

The Channel Ambient Language

© Andrew Phillips 2005

Version 0.081

Contents

1	Introduction	3
2	Runtime	3
2.1	Installation	3
2.2	Execution	3
2.3	Debugging	4
2.4	Networking	4
3	Programs	5
3.1	Syntax	5
3.2	Local Site	5
3.3	Network Site	5
3.4	Nested Site	5
3.5	State	6
3.6	Execution	6
4	Processes	7
4.1	Syntax	7
4.2	Null	7
4.3	Parallel Composition	7
4.4	Restriction	8
4.5	Agent	8
4.6	Action	8
4.7	Replicated Action	8
4.8	Pattern Matching	8
4.9	Definitions	9
5	Actions	9
5.1	Syntax	9
5.2	Sibling Output	10
5.3	Child Output	10
5.4	External Input	11
5.5	Parent Output	11
5.6	Local Output	12
5.7	Internal Input	13
5.8	Enter	13
5.9	Accept	14
5.10	Leave	14
5.11	Release	15

6	Patterns	16
6.1	Syntax	16
6.2	Substitution	16
7	Types	16
7.1	Syntax	16
7.2	Typechecking	16
8	Values	17
8.1	Syntax	17
8.2	Basic Values	17
8.3	Compound Values	17
8.4	Expressions	17
9	Lexical Syntax	18
9.1	Regular Expressions	18
9.2	Constants	18
9.3	Variables	19
9.4	Comments	19
10	Example	20
10.1	Program Code	20
10.2	Initial State	20
10.3	Execution	21
10.4	Final State	22
10.5	Further Examples	22
A	Language Summary	23

1 Introduction

This article presents the Channel Ambient Language (CAL), together with a corresponding runtime system. The Channel Ambient Language is a programming language for writing mobile applications, based on a formal model. The language uses the notion of an *ambient* to program the main components of a mobile application, including mobile agents and hardware sites. The applications are assumed to execute on networks that support the widely-used TCP/IP protocol version 4. The article is structured as follows:

- 2 Runtime** briefly describes the Channel Ambient Runtime, which is used to execute programs written in the Channel Ambient Language. This section also describes a number of debugging mechanisms for visualising the execution state of the runtime, and presents the basic structure of the networks in which programs are assumed to execute.
- 3 Programs** describes the syntax of Channel Ambient *Programs*, and shows how a given *Program* can be executed by the runtime. This section also describes the internal state of the runtime when it executes a given program.
- 4 Processes** describes the syntax of *Processes*, and shows how a given *Process* can be executed by the runtime.
- 5 Actions** describes the syntax of *Actions*, and shows how a given *Action* can be executed by the runtime.
- 6 Patterns** describes the syntax of *Patterns*, and shows how a given *Pattern* can be substituted with a *Value* inside a given *Process*.
- 7 Types** describes the syntax of *Types*, and shows how a given program can be checked for type errors.
- 8 Values** describes the syntax of *Values*.
- 9 Lexical Syntax** describes the lexical syntax of *Constants*, *Variables* and *Comments*.
- 10 Example** describes an example program written in the Channel Ambient Language, and shows how this program can be executed by the Channel Ambient Runtime.

2 Runtime

2.1 Installation

The Channel Ambient Runtime is installed as follows:

1. Place the executable file *cam.exe* in a suitable directory, preferably one that is in the system PATH.
2. Uncompress the examples to a suitable directory.

2.2 Execution

The Channel Ambient Runtime can be used to execute a file *source.ca* by typing the following command in a console:

```
cam.exe source.ca
```

Alternatively, a graphical file navigation tool (e.g. Windows Explorer) can be configured so that *cam.exe* is used by default to open all *.ca* files.

2.3 Debugging


Each source file should be executed in a separate directory containing the following files:


source.ca source file to be executed by the runtime


state.html current state of the runtime

start.html initial state of the runtime




log log of outputs made by the runtime

_thread.gif  icon to display the state of a parallel thread in the runtime

_open.gif  icon to display the state of an ambient in the runtime

_closed.gif  icon to hide the state of an ambient in the runtime

The execution state of the runtime is regularly streamed to the file *state.html*, which can be viewed in a web browser and periodically refreshed to display the latest state information. For reference, the initial state of the runtime is stored in the file *start.html*.

The state of the runtime looks very much like a source file, and contains any program code that is currently being executed by the runtime. Each currently executing thread is displayed next to a *thread* icon , and each currently executing ambient is enclosed in a box with an *open* folder icon  next to the ambient's name. The state of the ambient can be hidden by clicking on this icon, which collapses the contents of the ambient and displays a *closed* folder icon  next to the ambient's name. The state of the ambient can be revealed again by clicking on this icon.

Any statements that are printed on the console are also recorded in a *log* file.

2.4 Networking

Applications written in the Channel Ambient Language are assumed to execute on networks that support the widely-used TCP/IPv4 protocol. Based on the properties of TCP/IP networks, a distinction is made between two types of ambients: *sites* and *agents*. Sites represent physical machines that are assumed to be immobile, while agents represent software programs that can move in and out of sites and other agents. A site name is an *Address* of the form *IP:N*, which represents the IP address and port number of a machine on the network, while an agent name is a simple string.

In a flat network topology, all sites are assumed to execute in parallel with each other, and a given site can potentially communicate with any other site in the network. In a hierarchical network topology, however, sites can be logically contained inside other sites to form Local Area Networks. As a result, a given site is unable to communicate directly with another site that is not in the same LAN. A site can also act as a *gateway* between two networks: the local network it contains and the global network in which it is contained. As a result, a gateway site is usually assigned two network addresses, one for the local network and one for the global network. Note that the main address of a site is its global address. The local address is used merely as an implementation mechanism, which allows messages or agents from child sites to be correctly routed to the parent site.

An example of how sites can be used to model the topology of Local and Wide Area Networks is illustrated in Figure 1. In this example, each site executes on a separate machine, using port number 3000. The sites 192.168.0.2:3000 - 92.168.0.4:3000 are part of a Local Area Network inside a gateway site with local address 192.168.0.1:3000 and global address 82.35.60.43:3000. The ambients inside the LAN will send messages or agents to a default parent, which will be automatically routed to the local address of the gateway. The ambients outside the LAN will send messages or agents to the global address of the gateway. Thus, the local and global addresses allow the gateway to distinguish between local and global interactions.

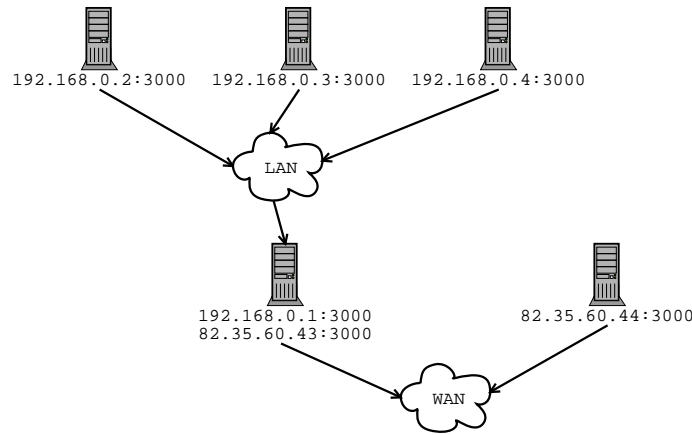


Figure 1: Network Topology

3 Programs

3.1 Syntax

The syntax of *Programs* is summarised below, where optional elements are enclosed in braces as *{Optional}*:

$Program ::=$	$Name$	$\{Address\}$	$[Process]$	Local Site		
		$Address$	$\{Address\}$	$[Process]$	Network Site	
		$Address$	$[Address$	$\{Address\}$	$[Process]]$	Nested Site

3.2 Local Site

$$Name_1 \{Address_1\} [Process_1]$$

Executes $Process_1$ inside a site with name $Name_1$ and local address $Address_1$. A site with local address $Address_1$ can logically contain a Local Area Network of child sites, which interact with the site using $Address_1$. If no $Address_1$ is specified then the site cannot logically contain any child sites.

3.3 Network Site

$$Address_2 \{Address_1\} [Process_1]$$

Executes $Process_1$ inside a site with local address $Address_1$ and global address $Address_2$. A site with global address $Address_2$ can interact with other sites on the global network using $Address_2$.

3.4 Nested Site

$$Address_3 [Address_2 \{Address_1\} [Process_1]]$$

Executes $Process_1$ inside a site with local address $Address_1$ and global address $Address_2$. The site also has a parent with local address $Address_3$.

3.5 State

When the runtime executes a given program it has an internal state that changes over time. The state of a runtime with a given *Address* and *Contents* is represented as:



The *Contents* of the runtime can be one of the following:

- A running *Process*:



- A blocked *Process*:



- A running *Ambient* with its own *Contents*:



- A blocked *Ambient* with its own *Contents*:



- Two *Contents* in parallel:

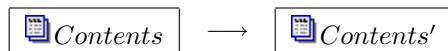


In addition, the following notation is used to represent an ambient that is either running or blocked:

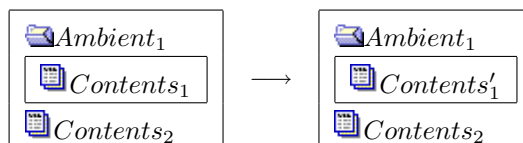


3.6 Execution

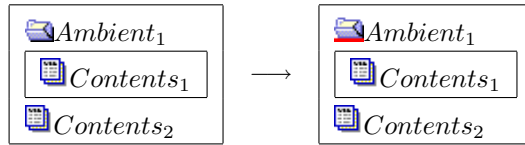
When the runtime executes a given program, its internal state is modified with each execution step. This transformation of state is described using rules of the form:



Note that if $Contents_1$ can evolve to $Contents'_1$ then this transformation can also take place inside $Ambient_1$:



If an $Ambient_1$ does not contain any unblocked actions or ambients then $Ambient_1$ is blocked:



If the entire contents of the runtime are blocked then execution is suspended until the runtime receives an interrupt from the network, announcing the arrival of new messages or ambients to be executed.

4 Processes

4.1 Syntax

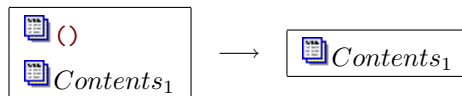
The syntax of *Processes* is summarised below, where optional elements are enclosed in braces as *{Optional}*:

$Process ::=$	$()$ $Process \mid Process$ $new \ Name:Type \ Process$ $Agent[Process]$ $Action\{; \ Process\}$ $!Action\{; \ Process\}$ $if \ Value\{-> \ Value\} \ then \ Process \ \{else \ Process\}$ $type \ Name = Type \ Process$ $let \ Pattern = Value \ in \ Process$ $let \ Macro(\{Pattern\}) = Process \ in \ Process$ $Macro\langle\{Value\}\rangle$ $(Process)$	Null Parallel Composition Restriction Agent Action Replicated Action Pattern Matching Type Definition Value Definition Macro Definition Macro Instantiation Nested Process
---------------	--	---

4.2 Null

$()$

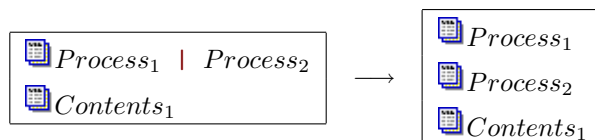
Represents the end of a process:



4.3 Parallel Composition

$Process_1 \mid Process_2$

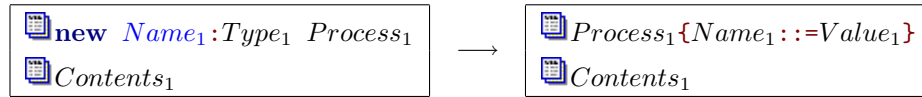
Creates two processes $Process_1$ and $Process_2$, which execute in parallel:



4.4 Restriction

new $Name_1:Type_1$ $Process_1$

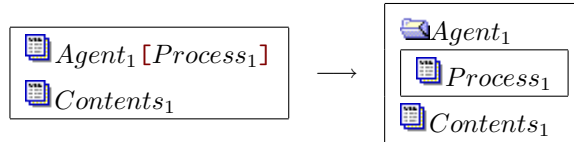
Creates a globally unique value $Value_1$ of type $Type_1$. This value is assigned to $Name_1$ in $Process_1$, which then executes:



4.5 Agent

$Agent_1[Process_1]$

Creates an agent with name $Agent_1$, which executes $Process_1$:



4.6 Action

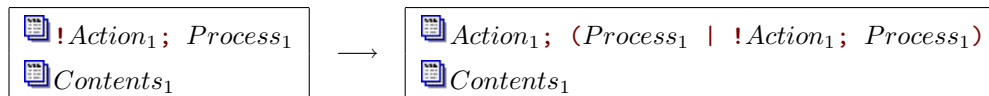
$Action_1\{; Process_1\}$

Tries to perform $Action_1$ and then execute $Process_1$. If no $Process_1$ is specified then the null process $()$ is used by default. The behaviour of the various actions is described in Section 5.

4.7 Replicated Action

$!Action_1\{; Process_1\}$

Tries to perform $Action_1$ and then execute $Process_1$ in parallel with the original replicated action process. This has the effect of repeatedly executing $Action_1$ followed by $Process_1$:

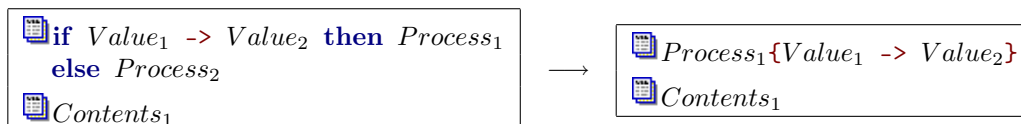


If no $Process_1$ is specified then the null process $()$ is used by default.

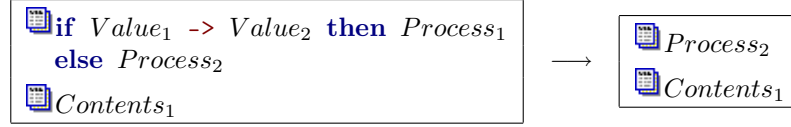
4.8 Pattern Matching

if $Value_1\{->Value_2\}$ **then** $Process_1\{$ **else** $Process_2\}$

Tries to match $Value_1$ with $Value_2$ and then execute $Process_1$. If there is a match then $Process_1$ executes, where each value in $Value_1$ is assigned to a corresponding variable in $Value_2$, written $Process_1\{Value_1 -> Value_2\}$:



This requires $Value_1$ and $Value_2$ to have compatible patterns. If there is no match then $Process_2$ executes:



If no $Process_2$ is specified then the null process $()$ is used by default. If no $Value_2$ is specified then the value **true** is used by default.

4.9 Definitions

The code of a Channel Ambient program can be simplified by defining $Value$, $Type$ and $Macro$ variables. These variables are substituted with their corresponding definitions by the runtime. The substitutions are performed during parsing, before any program code is executed.

type $Name_1 = Type_1 Process_1$

Short for $Process_1$, in which $Name_1$ is substituted with $Type_1$.

let $Pattern_1 = Value_1$ **in** $Process_1$

Short for $Process_1$, in which $Pattern_1$ is substituted with $Value_1$.

let $Macro_1(Pattern_1) = Process_1$ **in** $Process_2$

Short for $Process_2$, in which $Macro_1\langle Value_1 \rangle$ is substituted with $Process_1$. For each macro substitution, the parameter $Pattern_1$ is instantiated with $Value_1$ in $Process_1$.

5 Actions

5.1 Syntax

The syntax of $Actions$ is summarised below, where optional elements are enclosed in braces as $\{Optional\}$:

$Action ::=$	$Ambient.Channel\langle\{Value\}\rangle$	Sibling Output
	$Ambient/Channel\langle\{Value\}\rangle$	Child Output
	$Channel^\wedge(\{Pattern\})$	External Input
	$Channel^\wedge\langle\{Value\}\rangle$	Parent Output
	$Channel\langle\{Value\}\rangle$	Local Output
	$Channel(\{Pattern\})$	Internal Input
	in $Ambient.Channel$	Enter
	-in $Channel$	Accept
	out $Channel$	Leave
	-out $Channel$	Release

By definition, an $Ambient$ can be either a mobile $Agent$ or a remote $Site$. Based on the properties of TCP/IP networks, only a subset of actions are allowed inside a given $Site$:

$Action_{Site} ::=$	$Site.Channel\langle\{Value\}\rangle$	Site Output
	$Ambient/Channel\langle\{Value\}\rangle$	Child Output
	$Channel^\wedge(\{Pattern\})$	External Input
	$System^\wedge\langle\{Value\}\rangle$	System Output
	$Channel\langle\{Value\}\rangle$	Local Output
	$Channel(\{Pattern\})$	Internal Input
	-in $Channel$	Accept
	-out $Channel$	Release

Sites are constrained so that they cannot contain a sibling output to an agent. This reflects the assumption that agents do not have a network address, and therefore cannot be reached directly by sites over a network. In addition, sites are constrained so that they can only contain a parent output on a limited set of *System* channels. This simplifies the synchronisation between nested sites in a network. Finally, sites are constrained so that they cannot contain an enter or a leave. This reflects the assumption that sites are immobile.

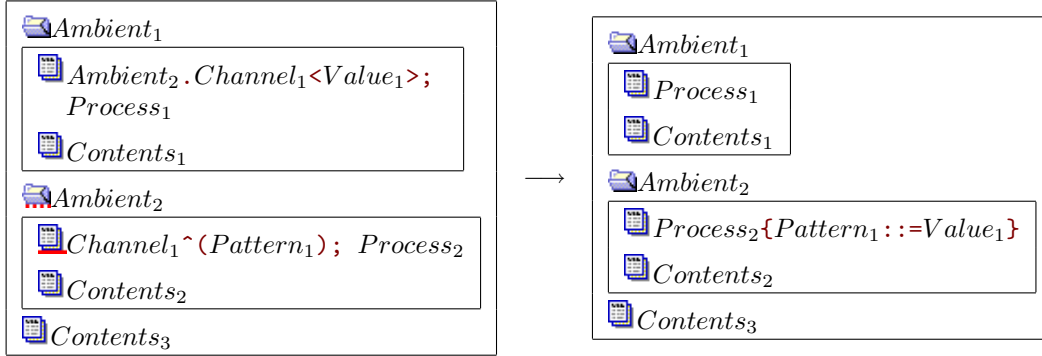
5.2 Sibling Output

$$Ambient_2.Channel_1\langle Value_1\rangle; Process_1$$

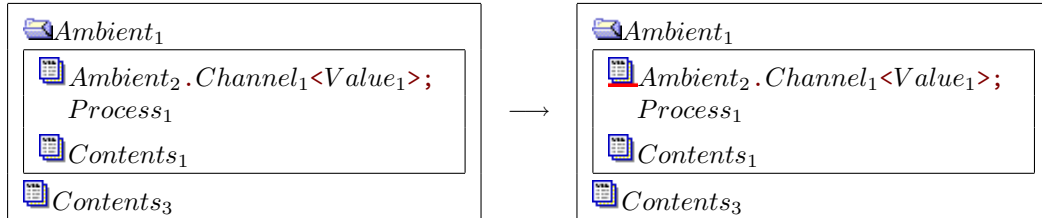
Tries to send $Value_1$ on $Channel_1$ to a sibling $Ambient_2$, and then execute $Process_1$. If this process executes inside $Ambient_1$, and there is a sibling $Ambient_2$ with a blocked External Input process

$$Channel_1^{\wedge}(Pattern_1); Process_2$$

then $Value_1$ is sent along $Channel_1$ and assigned to $Pattern_1$ in $Process_2$. If $Ambient_2$ is a remote *Site* then $Value_1$ is sent over the network using TCP/IP. Afterwards, $Ambient_1$ executes in parallel with $Ambient_2$, $Process_1$ executes inside $Ambient_1$ and $Process_2$ executes inside $Ambient_2$:



If there is no sibling $Ambient_2$ with a corresponding blocked External Input process, and $Ambient_2$ is a mobile *Agent*, then the Sibling Output process is blocked:



If $Ambient_2$ is a remote *Site* then the Sibling Output process remains active and is re-attempted after a specified delay. The delay is increased exponentially after each attempt, in order to limit consumption of network bandwidth.

5.3 Child Output

$$Ambient_1/Channel_1\langle Value_1\rangle$$

Tries to send $Value_1$ on $Channel_1$ to child $Ambient_1$. This is short for a corresponding sibling output process inside a private $Ambient_2$:

$$Ambient_2[Ambient_1.Channel_1\langle Value_1\rangle]$$

Once the output has been performed the resulting empty $Ambient_2$ can be garbage-collected immediately.

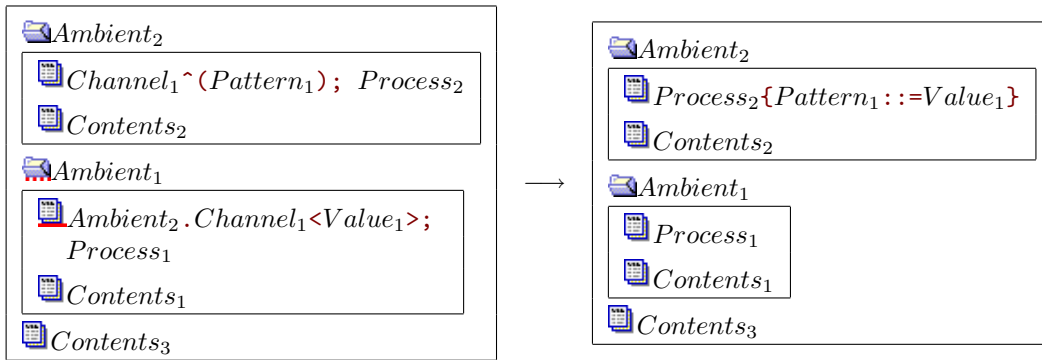
5.4 External Input

$$Channel_1 \hat{~} (Pattern_1); Process_2$$

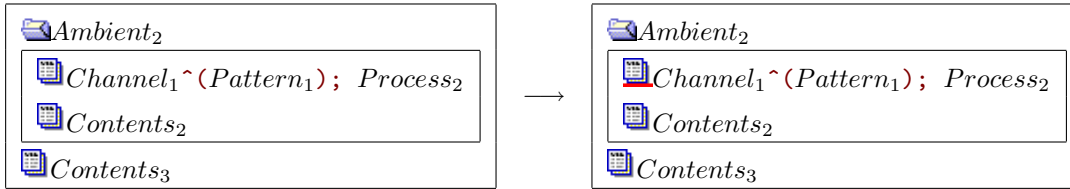
Tries to receive a value on $Channel_1$ from a sibling ambient, and then assign it to $Pattern_1$ in $Process_2$. If this process executes inside $Ambient_2$, and there is a sibling $Ambient_1$ with a blocked Sibling Output process

$$Ambient_2.Channel_1 \langle Value_1 \rangle; Process_1$$

then $Value_1$ is received along $Channel_1$ and assigned to $Pattern_1$ in $Process_2$. Afterwards, $Ambient_2$ executes in parallel with $Ambient_1$, $Process_2$ executes inside $Ambient_2$ and $Process_1$ executes inside $Ambient_1$:



If there is no sibling $Ambient_1$ with a corresponding blocked Sibling Output process then the External Input process is blocked:



Note that external inputs inside a site will block systematically, since corresponding sibling outputs to a site are never blocked.

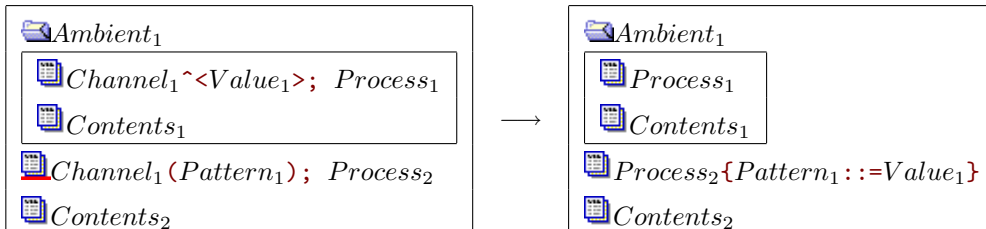
5.5 Parent Output

$$Channel_1 \hat{~} \langle Value_1 \rangle; Process_1$$

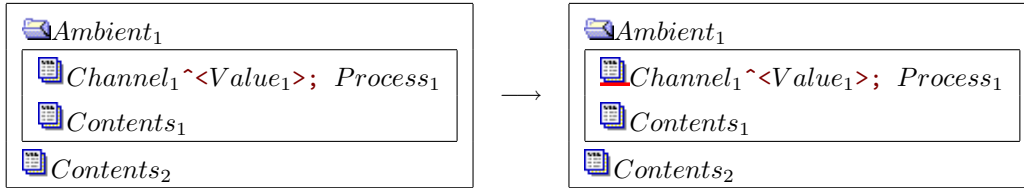
Tries to send $Value_1$ on $Channel_1$ to the parent ambient, and then execute $Process_1$. If this process executes inside $Ambient_1$, and in parallel there is a blocked Internal Input process

$$Channel_1 (Pattern_1); Process_2$$

then $Value_1$ is sent along $Channel_1$ and assigned to $Pattern_1$ in $Process_2$. Afterwards, $Process_1$ executes inside $Ambient_1$, and $Process_2$ executes in parallel with $Ambient_1$:



If there is no parallel blocked Internal Input process then the Parent Output process is blocked:



In order to simplify the interaction between nested sites in a network, parent outputs inside a site are only allowed on a limited set of *System* channels. These parent outputs correspond to system calls to the parent site. If a parent output on a *System* channel is performed inside a nested agent then it is treated as an ordinary parent output. However, a site can still forward messages from nested agents to its parent using forwarding channels. A separate forwarding channel can be defined for each agent or group of agents, allowing fine-grained access permissions to be implemented for each system call.

Conceptually, the parent site is assumed to contain a number of replicated input processes, each of which provides a given service:

```
print(s:string)
```

Prints the string *s* on the console of the runtime.

```
println(s:string)
```

Prints the string *s* followed by a newline character on the console of the runtime.

```
delay(duration:int,ack:<void>)
```

Sends a void message to the runtime on the *ack* channel after a delay of *duration* seconds.

```
run(program:string,arguments:string)
```

Runs the given executable *program* with the given string *arguments*. The program must be installed on the system in order to be executed.

```
read(file:string,result:<string>)
```

Reads the given *file* and sends its string contents to the runtime on the *result* channel.

```
write(file:string,contents:string)
```

Writes the string *contents* to the given *file*. If no such file exists, then a new file is created.

```
route(s:site,x:<'a>,v:<sub 'a>)
```

Sends the value *v* on channel *x* to site *s*. This allows a site to communicate asynchronously with a sibling via its parent site, as an alternative to using the direct sibling output process.

```
parent(x:<'a>,v:<sub 'a>)
```

Sends the value *v* on channel *x* inside the parent site. This allows a site to communicate asynchronously with its parent site.

5.6 Local Output

```
Channel1<Value1>
```

Tries to send *Value1* on *Channel1* locally. This is short for a corresponding parent output process inside a private *Ambient1*.

```
Ambient1[Channel1^<Value1>]
```

Once the output has been performed the resulting empty *Ambient1* can be garbage-collected immediately.

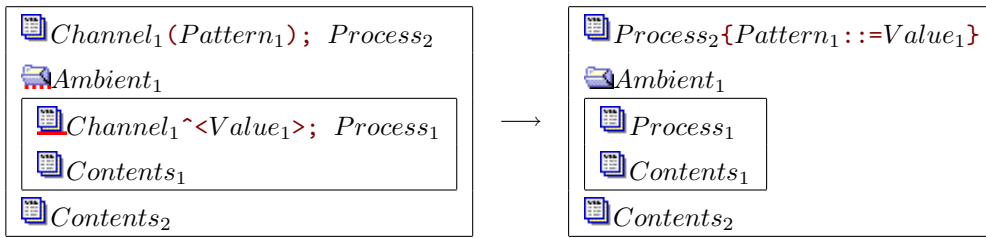
5.7 Internal Input

$$Channel_1(Pattern_1); Process_2$$

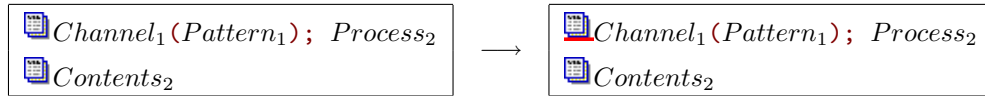
Tries receive a value on $Channel_1$ from a child ambient, and then assign it to $Pattern_1$ in $Process_2$. If there is a child $Ambient_1$ with a blocked Parent Output process

$$Channel_1^{\wedge}\langle Value_1 \rangle; Process_1$$

then $Value_1$ is received along $Channel_1$ and assigned to $Pattern_1$ in $Process_2$. Afterwards, $Process_2$ executes in parallel with $Ambient_1$ and $Process_1$ executes inside $Ambient_1$:



If there is no child $Ambient_1$ with a corresponding blocked Parent Output process then the Internal Input process is blocked:



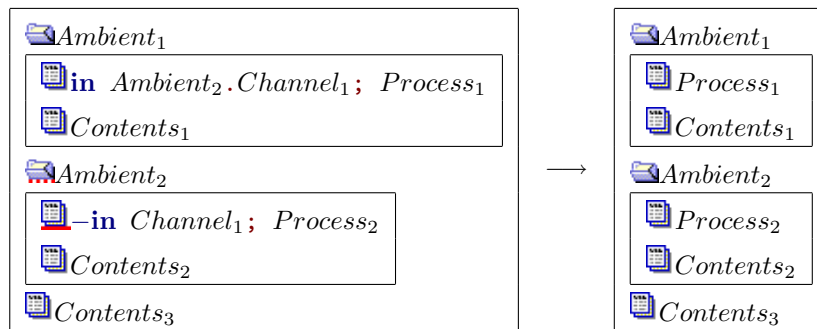
5.8 Enter

$$\mathbf{in} \text{ Ambient}_2.Channel_1; Process_1$$

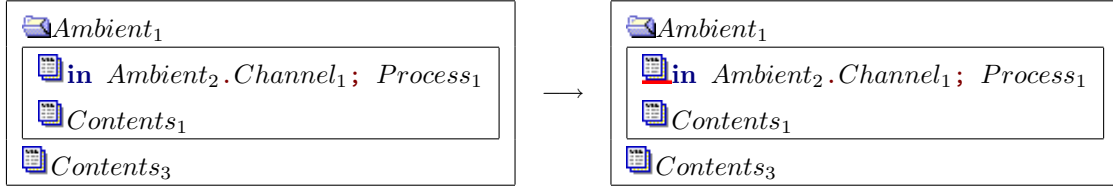
Tries to move the enclosing ambient on $Channel_1$ inside a sibling $Ambient_2$, and then execute $Process_1$. If this process executes inside $Ambient_1$, and there is a sibling $Ambient_2$ with a blocked Accept process

$$-\mathbf{in} \text{ Channel}_1; Process_2$$

then $Ambient_1$ enters $Ambient_2$ on $Channel_1$. If $Ambient_2$ is a remote *Site* then $Ambient_1$ is sent over the network using TCP/IP. Afterwards, $Ambient_1$ executes inside $Ambient_2$, $Process_1$ executes inside $Ambient_1$ and $Process_2$ executes inside $Ambient_2$:



If there is no sibling $Ambient_2$ with a corresponding blocked Accept process, and $Ambient_2$ is a mobile *Agent*, then the Enter process is blocked:



If $Ambient_2$ is a remote *Site* then the Enter process remains active and is re-attempted after a specified delay. The delay is increased exponentially after each attempt, in order to limit consumption of network bandwidth.

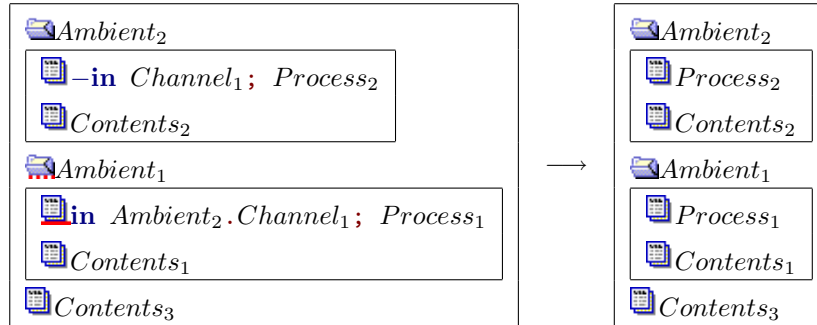
5.9 Accept

$\text{-in Channel}_1; Process_2$

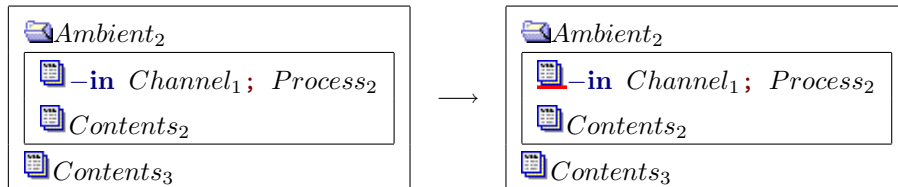
Tries to allow a sibling ambient to enter on $Channel_1$, and then execute $Process_1$. If this process executes inside $Ambient_2$, and there is a sibling $Ambient_1$ with a blocked Enter process

$\text{in Ambient}_2.Channel_1; Process_1$

then $Ambient_1$ enters $Ambient_2$ on $Channel_1$. Afterwards, $Ambient_1$ executes inside $Ambient_2$, $Process_2$ executes inside $Ambient_2$ and $Process_1$ executes inside $Ambient_1$:



If there is no sibling $Ambient_1$ with a corresponding blocked Enter process then the Accept process is blocked:



Note that accepts inside a site will block systematically, since corresponding enters to a site are never blocked.

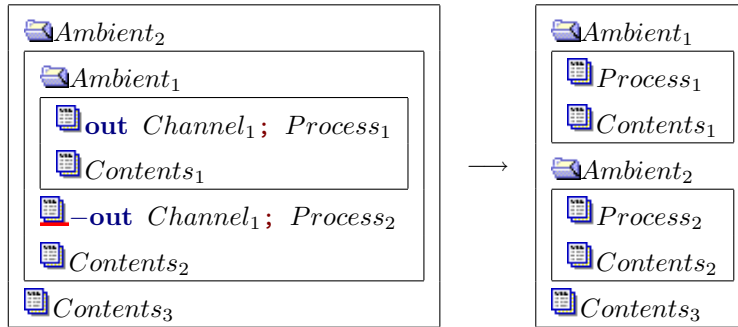
5.10 Leave

$\text{out Channel}_1; Process_1$

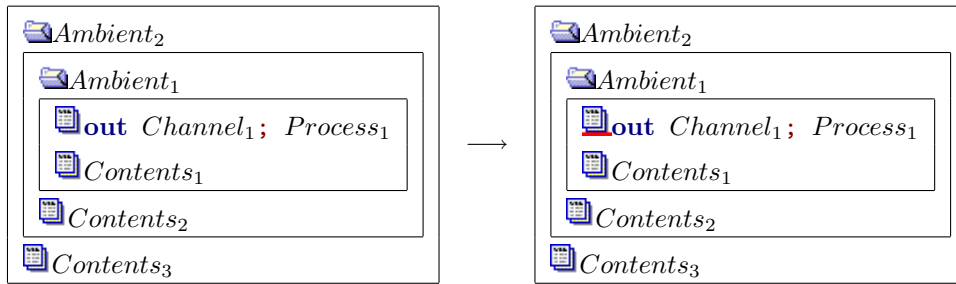
Tries to leave the parent ambient on $Channel_1$ and then execute $Process_1$. If this process executes inside $Ambient_1$ and there is a parent $Ambient_2$ with a blocked Release process

$\text{-out Channel}_1; Process_2$

then $Ambient_1$ can leave its parent $Ambient_2$. Afterwards, $Ambient_1$ executes in parallel with $Ambient_2$, $Process_1$ executes inside $Ambient_1$ and $Process_2$ executes inside $Ambient_2$:



If there is no parent ambient with a corresponding blocked Release process then the leave process is blocked:



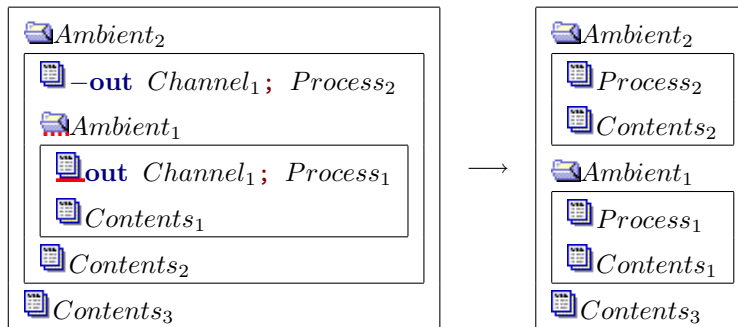
5.11 Release

-out Channel₁; Process₂

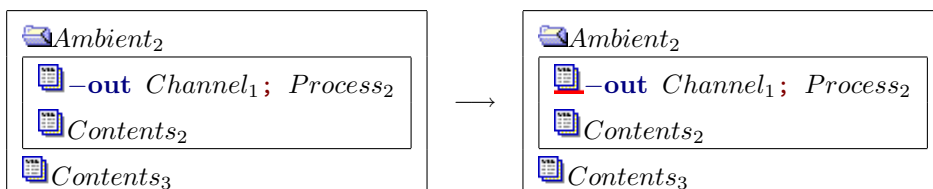
Tries to release a child ambient on *Channel₁* and then execute *Process₂*. If this process executes inside *Ambient₂* and there is a child *Ambient₁* with a blocked Leave process

out Channel₁; Process₁

then *Ambient₂* can release *Ambient₁*. Afterwards, *Ambient₂* executes in parallel with *Ambient₁*, *Process₂* executes inside *Ambient₂* and *Process₁* executes inside *Ambient₁*:



If there is no child *Ambient₁* with a corresponding blocked Leave process then the Release process is blocked:



6 Patterns

6.1 Syntax

$Pattern ::=$	$-$	Wildcard Pattern
	$()$	Void Pattern
	$Name\{ :Type\}$	Name Pattern
	$Pattern, Pattern$	Tuple Pattern
	$(Pattern)$	Nested Pattern

6.2 Substitution

A *Pattern* can be substituted with a *Value* inside a given *Process*, written

$$Process_1\{Pattern_1 := Value_1\}$$

The definition of substitution is standard, given that names are bound by Restriction, Input and Macro Definition processes. Each unbound name in $Pattern_1$ is substituted with the corresponding value in $Value_1$, inside $Process_1$.

7 Types

7.1 Syntax

$Type ::=$	<code>void</code>	Void Type
	<code>string</code>	String Type
	<code>int</code>	Integer Type
	<code>float</code>	Float Type
	<code>char</code>	Character Type
	<code>bool</code>	Boolean Type
	<code>site</code>	Site Type
	<code>agent</code>	Agent Type
	<code>migrate</code>	Migration Type
	$\langle\{Type\}\rangle$	Channel Type
	$Name$	Type Variable
	$'Name$	Polymorphic Type
	<code>sub Type</code>	Sub Type
	$Type, Type$	Tuple Type
	$Upper$	Type Constant
	$Upper Type$	Type Constructor
	$Type \mid Type$	Data Type
	$Type \text{ list}$	List Type
	<code>rec Name.Type</code>	Recursive Type
	$(Type)$	Nested Type

7.2 Typechecking

Before the runtime executes a given program, it checks to see whether the program is well-typed. The type-system for The Channel Ambient Language is based on the system for the Pict programming language, available from <http://www.cis.upenn.edu/~bcpierce/papers/pict/>

8 Values

8.1 Syntax

<i>Value</i> ::=	()	Void Value
	<i>String</i>	String Value
	<i>Integer</i>	Integer Value
	<i>Float</i>	Float Value
	<i>Character</i>	Character Value
	true	Boolean True
	false	Boolean False
	<i>Address</i>	Site Value
	<i>Name</i> : agent	Agent Value
	<i>Name</i> :<{ <i>Type</i> }>	Channel Value
	<i>Value</i> , <i>Value</i>	Tuple Value
	<i>Upper</i>	Data Constant
	<i>Upper Value</i>	Data Constructor
	[]	Empty List
	<i>Value</i> : <i>Value</i>	Value List
	<i>Name</i>	Variable
	<i>Value</i> + <i>Value</i>	Addition
	<i>Value</i> - <i>Value</i>	Subtraction
	<i>Value</i> * <i>Value</i>	Multiplication
	<i>Value</i> / <i>Value</i>	Division
	<i>Value</i> = <i>Value</i>	Equal
	<i>Value</i> <> <i>Value</i>	Different
	<i>Value</i> < <i>Value</i>	Less Than
	<i>Value</i> > <i>Value</i>	Greater Than
	<i>Value</i> <= <i>Value</i>	Less Than or Equal
	<i>Value</i> >= <i>Value</i>	Greater Than or Equal
	- <i>Value</i>	Negation
	show <i>Value</i>	String Representation
	(<i>Value</i>)	Nested Value

8.2 Basic Values

A *Value* can be a *String*, *Integer*, *Float*, or *Character* value, the void value (), boolean **true** or boolean **false**. In addition, a *Value* can be a Channel, Agent, or Site. A Channel value consists of a *Name* followed by a colon and the *Type* of values that the channel is capable of sending or receiving, enclosed in angle brackets. An agent value consists of a *Name* followed by a colon and the keyword **agent**. A site value consists of an IP *Address* followed by an *Integer* port number, enclosed in parentheses.

8.3 Compound Values

A *Value* can also be a tuple of values, where parentheses can be used to define nested tuples. In addition, a value can be a data constant, which is simply an *Upper* case name, or a data constructor, which is an *Upper* case name followed by a value. A value can also be a list, which can either be empty [] or of the form *Value*₁:*Value*₂, where *Value*₁ is the first element of the list and *Value*₂ is the remainder of the list. Note that all values in a list must be of the same type.

8.4 Expressions

A value can also be an expression, which is automatically evaluated by the runtime. The most basic expression is a variable *Name*, which can be substituted with a *Value* during program

execution. An expression can also consist of a prefix operator followed by a *Value* argument or an infix operator between two *Value* arguments.

The prefix operator `show` $Value_1$ converts $Value_1$ to a string value. By definition, every value has a string representation, although for certain values this may simply be the empty string. The prefix operator `-` $Value$ is defined in Table 1.

Infix operators take two arguments of any type, provided both types are the same. The comparison operators (`=`, `<`, `>`, `>=`, `<=`) return a result of boolean type. They rely on an ordering to compare both arguments. The arithmetic operators (`+`, `-`, `*`, `/`) return a result of the same type as their arguments. Table 1 describes the behaviour of the operators for each corresponding type of arguments. The `_` symbol means that the behaviour of the operator is unspecified, although the result will always be of the correct type.

Type	+	-	*	/	- (prefix)	=, <, >, >=, <=
String	Concatenate	—	—	—	—	Lexicographic Order
Integer	Add	Subtract	Multiply	Divide	Minus	Integer Order
Float	Add	Subtract	Multiply	Divide	Minus	Float Order
Character	—	—	—	—	—	ASCII Code Order
Boolean	Or	—	And	—	Not	Lexicographic Order
List	Append	—	—	—	—	Order of Elements
Data	—	—	—	—	—	Lexicographic Order
Other	—	—	—	—	—	—

Table 1: Operator Definitions

9 Lexical Syntax

9.1 Regular Expressions

Regular Expressions (*Regex*) are used to describe the syntax of Constants and Variables in the Channel Ambient Language:

<i>Regex</i> ::= <i>c</i>	Character
<i>c</i> · · <i>c</i>	Character Range
\neg <i>c</i>	Character Complement
<i>Regex</i> <i>Regex</i>	Concatenation of Expressions
<i>Regex</i> <i>Regex</i>	Alternative Expressions
<i>Regex</i> [?]	Optional Expression
<i>Regex</i> [*]	Repetition of Expression
<i>Regex</i> ⁺	Strict Repetition of Expression
(<i>Regex</i>)	Nested Expression

9.2 Constants

An *Integer* constant consists of an optional negative sign followed by one or more digits:

$$Integer ::= (-)^?(0 \cdot \cdot 9)^+$$

A *String* constant consists of a sequence of zero or more characters enclosed in double quotes. The sequence can only contain a double quote if it is preceded by a backslash:

$$String ::= "(\neg | \backslash | \")^*"$$

A *Float* constant consists of an *Integer*, followed by a decimal point and one or more digits, followed by an optional exponent. The exponent consists of *e* or *E*, followed by `+` or `-`, followed

by one or more digits:

$$Float ::= Integer.(0\cdots9)^+((e|E)(+|-)(0\cdots9)^+)?$$

An *IP* address constant consists of a sequence of four numbers between 0 and 255, separated by a decimal point. The IP address 0.0.0.0 is short for the default IP address of the host machine:

$$IP ::= (0\cdots9)^+. (0\cdots9)^+. (0\cdots9)^+. (0\cdots9)^+$$

A *Character* constant consists of any character enclosed in single quotes, apart from the single quote character. It can also consist of a backslash, followed by a special *escaped* character or a three-digit decimal number, enclosed in single quotes:

<i>Character</i> ::=	'(^)'	Regular Character
	'\"'	Single Quote
	'\\'	Backslash
	'\n'	Linefeed
	'\r'	Carriage Return
	'\t'	Horizontal Tabulation
	'\b'	Backspace
	'\ (0\cdots9)(0\cdots9)(0\cdots9)'	ASCII Character Code

9.3 Variables

A *Name* variable consists of a letter followed by zero or more letters, digits, underscores or single quotes:

$$Name ::= (a\cdots z)(A\cdots Z | a\cdots z | 0\cdots9 | _ | ')^*$$

An *Upper* case variable consists of an upper case letter followed by zero or more letters, digits, underscores or single quotes:

$$Upper ::= (A\cdots Z)(A\cdots Z | a\cdots z | 0\cdots9 | _ | ')^*$$

Channel, *Agent* and *Site* variables are *Names* representing *Channel* values, *Agent* values and *Site* values, respectively. An *Ambient* variable is a *Name* representing either an *Agent* value or a *Site* value. A *Macro* variable is an *Upper* case variable representing a macro:

$$\begin{aligned} Channel &::= Name \\ Agent &::= Name \\ Site &::= Name \\ Ambient &::= Name \\ Macro &::= Upper \end{aligned}$$

The following variables are reserved keywords of the language:

agent	bool	char	else	float	false	if	in
int	let	list	migrate	new	out	rec	show
site	string	sub	then	true	type	void	

9.4 Comments

A comment starts with the sequence of characters (*** and ends with the sequence of characters ***). Comments can be nested, but they cannot occur inside single or double quotes.

```

192.168.0.3:3000
[ let server = 192.168.0.2:3001 in
  let client = 192.168.0.3:3000 in
  let register =
    register:<site,<<migrate>>> in
  let logout = logout:<migrate> in
  let Monitor() = print<'monitor'> in
  let Continue() = print<'continue'> in
  ( new ack:<<migrate>>
    server.register<client,ack>;
    ack^(x);
    monitor
    [ out logout;
      in server.x;
      Monitor()
    ]
  | -out logout;
    Continue()
  )
]

192.168.0.2:3001
[ let Report() = print<'report'> in
  let register =
    register:<site,<<migrate>>> in
  !register^(c,k);
  new login:<migrate>
  c.k<login>;
  -in login;
  Report()
]

```

Figure 2: Program Code for Client and Server

10 Example

10.1 Program Code

The Channel Ambient language can be used to program a *client* device, which sends a mobile agent over a network to monitor resources on a remote *server* device.

In this example the server runs on port 3001 of IP address 192.168.0.2 and continually listens on the *register* channel for a client name *c* and an acknowledgement channel *k*. Each time a registration request is received, the server creates a new *login* channel. The server tries to send the login channel to the client on the acknowledgement channel, accept an agent on the login channel and then execute the *Report()* process. The *Report()* process forwards data about the resource to the agent on the server. The code for the server is given in Figure 2.

The client runs on port 3000 of IP address 192.168.0.3 and tries to send its name and an acknowledgement channel *ack* to the server on the *register* channel. After sending the registration request, the client waits for a login channel *x* on the acknowledgement channel. After receiving the login channel, the client creates a new *monitor* agent, which tries to leave on the *logout* channel, enter the server on the login channel and then execute the *Monitor()* process. The *Monitor()* process monitors data about the resource on the server. In parallel, the client tries to release an agent on the logout channel and then execute the *Continue()* process. The code for the client is given in Figure 2.

In this example the *Report()*, *Monitor()* and *Continue()* processes are defined as simple outputs. In general, however, they can be defined as arbitrarily complex processes.

10.2 Initial State

The two programs are executed on the client and server devices by separate runtimes. Initially, each runtime parses the program code and substitutes any value, type or process definitions:

192.168.0.3:3000

```
( new ack:<migrate>
  192.168.0.2:3001.register<192.168.0.3:3000,ack>;
  ack^(x);
  monitor
  [ out logout;
    in 192.168.0.2:3001.x;
    print<"monitor">
  ]
  | -out logout;
    print<"continue">
)
```

192.168.0.2:3001

```
!register^(c,k);
  new login:<migrate>
  c.k<login>;
  -in login;
  print<"report">
```

The remainder of this section describes how the internal state of the client and server runtimes can evolve during program execution.

10.3 Execution

The server blocks a replicated external input on the *register* channel, waiting to receive a value on this channel. In parallel, the client creates a new acknowledgement channel, where ack_1 is an abbreviation for a private channel name. All occurrences of the name *ack* are substituted with the private channel name ack_1 . The client then blocks a release on the *logout* channel, waiting to release an agent on this channel.

192.168.0.3:3000

```
192.168.0.2:3001.register<192.168.0.3:3000,ack1>;
  ack1^(x);
  monitor
  [ out logout;
    in 192.168.0.2:3001.x;
    print<"monitor">
  ]
  | -out logout;
    print<"continue">
```

192.168.0.2:3001

```
!register^(c,k);
  new login:<migrate>
  c.k<login>;
  -in login;
  print<"report">
```

The client sends its name and an acknowledgement channel to the server on the *register* channel. The client then blocks an external input on the acknowledgement channel, waiting to receive a value on this channel.

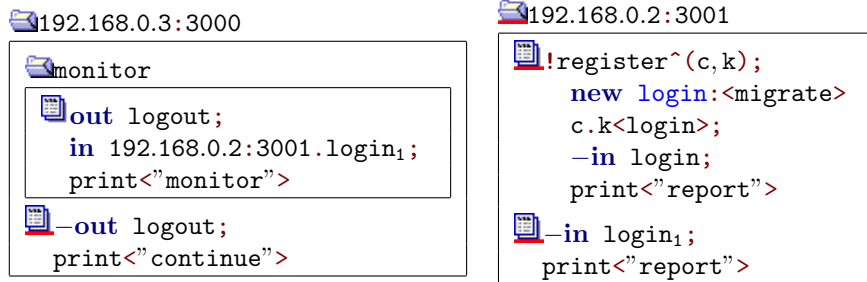
192.168.0.3:3000

```
ack1^(x);
  monitor
  [ out logout;
    in 192.168.0.2:3001.x;
    print<"monitor">
  ]
  | -out logout;
    print<"continue">
```

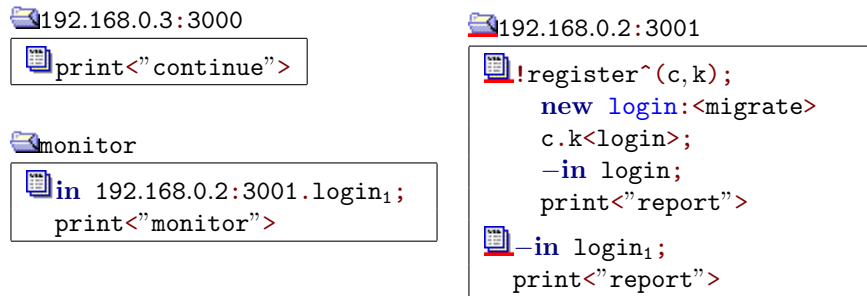
192.168.0.2:3001

```
!register^(c,k);
  new login:<migrate>
  c.k<login>;
  -in login;
  print<"report">
192.168.0.3:3000.ack1<login1>;
  -in login1;
  print<"report">
```

After creating a globally unique $login_1$ channel, the server sends this channel to the client on the acknowledgement channel. The server then blocks an accept on the login channel, waiting to accept an agent on this channel.

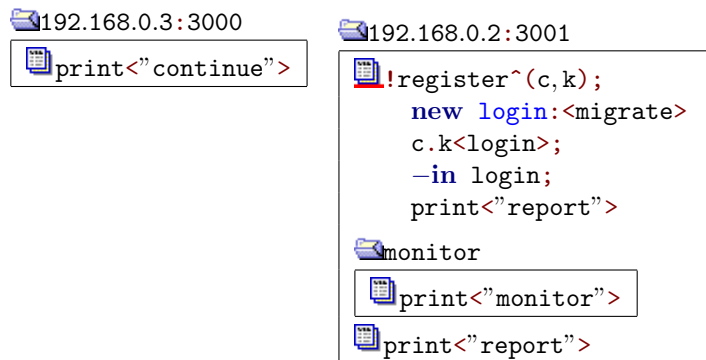


After the client creates a *monitor* agent, the agent leaves the client on the logout channel, and the client continues executing.



10.4 Final State

The monitor agent enters the server on the login channel and then executes a process to monitor a resource on the server. In parallel, the server executes a process to forward information about the resource to the monitor.



10.5 Further Examples

Further examples can be downloaded from <http://www.doc.ic.ac.uk/~anp/Examples/>, along with debugging information about the initial and final states of the runtime.

A Language Summary

The Channel Ambient Language is summarised below, where optional elements are enclosed in braces as *{Optional}*:

<i>Program</i>	<code>::= Name {Address} [Process]</code>	Local Site
	<code> Address {Address} [Process]</code>	Network Site
	<code> Address [Address {Address} [Process]]</code>	Nested Site
<i>Process</i>	<code>::= ()</code>	Null
	<code> Process Process</code>	Parallel Composition
	<code> new Name : Type Process</code>	Restriction
	<code> Agent [Process]</code>	Agent
	<code> Action { ; Process }</code>	Action
	<code> !Action { ; Process }</code>	Replicated Action
	<code> if Value { -> Value } then Process { else Process }</code>	Pattern Matching
	<code> type Name = Type Process</code>	Type Definition
	<code> let Pattern = Value in Process</code>	Value Definition
	<code> let Macro ({Pattern}) = Process in Process</code>	Macro Definition
	<code> Macro < {Value} ></code>	Macro Instantiation
	<code> (Process)</code>	Nested Process
<i>Action</i>	<code>::= Ambient . Channel < {Value} ></code>	Sibling Output
	<code> Ambient / Channel < {Value} ></code>	Child Output
	<code> Channel ^ ({Pattern})</code>	External Input
	<code> Channel ^ < {Value} ></code>	Parent Output
	<code> Channel < {Value} ></code>	Local Output
	<code> Channel ({Pattern})</code>	Internal Input
	<code> in Ambient . Channel</code>	Enter
	<code> -in Channel</code>	Accept
	<code> out Channel</code>	Leave
	<code> -out Channel</code>	Release
<i>Action_{Site}</i>	<code>::= Site . Channel < {Value} ></code>	Site Output
	<code> Ambient / Channel < {Value} ></code>	Child Output
	<code> Channel ^ ({Pattern})</code>	External Input
	<code> System ^ < {Value} ></code>	System Output
	<code> Channel < {Value} ></code>	Local Output
	<code> Channel ({Pattern})</code>	Internal Input
	<code> -in Channel</code>	Accept
	<code> -out Channel</code>	Release

<i>Pattern</i>	::=	-	Wildcard Pattern
		()	Void Pattern
		<i>Name</i> {: <i>Type</i> }	Name Pattern
		<i>Pattern</i> , <i>Pattern</i>	Tuple Pattern
		(<i>Pattern</i>)	Nested Pattern
<i>Type</i>	::=	void	Void Type
		string	String Type
		int	Integer Type
		float	Float Type
		char	Character Type
		bool	Boolean Type
		site	Site Type
		agent	Agent Type
		migrate	Migration Type
		< { <i>Type</i> }	Channel Type
		<i>Name</i>	Type Variable
		' <i>Name</i>	Polymorphic Type
		sub <i>Type</i>	Sub Type
		<i>Type</i> , <i>Type</i>	Tuple Type
		<i>Upper</i>	Type Constant
		<i>Upper</i> <i>Type</i>	Type Constructor
		<i>Type</i> <i>Type</i>	Data Type
		<i>Type</i> list	List Type
		rec <i>Name</i> . <i>Type</i>	Recursive Type
		(<i>Type</i>)	Nested Type
<i>Value</i>	::=	()	Void Value
		<i>String</i>	String Value
		<i>Integer</i>	Integer Value
		<i>Float</i>	Ffloat Value
		<i>Character</i>	Character Value
		true	Boolean True
		false	Boolean False
		<i>Address</i>	Site Value
		<i>Name</i> : agent	Agent Value
		<i>Name</i> : < { <i>Type</i> }	Channel Value
		<i>Value</i> , <i>Value</i>	Tuple Value
		<i>Upper</i>	Data Constant
		<i>Upper</i> <i>Value</i>	Data Constructor
		[]	Empty List
		<i>Value</i> :: <i>Value</i>	Value List
		<i>Name</i>	Variable
		<i>Value</i> + <i>Value</i>	Addition
		<i>Value</i> - <i>Value</i>	Subtraction
		<i>Value</i> * <i>Value</i>	Multiplication
		<i>Value</i> / <i>Value</i>	Division
		<i>Value</i> = <i>Value</i>	Equal
		<i>Value</i> <> <i>Value</i>	Different
		<i>Value</i> < <i>Value</i>	Less Than
		<i>Value</i> > <i>Value</i>	Greater Than
		<i>Value</i> <= <i>Value</i>	Less Than or Equal
		<i>Value</i> >= <i>Value</i>	Greater Than or Equal
		- <i>Value</i>	Negation
		show <i>Value</i>	String Representation
		(<i>Value</i>)	Nested Value

<i>Regexp</i>	::=	c	Character
		c..c	Character Range
		¬c	Character Complement
		<i>Regexp</i> <i>Regexp</i>	Concatenation of Expressions
		<i>Regexp</i> <i>Regexp</i>	Alternative Expressions
		<i>Regexp</i> [?]	Optional Expression
		<i>Regexp</i> [*]	Repetition of Expression
		<i>Regexp</i> ⁺	Strict Repetition of Expression
		(<i>Regexp</i>)	Nested Expression

<i>Character</i>	::=	'(¬)'	Regular Character
		'\"'	Single Quote
		'\\'	Backslash
		'\n'	Linefeed
		'\r'	Carriage Return
		'\t'	Horizontal Tabulation
		'\b'	Backspace
		'\ (0..9)(0..9)(0..9)'	ASCII Character Code

<i>Channel</i>	::=	<i>Name</i>	Channel Variable
<i>Agent</i>	::=	<i>Name</i>	Agent Variable
<i>Site</i>	::=	<i>Name</i>	Site Variable
<i>Ambient</i>	::=	<i>Name</i>	Ambient Variable
<i>Macro</i>	::=	<i>Upper</i>	Macro Variable

<i>Integer</i>	::=	(-) [?] (0..9) ⁺	Integer Constant
<i>String</i>	::=	"((¬) (\\")) [*] "	String Constant
<i>Float</i>	::=	<i>Integer</i> .(0..9) ⁺ ((e E)(+ -)(0..9) ⁺) [?]	Float Constant
<i>Address</i>	::=	(0..9) ⁺ .(0..9) ⁺ .(0..9) ⁺ .(0..9) ⁺ : <i>Integer</i>	Address Constant
<i>Name</i>	::=	(a..z)(A..Z a..z 0..9 _ \') [*]	Name Constant
<i>Upper</i>	::=	(A..Z)(A..Z a..z 0..9 _ \') [*]	Uppercase Constant
<i>Keywords</i>	::=		

agent	bool	char	else	float	false	if	in
int	let	list	migrate	new	out	rec	show
site	string	sub	then	true	type	void	