list)]*)** → *stnode*

Like **st-include**, but allows you to include several stnodes simultaneously. Returns **parent**. See the example below.

Example: Given an stnode with input ports A, B, and C and output ports E, F, and G, consider the effect of the following alias lists:

- $((A\ .\ A')(A\ .\ A'')(E\ .\ F))$ connects B, C, F, and G to the bus wires of the same name. A is connected to the buses A' and A''. E is connected to the bus F. Renamed ports such as A or E are not connected to the bus wire of the same name.
- $((A\ .\ ())(E\ .\ E)(E\ .\ F))$ connects B, C, F, and G to the bus wires of the same name. A is not connected at all. E is connected to the wires E and F.
- $((A\ .\ ())(A\ .\ B))$ is equivalent to $((A\ .\ B))$.

(st-link old^{stnode} **new**^{stnode} **[alias-list])** → *stnode*

Makes **old** and **new** descendents of the same stnode. If **old** is already encapsulated, **new** shares the same parent. If **old** is not encapsulated, an opaque stnode is created as a mutual parent. This function assumes that **new** is not already encapsulated. When present, the **alias-list** is used to rename **new**'s ports. No alias is used for **old**'s ports when an opaque parent is created. Returns the parent stnode.

(st-exclude stnode) → *stnode*

Removes **stnode** as a descendent of its parent. Does nothing if **stnode** is an orphan. Returns **stnode**. This function does not affect the descendents or encapsulated object of **stnode**.

(st-kill stnode) → *nil*

This function kills an stnode. Death proceeds as follows: the method st-terminate is applied to **stnode**. By default, this function does nothing, but may be redefined for each stnode type by the user. Next, if **stnode** is atomic, the object it encapsulates is unstored. If non-atomic, all of **stnode**'s descendents are excluded from it (but not killed). Finally, **stnode** is excluded from its parent.

(st-kill-rec stnode) → *nil*

This function kills an stnode and all of its descendents recursively. Death proceeds as follows: the method st-terminate-rec is applied to **stnode**. If **stnode** is atomic, the object it encapsulates is unstored. Otherwise, the function **st-kill-rec** is applied to all the descendents of **stnode** recursively. In either case, the method st-terminate is then applied to **stnode**. By default, the methods st-terminate-rec and st-terminate do nothing, but may be redefined for each stnode type by the user.

(st-alias-list stnode) → *alias-list*

Returns the alias-list used for connecting the ports of **stnode** to bus lines within its parent. Returns nil if **stnode** is an "orphan" or has no renamings.

(st-map parent^{stnode} **lambda [arg**$_1$**] ... [arg**$_n$**])** → *nil*

Applies **lambda**, with optional arguments if present, to every descendent of **parent**. The first argument of **lambda** must be an stnode.

The optional arguments are evaluated in parallel and only once. Does nothing if **parent** is atomic.

(st-map-alias parent^{stnode} **lambda** [**arg**$_1$] ... [**arg**$_n$]) $\rightarrow$ *nil*

Like **st-map**, but **lambda** must take at least two arguments, a st-node and an alias list.

Example:

```
? (st-declare boola (in) (out))
= #:stnode:boola

? (setq parent (st-create 'boola))
= boola:[net<0>]

? (setq s1 (st-create 'boola))
= boola:[net<0>]

? (setq s2 (st-create 'boola))
= boola:[net<0>]

; Rename "out" port of s2 to "up"
? (st-include* parent s1 (cons s2 ((out . up))))
= boola:[net<2>]

? (let ((*st-print-ports* t))
?    (send 'prin parent 2))
boola:[net<2>
    :in -> in
    :out -> out ]
  ((out . up))boola:[net<0>
                  :in -> in
                  :out -> out ]
  boola:[net<0>
      :in -> in
      :out -> out ]= boola:[net<2>]

? (st-map parent (lambda (n) (print n)))
boola:[net<0>]
```

```
boola:[net<0>]
= ()

? (setq bro (st-create 'boola))
= boola:[net<0>]

? (st-link parent bro)  ;;make bro a sibling of parent
= ():[net<2>]

? ;Opaque st-node encapsulates parent:
? (st-up parent)
= ():[net<2>]

? (st-exclude s1)
= boola:[net<0>]

? (send 'prin parent 2)
boola:[net<1>]
  ((out . up))boola:[net<0>]= boola:[net<1>]

? (st-kill-rec parent)
= ()

; parent still exists, but no parent...
? (st-up parent)
= ()
; ... and no more descendents.
? (st-map parent (lambda (n) (print n)))
= ()
```

1.1.4 Messages

Emit

Messages may be emitted by either an stnode or an encapsulated object.

(st-emit stnode port [**arg**$_1$] ... [**arg**$_n$]) $\rightarrow$ *nil*

The behavior of this function depends on whether **stnode** is atomic or non-atomic, and whether **port** is an input or output port name.

Optional arguments are transmitted with the message. Triggers an error if **port** is not a valid port name for **stnode**.

- If **port** is an input port, then:
  - If **stnode** is atomic, then the input callback for this port is activated.
  - If **stnode** is non-atomic, then the message circulates among **stnode**'s descendents with ports connected to this bus line (possibly after renaming).
- If **port** is an output port, then:
  - The message circulates among **stnode**'s siblings on the bus line(s) to which the port is connected (possibly after renaming). If an output port of **stnode**'s parent is connected to this bus line, the message is also emitted by the parent.

(stobject-emit object typed-port [**arg**$_1$] ... [**arg**$_n$]) $\rightarrow$ *nil*

The **typed-port** is a symbol made up of an stnode type followed by a port name. This function applies **st-emit** to each stnode that encapsulates **object** whose type is a subtype of the type part of **typed-port**. Triggers an error if the stnode type does not have the indicated port. Does nothing if **object** is not encapsulated.

(stobject-checked-emit object typed-port [**arg**$_1$] ... [**arg**$_n$]) $\rightarrow$ *nil*

Like **stobject-emit**, but triggers an error if no stnode matches the type constraint.

Callbacks

When an atomic stnode receives a message on an input port, it applies a callback method as follows:

```
(apply <function> <st-node> <port> <arg>*)
```

where :

- <`function`> is an action associated with the port (see `st-action` below). The function may be a symbol or a Le-Lisp lambda expression. When it is a symbol, it is converted into a function name, using the Le-Lisp inheritance mechanism, by the call:

```
(getfn (packagecell symbol) symbol)
```

- <`st-node`> is the atomic stnode.
- <`port`> is the input port name.
- <`arg`>`*` are optional arguments. Note that the <`function`> formal parameters must match the first two arguments plus these additional arguments.

If no callback method has been defined for an input port, the default action is invoked. It takes any number of arguments and does nothing.

(st-action stnode port [function]) → *function*

Gets the callback method bound to **port** for **stnode** when **function** is absent. Otherwise sets the callback function to **function**, which may be a function name or lambda expression. Triggers an error if **stnode** is not atomic or **port** is not an input port. Returns **function**.

(st-input-action stnode function) → *stnode*

Sets the callback method for all input ports of **stnode**. Triggers an error if **stnode** is not atomic. Returns **stnode**.

(st-reset-action stnode port) → *stnode*

Resets the callback method bound to **port** for **stnode** to the default method. Triggers an error if **stnode** is not atomic or **port** is not an input port. Returns **stnode**.

(st-reset-input-action stnode) → *stnode*

Resets the callback method for all of **stnode**'s input ports to the default method. Triggers an error if **stnode** is not atomic. Returns **stnode**.

Example:

```
? (st-declare green (eggs ham) (fox box))
= #:stnode:green

? (setq parent (st-create 'green))
= green:[net<0>]

? (setq s1 (st-create 'green))
= green:[net<0>]

? (st-include parent s1)
= green:[net<1>]

? (setq myobj '(1 2 3))
= (1 2 3)

? ; prefix for all inputs : '#:demo
? (st-store-and-set-action s1 myobj '#:demo)
= green:[atom<cons>]

? (de #:demo:eggs (node port myarg)
      (print "Value on """
             port
             """ is "
             myarg))
= #:demo:eggs

? (st-emit s1 'eggs 5)
Value on "eggs" is 5
= 5

? (stobject-nodes myobj '#:stnode)
= (green:[atom<cons>])

? ;In the next call, note that the typed
? ;port name is '#:green:eggs and not '#:demo:eggs !
```

```
? (stobject-emit myobj '#:green:eggs 8)
Value on "eggs" is 8
= t

? ;Set action for ham port

? (st-action s1 'ham '#:a:different:name)
= #:a:different:name

? (de #:a:different:name (node port)
      (print "Green eggs and " port))
= #:a:different:name

? (st-emit s1 'ham)
Green eggs and ham
= ham

? (st-declare rev-green (box fox) (eggs ham))
= #:stnode:rev-green

? (setq s2 (st-create 'rev-green))
= rev-green:[net<0>]

? (st-include parent s2)
= green:[net<2>]

? (st-store-and-set-action s2 myobj '#:other)
= green:[atom<cons>]

? (de #:other:box (node port myarg)
      (print "Different value on """
             port
             """ is "
             myarg))
= #:other:box

? ; Emit from s1 (s2 receives)

? (st-emit s1 'box 15)
```

```
Different value on "box" is 15
= ()

? ; Emit from s2 (s1 receives)

? (st-emit s2 'eggs 12)
Value on "eggs" is 12
= ()
```

Message semantics

One of the most common errors one can make when designing a network is to suppose that messages are sent and received in a certain order. Consider the following scenario: Stnode A sends a message M_1 that is received by two siblings B and C. Stnode B replies by sending a message M_2 that is also received by C. It may seem likely that M_1 reaches C's ears before M_2, but in fact, *nothing guarantees this order*, and design should not rely on a supposed order. To be sure, please note the following semantics:

- Messages M_1 and M_2 emitted by an stnode A directly to an stnode B (passing by no intermediate stnode's), will arrive in the same order as they are emitted.
- Messages M_1 and M_2 emitted by an stnode A to an stnode B in a different process **or** via an intermediate stnode O, may arrive in any order.
- A message M emitted by an object A and received by more than one object O_i is not received "in parallel" by all O_i. The object O_i will receive M at one moment and O_n at another moment. Object O_n may receive other messages before receiving M.

1.2 Spy reference manual

We provide a generic package which allows one to observe an stnode using a special type of stnode called a *spy*. It is possible to observe all or some of the messages emitted or received by all or some of the stnodes in a network. Adding spies to a network does not affect the behavior of the network.

1.2.1 Spy semantics

Every time a spy is created for an stnode instance of the class **C**, a special class **C:spy** is created whose input ports are the sum of **C**'s input and output ports. To spy on a **C** instance, we create a **C:spy** instance and make them siblings. This way, the spy listens to messages that circulate on the bus lines to which all ports of the **C** instance are connected. This observation technique has several consequences. A spy does not distinguish between emitters. Since a spy's ports are connected to bus lines, possibly used by many stnodes, the spy receives all circulating messages, not just those received or emitted by the stnode it spies on. Also, a spy input callback may be activated several times for a single message if the observed stnode has several ports connected to the bus line.

A spy stnode is atomic. When created, the spy default action is bound to each of the spy's ports. The default action is the value of the global variable ***st-spy-action*** when a spy is created. By default, the value of this variable is the function **st-spy-action**, described below.

1.2.2 Spy primitives

(st-set-spy stnode alias-list) → *nil*

Associates a spy with **stnode**. If **stnode** is an orphan, a parent opaque node is created for **stnode** and the spy. If **alias-list** is nil, the **stnode**'s alias list is used to connect the spy's ports to bus lines. Otherwise, the new **alias-list** is used.

(st-set-spy-rec stnode alias-list) → *nil*

Calls **st-set-spy** on every descendent of **stnode** recursively and then on **stnode**. If **alias-list** is nil, the **stnode**'s alias list is used to connect the spy's ports to bus lines. Otherwise, the new **alias-list** is used. The alias list only applies to **stnode** and not its descendents.

(st-get-spy stnode) → *spy*

Returns the spy associated with **stnode**, or nil if no spy exists.

(st-spyp lisp-object) → *spy*

Returns **lisp-object** when it is a spy.

(st-kill-spy stnode) → *stnode*

Removes the spy associated with **stnode** if such a spy exists, and returns **stnode**.

(st-kill-spy-rec stnode) → *stnode*

Calls **st-kill-spy** on every descendent of **stnode** recursively and then on **stnode**.

(st-spy-action spy^{stnode} **port** [**arg**$_1$] ... [**arg**$_n$]) → *nil*

The default spy action. Prints the name of the port, a **?** if the message is on an input port or **!** if the message is on an output port, and then additional message arguments. Finally, calls the prin method on the stnode observed by **spy**.

Example:

```
? (st-declare spydemo (in) (out))
= #:stnode:spydemo

? (setq parent (st-create 'spydemo))
= spydemo:[net<0>]

? (setq s1 (st-create 'spydemo))
= spydemo:[net<0>]

? (st-include parent s1)
= spydemo:[net<1>]

? (setq myobj '(1 2 3))
= (1 2 3)

? (st-store-and-set-action s1 myobj '#:spydemo)
```

```
= spydemo:[atom<cons>]

? (de #:spydemo:in (node port myarg)
      (print "Value on """
             port
             """ is "
             myarg))
= #:spydemo:in


? ;What is the current spy action :
? ;The function st-spy-action
? *st-spy-action*
= st-spy-action

? (st-set-spy s1 ())
= ()

? (st-emit parent 'in 5)
in? (5) <- spydemo:[atom<cons>]
Value on "in" is 5
= ()

;;When we emit on the "in" port of s1,
;;the sibling does not receive the signal.
? (st-emit s1 'in 5)
Value on "in" is 5
= ()

? (st-emit s1 'out 5)
out! (5) <- spydemo:[atom<cons>]
= ()

? (st-kill-spy s1)
= spydemo:[atom<cons>]

? (st-emit parent 'in 5)
Value on "in" is 5
= ()
```

```
? (st-emit s1 'out 5)
= ()
```

Please consult appendix A for further information on spy implementation.

1.3 Appendix A : The Le-Lisp implementation of Sophtalk

An stnode type is characterized by the following:

- *Signature* : We declare a type as having certain *private* input and output ports. A type also inherits input and output ports from its supertypes. The sum of the inherited and private ports makes up the type's *signature*.
- *Semantics* : A type declaration also represents a certain port composition semantics. For example, the default semantics naturally establish the declared input ports as input ports and the declared output ports as output ports. Other semantics may be interesting, such as reversing input and output ports, adding ports to those declared, etc. Later on we will see examples of alternative semantics.

A declared stnode type thus inherits both port information and composition semantics from its supertypes. To designate a subtype **foo**, we must identify it in both hierarchies. In the Le-Lisp implementation of the system, only one path is used to refer to the two hierarchies. The path begins with the semantics path and ends with the port path. For example, if `foo` inherits its semantics from the `#:stnode:sub1:s2:s1` and its ports from `p1:p11`, the path referring to this type is `#:stnode:sub1:s2:s1:p1:p11:foo`. The unique path tells us nothing about where that semantics part ends and the port part begins.

In the functions defined in this reference manual, a **type** argument may either be a full path (semantics + signature) or just a signature path. If the semantics part of the path is left out, the subtype is assumed to inherit from the default root semantics type **#:stnode**. This is illustrated in figure 1.1.

1.3.1 The Le-Lisp type extension

We have created a module, independent of Sophtalk but imported by Sophtalk, that extends the Le-Lisp type system. The tools offered by this module allow one to manipulate type hierarchies. Each type hierarchy is distinguished by a *root type*. The part of the hierarchy below the root is called the *short type*. The concatentation of a root type and short type is called a *long type*. To build an isomorphic hierarchy from an existing one, it is sufficient to create a new root type and concatenate it with the existing short type.

The following functions are exported by the module **sophtalk/stnode/types**.

(make-subtype root-type basic-type) → *long-type*

Creates a long-type. If **basic-type** is a subtype of **root-type**, returns **basic-type**, otherwise returns the concatenation of **root-type** and **basic-type**.

(basic-type root-type long-type) → *basic-type*

Extracts a basic-type. If **long-type** may be written **root-type:basic-type**, returns **basic-type**, otherwise returns **long-type**.

(sendt method type) → *lisp*

Like `send`, but operates on types instead of objects. For an object O of type T, the call `(sendt 'a 'T)` behaves like `(send 'a (type-of O))`.

Example:

```
? (make-subtype '#:a:b:c  '#:d:e)
= #:a:b:c:d:e

? (basic-type '#:a:b:c '#:a:b:c:d:e)
= #:d:e

? (st-declare typetest (a b c) (d e f))
= #:stnode:typetest
```

1.3.2 Stnode types

Sophtalk uses the type extension package to break down an stnode type into two parts, a semantic part (a root-type) and a signature part (a basic-type). An **st-type** may either be a long-type or a basic-type whose root is the default root type (**#:stnode**).

An stnode type, declared with the function **st-declare**, is implemented as a Le-Lisp structure. Therefore, you may not define another structure bearing the same name. When you declare an stnode type, the following methods are automatically defined:

```
(send 'inputs <stnode>) -> <ports>
```

```
(send 'outputs <stnode>) -> <ports>
(send 'private-inputs <stnode>) -> <local-ports>
(send 'private-outputs <stnode>) -> <local-ports>
```

The following functions are exported by the module **sophtalk/stnode/defs**.

(st-type type) → *st-type*

Computes and returns the st-type associated with **type**. If **type** has the form of an st-type, returns **type**. If **type** is a basic-type, concatenates the default root type with **type** using **make-subtype**. The **type** need not have been declared previously.

(st-typep type) → *st-type*

Computes and returns the st-type associated with **type**.

(st-friend-type type root-type) → *st-type*

Declares a new type hierarchy under **root-type**. The hierarchy is isomorphic to the standard hierarchy (hence the term "friend"). The **type** (a basic or st-type) must have already been declared. The resulting type is the concatentation of **root-type** and the basic-type of **type**.

Example: A spy is an stnode whose input/output signature should exactly match that of the stnode it observes. The port semantics differ however —the spy duplicates the observed stnode's input and output ports, but adopts the union as input ports. To distinguish these semantics, we define a new spy root-type, called **#:stnode:spy**. To create a spy type from a standard type, we simply sever the standard basic-type, and graft it under the spy root-type.

Suppose we want to spy on an stnode whose root-type is `#:stnode` and whose basic-type is `#:a:b:c`, making the st-type `#:stnode:a:b:c`. The spy creation function calls `friend-type` with the new root-type `#:stnode:spy` and the existing basic type. The spy's st-type is thus `#:stnode:spy:a:b:c`. Note that the parent types `#:stnode:spy:a` and `#:stnode:spy:a:b` are declared automatically within the call to `friend-type`. It is during these declaration that all port names are regrouped as input ports for the spy's type.