The SICLE Control Package

version 1.0

Pawel Gburzynski

Olsonet Communications Corporation
and
Department of Computing Science
University of Alberta
Edmonton, Alberta, Canada T6G 2H1

Contents

1 Introduction

The SICLE version described in this manual is identified in the title. Information contained herein is subject to change in future versions of the package.

This manual makes many references to SIDE, which is described in a separate document. Although it may be useful to become acquainted with SIDE before using SICLE, there is no absolute need to do so. It is assumed, however, that the reader is familiar with Tcl.

1.1 Purpose

This manual describes SICLE—a Tcl-based software package for developing and running reactive scripts. The primary purpose of SICLE was to control *reactive systems* understood as collections of physical sensors and actuators. Conceptually, and to a large extent functionally, SICLE is a re-implementation of SIDE in a scripting language.[1] However, owing to the nature of Tcl (and scripting in general), SICLE, in comparison to SIDE, offers some extra features:

- SICLE is ideal for hacking quick ad-hoc solutions
- SICLE is naturally and trivially portable to all environments that support Tcl
- in a sense, SICLE is more powerful than SIDE because it can execute dynamically created (e.g., uploaded) code

One drawback of SICLE with respect to SIDE is the slower execution speed. This need not be a serious problem in many applications. Besides, serious applications may consist of several modules, some of them implemented in SICLE and some in SIDE. SICLE follows essentially the same programming paradigm as SIDE (lightweight uninterruptible threads looking like fast interrupt service routines with multiple entries), and the same communication paradigm. Therefore, SICLE programs can be naturally interfaced with SIDE programs via TCP/IP ports.

The primary purpose of SICLE was implementing control programs for reactive systems in not so time-critical applications. To this end, SICLE comes equipped with an interface to SDS sensors and X10 modules, including a built-in authenticating server accepting remote sensor commands on a TCP/IP port. Thus, it can be directly used to control X10-compliant home appliances and industrial SDS networks, possibly over the Internet.

Another class of applications for SICLE include lightweight servers, e.g., for web databases. The package has been used to implement a web database for processing applications for graduate studies at the Computing Science Department of the UofA.

[1]The name SICLE comes from combining “side” with “tickle” (which is the generally accepted way of pronouncing “Tcl”).

1.2 Reactive systems

By a reactive system we understand any physical system that responds to external *stimuli* and triggers *events* that may be perceived by its observer. This definition is wide enough to encompass all physical devices that exhibit some organized behavior, as well as interconnections of such devices into possibly large networks of dynamic and interacting components. One example of such a network is a modern factory in which manufacturing equipment is interconnected and organized around a common goal—the production of some goods.

For the purpose of computer control and algorithmic description, a reactive system is viewed as a communication network equipped with some processing power, whose terminal devices are of two basic types: *sensors* and *actuators*—see figure 1. The sensors perceive the world and transform this perception into events. Those events propagate to the processing agents of the network (i.e., programs run on computers), where they are interpreted and transformed into messages sent to the actuators. In response to those messages, the actuators perform specific physical actions that make the system behave in a prescribed way.

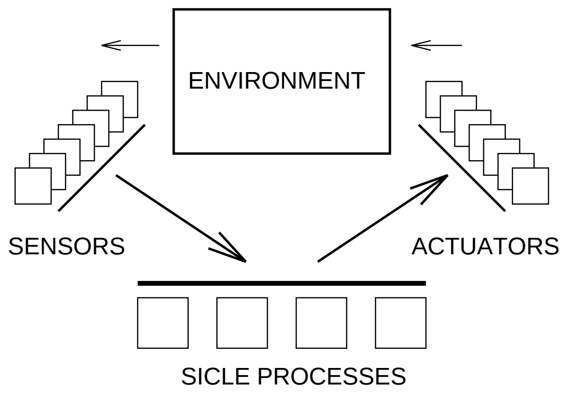

Figure 1: A reactive system.

One immediate application area for SICLE is any physical system that can be represented by a network of sensors and actuators. The primary goal of SICLE is to provide a platform for developing and executing control programs for networks of sensors and actuators that may be interconnected or accessed via the Internet.

1.3 Philosophy of SICLE

SICLE offers a collection of Tcl functions (a Tcl package) for implementing multi-threaded reactive programs. The threads of those programs look very similar to SIDE *processes* and follow essentially the same idea. Thus, following the tradition, we will call them "processes," although they rather resemble uninterruptible interrupt service routines responding to events.

SICLE threads are objects, which means that SICLE is object-oriented, at least when needed. At the same time, all it provides is a set of Tcl functions, and Tcl itself does not look like an object-oriented language. Although there exist object-oriented extensions of Tcl that could be used as the basis for developing our platform, we decided to do it our private way and base SICLE on pure Tcl. One reason for this was our conviction that scripting languages need not (and should not) be inherently object-oriented.

The object-orientedness of SICLE only comes into play when needed, and the chunks of code that need not formally look like methods belonging to some specific classes are in fact (global) functions. One may say that C++ gives us the same paradigm, so there is nothing new in this approach. However, owing to the associative way of implementing objects in SICLE, one can have a single (formally global) function that can be invoked as a method of many formally different objects. This gives us a significant amount of flexibility and, combined with dynamic binding of constructors and destructors, results in an amazingly simple, powerful, and unconstrained scripting flavor of object-orientedness.

A program in SICLE can be implemented as a single multithreaded, event-driven module, or as a set of modules run on independent (possibly diverse) machines connected via the Internet. Some of those modules (those for which execution time is critical) may be implemented in SIDE. All the comments from the SIDE manual regarding the character of a module and its ability to communicate with other modules (or with the outside word) also apply to SICLE. But in contrast to SIDE, SICLE does not have the simulation mode: scripting simulation does not seem to make a lot of sense.

1.4 Installing and using SICLE

SICLE must be installed as a Tcl package before it can be used. To do it, you have to perform the following steps:

1. Unpack the SICLE archive (file `siclexx.tar.gz`) wherever convenient.
2. Move to directory `siclexx` (`xx` stands for the version number).
3. Edit the `Makefile` and set `PACKDIR` to the directory where Tcl packages are kept.
4. Execute `make all` followed by `make install`. The second command must be executed as `root`.

This will in fact install two packages called `siclef` and `sicled`. They are functionally identical, except that the second one is to be used for debugging, while the first one is a "production" version of the package. The primary difference is in execution time. The debugging variant is slower because it validates arguments of SICLE functions and keeps track of a number of most recent function calls.

To get access to SICLE functions, you have to `require` one of the two packages, e.g., with the following Tcl statement:

```
package require sicled 1.0
```

You can write `siclef` instead of `sicled` to use the faster version.

2 Objects in SICLE

Not all features of SICLE must be used in all programs. For example, the package implements DES encryption/decryption with salt, which is intended for authentication. This feature can be used alone (it is visible as two functions presented in section 5.3.1) in any Tcl program. Similarly, SICLE objects constitute a logically separate concept. They can be used in any Tcl program, possibly one that doesn't care about other features, like processes or sensors.

2.1 Object type declaration

A program in Tcl is entirely dynamic, which means that there are no declarations with a static meaning. However, some SICLE (and Tcl) operations resemble declarations in that they assign interpretation to the subsequent occurrences of some symbols in the program. Standard Tcl functions `proc`, `global`, and `upvar` fall into this category.

The following SICLE operation declares an object type:

`class` *typename arglist constructor destructor*

where the only required argument is *typename* (the remaining arguments default to empty strings). Following the execution of this operation, *typename* becomes a legitimate object type.

Argument *arglist* gives the formal list of arguments to be passed to the constructor (represented by *constructor*). It makes no sense to specify *arglist* if *constructor* is empty. The constructor is a Tcl program to be executed when an object of the declared type is created (section 2.3). Similarly, the destructor is a program executed when the object is destroyed (section 2.3). Note that destructors have no arguments.

Examples

Consider the following type declaration:

```
class Complex { re im } {
  useown Re Im
  set Re $re
  set Im $im
}
```

which defines an object type with a constructor but with no destructor. We can easily guess that type is intended to represent complex numbers.

Here we have another type declaration:

```
class Item { what } {
  useown MyStuff
  global List
  set MyStuff $what
  set List($This) ""
} {
  global List
  useown
  unset List($This)
}
```

which defines both a constructor and a destructor. Operation `useown` is described in section 2.5.2.

A constructor/destructor is executed as a function invoked at the level at which the operation creating/destroying the object was performed. The actual arguments of the `create` operation (section 2.3) are passed to the constructor according to the formal specification at `class` declaration.

Similar to a regular Tcl function, the last argument of a constructor can be `args` which stands for the list of all the remaining arguments specified at object creation. This way, constructors can accept actual argument lists of variable size.

There is an operation to check whether a given name represents a valid (declared) object type. The function

`defined class` *typename*

returns 1 if the specified type name corresponds to a declared class type, and 0 otherwise. Operation `defined` can verify several other predicates about objects, object types, attributes, methods, etc. Its full semantics is described in section 6.4.

2.2 Type qualification

Formally, there is no type inheritance in SICLE. Regardless of its formal type, a SICLE object can be built of whatever attributes we decide to associate with it. Those attributes can be referenced in a uniform way by any function, just by specifying the object handle and the attribute name.

In some cases, however, we would like to be able to say that objects of some type are to be treated in some special standard way. For example, a SICLE process (see section 3) is an object of any type that has been “qualified” as a process type. Similarly, a SICLE mailbox (section 4.3) is an object of any type qualified to be a mailbox type. In a sense, type qualification works like reverse inheritance: the current structure of the qualified type is augmented by some standard attributes and operations (methods) that all types qualified this way are expected to have. For example, each type qualified as "process" receives the following attributes: `State`, `Message`, `Priority`, and one extra constructor (see section 2.4) that starts the process up upon creation.

The following simple tool marks a type as qualified:

`qualify` *typename qualifier*

where *typename* is the name of the qualified type, and *qualifier* is the qualification. At this level, the qualification is formal and consists exclusively in marking the type as being qualified to the specified qualifier (which is just a name). The built-in operations for declaring processes, mailboxes, and other qualified types call `qualify`, but they also make sure that the qualified types receive their special attributes and methods that come with the qualification.

There exist variants of `defined` to check whether a given type (or object) is qualified. The following function:

`defined qualify` *typename qualifier*

returns 1 if the given object type is qualified to the specified qualifier, and 0 otherwise. By replacing `qualify` with `is`, one can put an object handle (section 2.3) in place of *typename* and then the operation will verify the type qualification of the indicated object.

2.3 Object creation and destruction

An object is created with the following function:

`create` *typename arg1 ... argn*

where *typename* is the type name of the created object, and *arg1 ... argn* represent the constructor arguments. Their number must coincide with the number of formal arguments for the constructor (section 2.1) or multiple constructors (section 2.4) defined for the object type.

The function returns an object handle, which is a string acting as a "pointer" to the object. Different objects are guaranteed to have different handles. The handles are never recycled, even if the objects are destroyed; therefore, they can be used as absolutely unique identifiers of objects throughout the entire execution time of a program.

The following operation deletes an object:

`delete` *handle*

where the argument specifies the object handle. The argument can be absent in which case the operation deletes the *current* object (section 2.5.2).

Examples

With the following operation, we create a complex number described by the object type from section 2.1:

```
set com [create Complex 1.0 1.0]
```

Then, the following operation will destroy it:

```
delete $com
```

When an object is deallocated, all its attributes (section 2.5.2) are destroyed and cease to exist. However, if some of those attributes store object handles, the objects pointed to by those handles are not destroyed.

The handle of a deleted object becomes invalid. An attempt to reference such an object (e.g., to get its attribute or invoke its method—section 2.5) will trigger an exception. There is a way to check whether an object handle points to an existing object. The following variant of `defined` (section 6.4):

```
defined valid handle
```

returns 1 if the handle points to an existing object, and 0 otherwise.

Using `defined`, it is also possible to determine whether an object handle points to an existing object of a given type. This is accomplished by the following variant:

```
defined belongs handle type
```

which returns 1 if the object pointed to by the handle exists and its type is *type*, and 0 otherwise.

Another useful function is

```
gettype handle type
```

which returns the name of the object's type or an empty string, if the specified handle does not point to an object.

2.4 Dynamically added constructors and destructors

It is possible to add new constructors and destructors to an object type after the object type has been declared. Specifically, a new constructor is added by the following operation:

`constructor` *typename arglist constr*

Similarly, a new destructor is added by

`destructor` *typename constr*

All arguments are required. In both cases, the first argument identifies the object type to which the constructor/destructor is added, and the last argument specifies its code.

Semantically, multiple constructors are put into a list and executed in the (dynamic) order of their definition. The multiple argument lists are concatenated in the same order; the actual argument list specified at `create` must match the combined argument list of all constructors. If `args` is used as the last argument of a constructor (to represent a variable length trailer of the argument list), it must be the last constructor defined for the given object type.

Multiple destructors are executed in the reverse order of their definition, i.e., the last defined destructor is executed first.

Examples

Let us add a constructor and destructor for the `Item` type from section 2.1.

```
constructor Item { slist } {
  useown SecondList
  upvar $slist SList
  set SList($This) ""
  set SecondList $slist
}
destructor Item {
  useown SecondList
  global $SecondList
  unset ${SecondList}($This)
}
```

The meaning of `This` and `useown` is described in section 2.5.2. We can guess that the second constructor adds the handle to the object to a second list (an array) whose name is specified in the argument. The destructor just removes that entry.

The right way to create an object of type `Item` is to specify two constructor arguments at `create`, e.g.,

```
set NewItem [create Item $Stuff KeepTrack]
```

where `KeepTrack` is an array.

Using `defined` (see section 6.4), one can check whether a given object type defines constructors and/or destructors. The operations

```
defined constructor typename
defined destructor typename
```

return 1, if there exist constructors/destructors for the specified type, and 0 otherwise.

2.5 Object methods and attributes

There is no need to specify (declare) all attributes associated with an object when the object type is declared. Objects in SICLE are not implemented as chunks of memory with preallocated space to hold their attributes, but rather as abstract and necessarily untyped baskets that can be filled with attributes as needed. In other words, attributes are associated with objects dynamically, as they are introduced in the program. It is quite possible to have two instances of formally the same object type with completely different and unrelated sets of attributes.

2.5.1 Methods

An object method is a function associated with a specific object type. This association is established dynamically by performing the following operation:

```
method typename methodname arglist body
```

All arguments are required. The operation associates a method named *methodname*, whose formal arguments and body are specified by the last two arguments, with the object type specified as the first argument.

Example

This is a sample (argument-less) method associated with type `Complex` defined in section 2.1:

```
method Complex module { } {
  useown Re Im
  return [expr sqrt (Re*Re + Im*Im)]
}
```

There are two ways to call a method: one with the `invoke` operation, `invoke` *handle methodname arg1 ... argn*

and the other with `ask`,

`ask` *methodname arg1 ... argn*

The second style takes no object handle and assumes that the method being invoked belongs to the *current* object (see section 2.5.2).

Constructors and destructors can be viewed as special (unnamed) methods that are invoked automatically under certain circumstances.

Using `defined` (section 6.4), it is possible to check whether a specific method is defined for a given type or object. The following operation:

`defined method` *typename methodname*

returns 1 if the specified method is defined for objects of the indicated type, and 0 otherwise. Given an object handle (rather than object type), the following function:

`defined callable` *handle methodname*

returns 1 if the object pointed to by the handle exists and its type defines the indicated method, and 0 otherwise.

The following alternative to `invoke`:

`implore` *handle methodname arg1 ... argn*

does the same job as `invoke`, except that it does not trigger an exception if the called method does not exist.

To see a possible application of this feature, consider a function whose role is to process objects belonging to several different, albeit similar, types. Note that in SICLE, where object attributes are associative rather than strongly typed into the objects, objects of different types may be processed by the same functions, as long as these functions restrict themselves to using only those attributes and methods that are the same for all the objects being processed. This is why inheritance is not much needed in SICLE: we may live quite comfortably without it, especially, if we avoid introducing unnecessary restrictions. Occasionally we may get into a situation where some objects define a specific method (e.g., for preprocessing or initialization), whereas some others do not (e.g., because they don't need it). By using `implore` instead of `invoke` to call such a method we make sure that it only gets called if it exists; otherwise, the call is quietly ignored.

At any moment, while a SICLE program is being executed, it either *is* within some object or not. At the beginning, when the program starts, it clearly isn't in any object. When an object is created (by `create`—section 2.3), the object's constructor executes within the created object. Similarly, when an object method is invoked (by `invoke`—section 2.5.1), it executes within the object pointed to by the handle. This is exactly how it happens in a traditional object oriented language, e.g., C++.

However, in SICLE, when you call any (non-method) function while within an object, the function itself will execute within the same object. The primary difference between a method and a regular function in SICLE is that a method is always associated with a specific object type; thus, you may have several different methods with the same name associated with different object types. It makes no sense to call a method without specifying explicitly (`invoke`) or implicitly (`ask`) the object type to which the method belongs. On the other hand, a function can be called anywhere and it is always the same function regardless of the context.

Whenever a method is `invoke`d (section 2.5.1), and the specified handle identifies an object different from the current object, the current object is pushed on the stack and a new current object is assumed. When the method eventually returns, the previous current object is popped from the stack and it becomes the present current object. If there was no current object underneath (i.e., the method returns to level 0), current object becomes undefined.

One more feature somewhat related to methods is the possibility to change the notion of current object without actually invoking a method. By executing

`enter` *handle*

we put the current object on the stack and assume the context of the object pointed to by the handle. To revert to the previous current object, execute `leave` (which takes no arguments).

2.5.2 Attributes

At any moment when current object is defined (section 2.5.1), the following operation is legitimate:

`useown` *att1 ... attn*

where *att1 ... attn* identify some attributes of the current object. These attributes become visible (they can be referenced as regular local variables) until the function or method that performed `useown` returns, or until their names are subsumed by another `useown`, `use` (see section 2.5.3), `upvar`, `global`, etc. Thus, the operation has essentially the same meaning as `global`, except that instead of specifying global variables, it specifies the attributes of the current object to be visible from the scope of the present function.

Besides the indicated list of attributes, `useown` makes one more variable visible to the current function or method. This variable is called `This` and it represents the handle to the current object. Thus, it makes sense to execute `useown` without arguments.

We have seen examples of `useown` in sections 2.1 and 2.4. Notably, the operation can be used anywhere (not only in a method, constructor, or destructor), as long as current object is defined. In particular, any function called from a method can access the attributes of the current object in exactly the same way as the method itself.

2.5.3 Other ways of referencing attributes

Sometimes `useown` is insufficient, e.g., how to reference an attribute of an object that doesn't happen to be the current object at the moment? Given an object handle, the following two operations can be used for this purpose:

> `getattr` *handle attname*
> `setattr` *handle attname value*

The first operation is a function that returns the current value of the attribute whose name is specified as the second argument. Of course, this name can identify an array element. With the second operation, the attribute specified by the second argument is set to the third argument.

Example

The following method of a hypothetical type `QueueItem` locates the end of the indicated queue of items and appends the current object at the end:

```
method QueueItem append { iqueue } {
  upvar $iqueue ItemQueue
  useown Next
  set Next ""
  if { $ItemQueue == "" } {
    # the queue is empty
    set ItemQueue $This
  } else {
    set previous $ItemQueue
    while { [getattr $previous Next] != "" } {
      set previous [getattr $previous Next]
    }
    setattr $previous Next $This
  }
}
```

A `getattr` performed on an undefined attribute will trigger an exception. There is a variant of `defined` (section 6.4) checking whether a specific attribute is defined for an object (similar to `info exists` for regular variables). The following function:

`defined attribute` *handle attname*

returns 1 is the attribute *attname* is defined for the object pointed to by *handle*, and 0 otherwise.

In some circumstances, it may be convenient to use the following function:

`checkattr` *handle attname*

which works exactly as `getattr`, except that it returns an empty string (rather than triggering and exception) if the referenced attribute doesn't exist.

Referencing many attributes of the same non-current object with `getattr` and `setattr` may be inconvenient. The following operation:

`use` *handle att1 ... attn*

is very similar to `useown`, except that the attributes are from the object pointed to by the handle. Another difference is that, in contrast to `useown`, `use` does not define `This` (section 2.5.2).

Example

The following function copies some attributes of one object to another:

```
proc copyit { o1 o2 } {
  use $o1 Width Height Depth
  use $o2 Wi Hi Dp
  set Wi $Width
  set Hi $Height
  set Dp $Depth
}
```

The above example is simple although somewhat artificial. If the names of the relevant attributes were the same in both objects (a more likely scenario) we couldn't get away so easily. One can always use `getattr` and `setattr` in such circumstances, but it is also possible to assign different local names to the attributes. This is accomplished by the following operation:

`usealias` *handle atlist allist*

where *atlist* is the list of attributes to become visible, and *allist* is the list of local aliases under which they should appear. The two lists must have the same number of items.

Example

Now we can rewrite the above example as follows:

```
proc copyit { o1 o2 } {
  usealias $o1 { Width Height Depth } { W1 H1 D1 }
  usealias $o2 { Width Height Depth } { W2 H2 D2 }
  set W2 $W1
  set H2 $H1
  set D2 $D1
}
```

As we said before, it is not uncommon for different objects belonging to the same type to consist of quite different sets of attributes. It is possible to get the list of all attributes associated with a given object with the help of the following function:

`attributes` *handle*

The argument is optional and it defaults to an implicit handle to the current object. One should be warned that not all attributes occurring on the list returned by `attributes` must have their values defined. For example, when an attribute appears on the `use` or `useown` list, it becomes associated with the object (and its name will appear on the list returned by `attributes`) even if it is never assigned a value. An array is treated as a single attribute, i.e., only the array's name appears on the list.

Example

This is the way to make all the already existing attributes of the current object locally visible:

```
useown [attributes]
```

3 Processes

Using the tradition of SIDE and its predecessors SMURPH and LANSF, we will call SICLE threads "processes," although this name may appear somewhat exaggerated. SICLE processes are in fact simple and uninterruptible co-routines with implicit control transfer, which makes them appear as independent, possibly communicating, threads.

Usually, the code of a process is structured as a finite state machine with clearly visible states represented by multiple cases of a `switch` statement. A process is typically dormant most of the time awaiting some events. The occurrence of the earliest of them wakes the process up in a specific prescribed state. Then the process performs some operations and goes back to sleep indicating the events that will wake it up in the future.

Thus a process can be viewed as a conceptually fast interrupt service routine with a re-programmable configuration of serviced interrupts. Those interrupts, i.e., waking events, may arrive from other processes, from timers, and also from external agents like peripheral devices or networking ports.

3.1 Using processes

As we mentioned earlier, a program requesting the SICLE package is not required to use all its features. In particular, for as long as the program doesn't want to use processes, it doesn't have to organize itself in any special way. All the tools for manipulating and accessing objects described in section 2 are freely usable this way, i.e., as Tcl functions callable in a straightforward way from a straightforward Tcl program.

Things somewhat change when we decide to use SICLE processes. As multiple processes can co-exist at any given moment, they require a "kernel" of sorts responsible for scheduling them and interpreting their so-called *wait requests.* At some point, the program must enable the kernel, declaring that it has initialized itself, i.e., created some initial configuration of processes, and the rest of its activities are to be carried out by those processes.

A program that intends to use processes should execute the following operation:

`sicle` *arg*

before creating the first process. Although `sicle` is not required to be the first statement of the program, it is safe and natural to put it at the very beginning, immediately after `package require` (section 1.4).

If the (optional) argument of `sicle` contains the string `sensors`, it switches on the initialization of the built-in interface to sensors and actuators (see section 5). This interface consists of a set of functions interpreting the contents of the *sensor map* file and a daemon making operations on sensors accessible remotely via a TCP/IP port.

Having created an initial configuration of processes, the program should execute

```
kernel
```

which only makes sense as the last statement of the program. The operation never returns: it enters the kernel's event loop to interpret events in the system and schedule the processes.

3.2 Process declaration, creation and destruction

A processes is an object whose type has been qualified (section 2.2) as "process." This is accomplished with the following operation:

`process` *typename code*

where *typename* is an unqualified object type and *code* is the body of the process *code method.*

Example

Let us define a simple process that will wake up every prescribed interval and write a line of text to the standard output. We start with the following type definition:

```
class Alarm { delay } {
  useown Delay
  # convert the delay to milliseconds
  set Delay [expr 1000 * $delay]
}
```

Typically, the basic type of a process describes the initialized attributes of the process and provides a constructor to initialize them.

Now we can say that `Alarm` is actually a process type:

```
process Alarm {
  global Timer
  useown Delay
  puts stdout "Alarm awakened"
  wait $Timer $Delay
}
```

We can create our process in the following way:

```
set AHandle [create Alarm 10]
```

The process code method will write a message to the standard output, then call `wait` and exit. We say that the process "issues a wait request to the timer"—to be awakened `$Delay` milliseconds (10 seconds) after the current moment. At that time, the process code method will be executed again.

Operation `process` adds one constructor to the list of constructors of the qualified type. This constructor accepts one optional argument that specifies the so-called "startup priority" of the process (see section 3.3.1). The legitimate values of this argument are 0 and 1 (alternatively `low` and `high`), which stand for *low* and *high* startup priority. The default value of startup priority is 0.

Every process also receives three extra attributes which are primarily intended to describe the process's *waking environment*. One of them, called `State`, identifies the current state of the process viewed as a finite-state machine. When the process code method is invoked for the first time (immediately after process creation), the value of `State` is `Start`.

A process is terminated by executing `delete` on its object handle. Such a process is stopped and discarded from the system. Note that a process can terminate itself this way (i.e., it is legal to deallocate the process object from its code method), but it can also be terminated by another process.

Example

Let us modify the process from the previous example in such a way that it will wake up only a prescribed number of times and then terminate itself. This is how it can be done:

```
class Alarm { delay ntimes } {
  useown Delay NTimes
  set Delay [expr 1000 * $delay]
  set NTimes $ntimes
}
process Alarm {
  global Timer
  useown Delay NTimes
  puts stdout "Alarm awakened"
  incr NTimes -1
  if { $NTimes } {
    wait $Timer $Delay
  } else {
    delete $This
  }
}
```

In fact, the `delete` operation is superfluous because a process that doesn't specify a waking condition before returning from its code method is effectively terminated and deallocated (see section 3.3).

3.3 Process operation

A process operates in a cycle consisting of the following steps:

1. the process is awakened by one of the awaited events
2. the process responds to the event, i.e., executes a fragment of its code method
3. the process puts itself to sleep

Before a process puts itself to sleep, it usually issues at least one *wait request*—to specify the event(s) that will wake it up in the future. A process may also issue a *persistent* wait request (see section 3.3.2), to be restarted cyclically by the subsequent occurrences of the same event. A process that goes to sleep without specifying a single waking condition is terminated. There is no sense to keep such a process around, as it will never run again. The effect is exactly the same as if the process performed `delete` on its object handle as its last statement (section 3.2).

To put itself to sleep, a process (its code method) can execute the following statement:

```
sleep
```

or simply return from the code method.

The following operation handles wait requests:

`wait` *ai event state priority*

Only the first two arguments are mandatory. The first of them identifies the agent (the so-called *activity interpreter* (*AI* for short) responsible for triggering the waking event; the second argument specifies the actual event.

Activity interpreters are discussed in section 4. We saw one of them, the timer, in the example in section 3.2. In that case, the event was the delay in milliseconds after which the timer was expected to go off.

Argument *state* is optional and defaults to an empty string. It specifies the state to be assumed when the awaited event wakes up the process. This state will be returned in the process attribute `State` which can be examined by the code method.

Example

The `Alarm` process from the example in section 3.2 has only one state. It is natural to organize the code method of a multiple-state process into a `switch` statement. Suppose that the first `puts` statement in `Alarm` should be executed after the delay, not immediately after the process is created. The new code method can be written as follows:

```
process Alarm {
  useown State
  switch $State {
    Start {
      global Timer
      useown Delay
      wait $Timer $Delay WakeUp
    }
    WakeUp {
      puts stdout "Alarm awakened"
      wait $Timer $Delay WakeUp
    }
  }
}
```

If a process issues more than one wait request before going to sleep, the multiple wait requests are interpreted as an **alternative** of waking conditions. It means that when the process becomes suspended, it will be waiting for a collection of event types, possibly coming from different *AI*s, and the occurrence of the **earliest** of those events will reactivate the process. When a process is restarted, its collection of wait requests is **cleared**, which means that in each operation cycle these events must be specified from scratch.

3.3.1 Event priority

It is possible that two or more events from the pool of events awaited by a process occur at the same time. This is not at all unlikely, because some events may be pending at the moment when the wait requests for them are issued. In such a case, the event for which the wait request was issued **first** will be the one presented to the process. This simple rules also explains how multiple processes are scheduled. When two or more processes are awakened by events occurring at the same time, the order in which they are restarted is determined by the order of the wait requests for those events.

This is what happens if the processes do not use the fourth argument of `wait` to specify the **priority** of the wait request. The argument can only take two values, 0 and 1, alternatively `low` and `high`, representing the low and high priority, respectively. The default value, assumed when the argument is absent, is 0.

If two events awaited with different priorities occur at the same time, the one with the high priority will be presented first. This rule applies to multiple events awaited by a single process, as well as to events awaited by multiple processes, which occur at the same time. If two or more high priority events occur simultaneously, they are processed in the order of the corresponding wait requests.

When a process is awakened, its standard attribute `Priority` tells the priority with which the process was restarted. The attribute can only take values 0 (low priority) and 1 (high priority). Another standard attribute, `Message` may contain an optional "message" passed to the process by the *AI* responsible for delivering the waking event. The contents of `Message` are *AI*-specific; if not used, `Message` contains an empty string.

Immediately when a process is created, a waking event for the process is generated and scheduled to occur immediately. This event will wake up the process for the first time with `State` equal to `Start`. Standard scheduling rules apply to the first waking event; by default, the priority of this event is `low`. To make it `high`, the last argument of `create` should be 1 (or `high`). This is the optional argument of the extra constructor for the process type, which was added by the `process` operation (section 3.2).

3.3.2 Persistent wait requests

Wait requests issued with `wait` (section 3.3) are "volatile": they are forgotten whenever the process is awakened. Sometimes we would like a wait request to remain active for a time significantly longer than a single invocation of the process's code method, perhaps for the entire lifetime of the process. Rather than repeat the request each time the process is awakened, we can resort to the following operation:

`monitor` *ai event state priority*

The arguments have exactly the same meaning as for `wait` (section 3.3). The only difference between `wait` and `monitor` is that a request issued with the latter command is *persistent*, i.e., it remains active until it is explicitly canceled with `cancel` *ai event*

The operation undoes the effect of the last `monitor` command with the specified *AI*/event combination.

Example

Let us redo once again the example from section 3.2. This time, we will do it right, i.e., in the most natural and efficient way. The basic type of our process is declared exactly as before, but its code method is now rewritten as follows:

```
process Alarm {
  useown State
  switch $State {
    Start {
      global Timer
      useown Delay
      monitor $Timer $Delay WakeUp
    }
    WakeUp {
      puts stdout "Alarm awakened"
    }
  }
}
```

This solution is similar to the version from section 3.3. The `wait` operation from the first state has been replaced by `monitor`. This makes the second `wait` statement (in state `WakeUp`) unnecessary.

3.3.3 Unconditional state transition

State-less processes, e.g., like the one introduced in the example in section 3.2, are rather uncommon. Typically, a process has several states and then its code method is organized into a `switch` statement (e.g., see section 3.3.2). The process transits among its multiple states by awaiting and receiving events.

Sometimes it is desirable to make a direct state transition, i.e., to perform a *goto* from one state to another. This can be accomplished with the following operation:

`proceed` *state*

The mandatory argument identifies the new state. The effect is as if the process code method was called again (from the previous level, not recursively) with the `State` attribute containing the specified value. In fact, rescheduling is possible during this transition, which is performed as a response to a hypothetical event awaited with `high` priority. It is semantically equivalent to the following sequence:

```
wait $Timer 0 state high
sleep
```

although a bit more efficient.

Example

By inserting `proceed WakeUp` after the `monitor` statement in the code method of `Alarm` in section 3.3.2, we can force the process to behave as its very first version in section 3.2. The process will execute the `puts` statement immediately after creation, and then every `$Delay` milliseconds afterwards.

4 Activity interpreters

Activity interpreters are the objects responsible for triggering waking events for processes. One of them is the timer (see the example in section 3.2) accessible via the global variable `Timer`. This *AI* occurs in a single instance and its primary purpose is to implement alarm clocks. Every process is also an *AI*; it can generate events perceptible by other processes at state transitions and at the moment of its termination. The remaining types of *AI* are *mailboxes* (used for process communication) and *interfaces* (which interface the program with peripheral devices or networking ports).

4.1 The timer *AI*

Although formally the timer *AI* occurs in a single instance (globally visible via the variable `Timer`), it can be viewed as an unlimited collection of individual alarm clocks available to processes. The most typical application of the timer *AI* is to put a process to sleep for a prescribed amount of time. The format of a `wait` request for this occasion is

```
wait $Timer delay state priority
```

where *delay* must be an unsigned integer number specifying the amount of delay in milliseconds. The awaited event will be triggered the prescribed number of milliseconds after the current time, assuming that the process will not be awakened by an earlier event.

It is possible to issue a persistent delay request to the timer using `monitor` instead of `wait` (section 3.3.2). In such a case, whenever the process is put to sleep, an implicit timer request will be issued to wake the process up after the specified delay. Note that the process may be awakened by another event (i.e., one that occurs before the timer goes off). Following the processing of that event, when the process puts itself to sleep again, the alarm clock will be reset for the original number of milliseconds.

Another possible timer event is one that occurs at the specified time of day. In this case, the *delay* argument of `wait` must be a string representing a time of day in one of the formats accepted by the standard Tcl function `clock scan`. This format of the *delay* argument can also be used with `monitor`, but then, once the event is triggered, it is automatically `cancelled` (section 3.3.2), as there is no point in keeping it around any more.

Example

Consider the following process:

```
class Ticker { u h } {
  useown Until HowOften
  set Until $u
  set HowOften $h
}
process Ticker {
  global Timer
  useown State HowOften Until
  switch $State {
    Start {
      monitor $Timer $HowOften Tick
      wait $Timer $Until Done
    }
    Tick {
      puts stdout "Tick"
      wait $Timer $Until Done
    }
    Done {
      puts stdout "No more ticks"
      delete $This
    }
  }
}
```

which writes “ticks” to the standard output every specified interval (`HowOften`) until the the time reaches (or exceeds) the specified limit (`Until`). At state `Done` the process terminates itself (section 3.2). Note that replacing `delete` with `cancel $Timer $HowOften` would have the same result. As we said in section 3.3, a process that goes to sleep (returns from its code method) without specifying a single waking condition that may resurrect it in the future is terminated and ceases to exist.

The process cane be simplified a bit by replacing the wait request in state `Start` with

```
monitor $Timer $Until Done
```

and removing the wait request in state `Tick`.

This is a sample way to create our process:

```
global Quarter
create Ticker "22:30:00" $Quarter
```

The global constant `Quarter` specifies the number of milliseconds in 15 minutes; thus, the process will "tick" every quarter until 10:30 p.m. Other convenient constants of the same kind are: `Second`, `Minute`, `FiveMinutes`, `TenMinutes`, and `Hour`.

Note: timer events never set the `Message` string (section 3.3.1). There is no message that comes with a timer event, other than the event itself.

4.2 The process *AI*

Every process is also an activity interpreter, which means that its handle can appear as the first argument of `wait` or `monitor`. The event description in this case is a string representing the process's state identifier. String "`Death`" is reserved for a special (and non-existent) state corresponding to process termination.

The semantics of a state wait request issued to another process is to ask for an event to be triggered when that process gets into the specified state. The event will be triggered **after** the process has completed its processing of that state.

Example

The most useful application of state waiting is to spawn a process and wait for its termination. For example, with the following statement:

```
wait [create Ticker "Monday" $Hour] Death TickerDone
```

we can create a `Ticker` process presented in section 4.1 and wait for its termination. When `Ticker` terminates, we will transit to state `TickerDone` (assuming we do not wait for other events that may interrupt our waiting).

Both operations `wait` and `monitor` can be used to issue state wait requests. Of course, the `Death` event for a given process can only occur once.

Note: process events never set the `Message` string (section 3.3.1). There is no message that comes with a process event, other than the event itself.

4.3 The mailbox *AI*

Mailboxes are containers for arbitrary strings, which we will call "items." It is possible to deposit an item into a mailbox, extract an item from a (nonempty) mailbox, or wait until a mailbox gets into some specific state. Items in a mailbox are stored in the order in which they have been deposited. It isn't absolutely necessary to extract them in this order, although in many cases this is the natural thing to do.

4.3.1 Declaring and creating mailboxes

A mailbox is an object of a class type qualified as a mailbox type. To qualify a type as mailbox, we use the following simple operation:

`mailbox` *typename*

where *typename* stands for an already defined (and unqualified) class type. The operation adds one extra constructor to the list of constructors already associated with the qualified type. This new constructor takes one optional argument that specifies the mailbox *capacity.* The default capacity of a mailbox is 0 (which is not necessarily useless).

Example

Below we have the simplest possible (yet quite sensible) mailbox type declaration.

```
class MyMailboxType
mailbox MyMailboxType
```

To create an instance of our mailbox, we can do this:

```
set mbx1 [create MyMailboxType 256]
```

This mailbox is capable of storing up to 256 items.

The capacity of a mailbox tells the maximum number of items that the mailbox can accommodate. An attempt to store a new item in a full mailbox will fail. A capacity-0 mailbox is always full and it cannot accommodate even a single item. However, it may still make sense to try to store an item in it, because this operation may trigger events on the mailbox (and wake up some processes).

If the mailbox type defines a method called `output`, it will be called whenever an item is deposited in the mailbox, with the item passed as the only argument. The item is passed "by variable," so that it can be modified by the method before being stored in the mailbox.

The `output` method, if defined, is also called for an item that is not actually stored in the mailbox because it happens to be full.

Similarly, whenever an item is removed from a mailbox for which a method called `input` is defined, that method is called with the item passed (by variable) as the only argument. Using the two methods, the mailbox class can define its private pre-processing for deposited items and/or post-processing for extracted items. It is legitimate to define only one method (and, of course, not to define any of them).

At first sight, the names `input` and `output` may be confusing because `output` gets called when we in fact "input" something to the mailbox and vice versa. We should look at it as if the mailbox were an i/o device: when we "write" something to it, we call the `output` method, and the other way around. This way of interpreting mailbox operations is also consistent with *interfaces* (section 4.4), which can be viewed as special-purpose mailboxes.

A mailbox can be destroyed and deallocated by `delete`—as any regular object.

4.3.2 Operations on mailboxes

The following function adds an item to a mailbox:

`deposit` *mailbox item*

where *mailbox* is a mailbox handle and *item* is the new item to be deposited in the mailbox. The function returns 1 or 0, depending on whether the new item was added to the mailbox or not (because the mailbox was completely filled). Note that for a capacity-0 mailbox, `deposit` always returns 0.

Adding an item to a mailbox (also trying to add an item to a capacity-0 mailbox) may trigger an event awaited by a process (section 4.3.3).

If the mailbox type defines a method called `output`, that method is called with the deposited item passed as the argument (by variable), before the item is actually deposited in the mailbox. It is also called if the item is not deposited because the mailbox happens to be full.

Extracting items from mailboxes is accomplished with the following function:

`extract` *mailbox where pattern*

The last argument is optional and defaults to an empty string.

The function attempts to extract an item from the mailbox pointed to by *mailbox* and store the item in the variable represented by *where*. If *pattern* is empty (or not specified), the first item is extracted from the mailbox. Otherwise, *pattern* is interpreted as a regular expression and the first item matching the expression is extracted.

If the function succeeds, i.e., an item is removed from the mailbox, the item is stored in *where* and the function returns 1. Otherwise, the function returns 0. If the mailbox type defines a method called `input`, that method is called with the extracted item passed as the argument (by variable), after the item is removed from the mailbox. The `input` method may modify the item before it is returned by `extract`.

4.3.3 Mailbox events

A mailbox wait request may specify one of the following events:

`Newitem`

This event is triggered when an item is deposited into the mailbox. Note that it can be triggered on a capacity-0 mailbox.

`Nonempty`
: This event is triggered when the mailbox is or becomes nonempty. It cannot be triggered on a capacity-0 mailbox.

`Receive`
: The event is triggered when the mailbox is or becomes nonempty. The first item is automatically extracted from the mailbox; it will be returned in `Message` when the process is awakened.

`Delete`
: The event is triggered when an item is removed (extracted) from the mailbox.

An integer number
: This event (the so-called *count event*) is triggered when the number of items in the mailbox becomes (or is) exactly equal to the specified number.

Anything else
: Any other string appearing as the event identifier (except "`Attention`" (section 4.5) is interpreted as a regular expression. This event (the so-called *pattern event*) is triggered if the mailbox contains an item matching the expression.

Most mailbox events return via the `Message` attribute of the restarted process (section 3.3.2) the item that has triggered the event. In some cases, this notion is obvious. For example, those events that can only be triggered when some specific item is involved (i.e., `Newitem`, `Receive`, `Delete`) return that particular item. Note that `Newitem` and `Delete` cannot occur at the moment when the wait request is being issued: they are only triggered when some operation is performed on the mailbox (a new item is being added or an existing item is being removed). The remaining events can occur immediately, if the mailbox contents already fulfill some criteria, or they can be triggered later, when the mailbox gets into the required state. Thus, for example, event `Nonempty` occurs immediately if the mailbox happens to be nonempty when the wait request is issued, or may be triggered later by a `deposit` operation (section 4.3.2) performed on the mailbox. In both cases, `Message` returns the first item from the mailbox. Note that the item is not removed when the event is presented.

The first item is also returned by a count event, assuming that the count is not zero. Otherwise, `Message` contains an empty string. A pattern event returns the first item matching the specified regular expression.

The purpose of `Receive` is to implement a safe item acquisition in a situation when multiple processes await items on the same mailbox. Note that the `Nonempty` status of a mailbox does not guarantee that a subsequent `extract` operation will succeed, if there are multiple processes awaiting the `Nonempty` event and extracting elements from the mailbox. Regardless of the priority of the wait request for `Receive`, the event is always triggered with the `high` priority.

Example

Mailboxes are natural tools for inter-process communication and synchronization. For illustration, let us use a mailbox to implement a counting semaphore. The idea is to have a critical section with the property that no more than K processes can be within it at the same time. We can program our mailbox in the following way:

```
class CSem
mailbox CSem
 ...
set Sem [create CSem $K]
```

A process willing to use the semaphore may define the following states whose purpose should be obvious:

```
global $Sem
 ...
  Enter {
    if [deposit $Sem ""] { proceed Entered }
    wait $sem Delete Enter
  }
 ...
  Entered { ... }
 ...
  Exit {
    extract $sem junk
  }
```

4.4 The interface *AI*

An *interface* has some properties of a mailbox. The primary difference is that an interface can be filled and/or emptied by an external agent, i.e., a peripheral device or a networking port. Interfaces provide a SICLE program with a reactive interface to the outside world.

4.4.1 Declaring and creating interfaces

An interface can be any class object qualified as an interface, which is accomplished with the following operation:

`interface` *typename*

The operation adds one extra constructor to the list of constructors already associated with the qualified type. This constructor takes three arguments. The `create` operation for an interface object looks as follows:

`create` *arg1 … argn how device bufsize*

where *arg1 … argn* are the arguments required by the constructors of the base type (in most cases that type has no constructors), and the last three arguments are the specific arguments of the `interface` constructor. In fact, only the first of them (*how*) is required. The remaining two default to an empty string (*device*) and to 256 (*bufsize*).

Argument *how* is a string describing the way in which the interface is to be bound to an external agent. This string may be combined from the following keywords, which may occur in any order and be separated by anything that cannot be confused with one of the legitimate keywords:

`internet`
This keyword means that the interface represents a TCP/IP socket. If it is not present, the interface represents a peripheral device.

`server`
This keyword is only meaningful in connection with `internet`. It declares the socket as a server-end socket, i.e., one on which we want to listen for incoming connections. Otherwise, the socket is a client socket.

`wait`
If used in connection with `server`, it means that we want to wait until there is at least one incoming connection request on the server socket. The constructor will not return before then. Otherwise, when used for a non-`server` (client) socket, it selects asynchronous (fast-return) connection, which is without perceptible semantical consequences. The keyword is irrelevant if used without `internet`.

`single`
This keyword is only meaningful in connection with `server` and `wait`. It means that we want to receive a single incoming connection on the server socket. In other words, we are not really a server and we want to engage in a session with a single party, but the other party will initiate the connection. When the constructor returns, the interface will represent our end of the peer-to-peer connection. Otherwise, i.e., when `single` is absent, the interface represents a "master" socket used for receiving connection requests.

`read`
This keyword is only meaningful for a device interface. It indicates that the device will be used for reading.

`write`
This keyword is only meaningful for a device interface. It indicates that the

device will be used for writing. Note that its is possible to use the same device for both reading and writing, and it is required to use it in at least one of the two i/o modes.

The meaning of *device* depends on the interface type determined by *how*. For a server socket, *device* is the TCP/IP port number on which the socket listens for incoming connections. For a client socket, *device* should have the following format:

hostname:*port*

e.g., `sheerness.cs.ualberta.ca:4987`. For a device interface, *device* should be the name of the device to be opened, as required by the `open` operation of Tcl. Optionally, this name can be followed by a colon, followed in turn by the serial parameters (if the device happens to be a serial device). The format of these parameters is that required by the `-mode` option of the `fconfigure` operation of Tcl.

The last argument specifies the buffer size, i.e., the maximum length of a sequence of bytes that can be acquired from the interface at a time. It should be at least as large as the maximum length of an item extracted from the interface with a single `extract` operation (section 4.4.2). Note that storing items in the interface (i.e., sending them to the external agent) is not subjected to this limitation.

Examples

Below we list a few sample `create` operations for various interfaces. In all cases, we assume that the constructors of the base type need no arguments. In the vast majority of situations, the base type of an interface is in fact trivial.

```
set sok [create Int internet legal.cs.ualberta.ca:3345]
```

The interface represents a client socket connected to port 3345 on `legal.cs.ualberta.ca`. Note that there must be a listener on that port at the moment when the operation is issued, as otherwise it will fail. The input buffer size for the interface is 256 bytes (the default).

```
set cli [create Int internet+server 6677]
```

This interface represents a server socket listening on port 6677. The input buffer size is irrelevant in this case because the sole purpose of this interface is to trigger events caused by incoming connection requests.

```
set peer [create MyInt internet+server+wait+single 6677 1024]
```

Here we have a peer interface disguised as a server interface. The operation will open a server socket on port 6677 and wait for the first incoming connection. Then, the server socket will be closed and the interface will be used to represent our end of the peer-to-peer socket. The input buffer size is 1024 bytes.

```
set d [create MyInt read /dev/ttyS0 1]
```

This is a device interface to be used for reading from `/dev/ttyS0`. Apparently, we want to read one character at a time because the input buffer length is 1.

```
set d [create MyInt read+write /dev/ttyS1:4800,n,8,1 64]
```

Another device interface. This time we specify the serial parameters: 4800bps, no parity, 8 bits, one stop bit. The buffer size is 64 bytes, the device will be used for both reading and writing.

Note that it is legal to create drastically different interface objects from the same, usually trivial, interface type. One possible situation when the interface type may not be trivial is when it defines methods `input` and/or `output` (section 4.4.2) for automatic preprocessing of data before it is presented to the program or expedited to the external agent. Sophisticated implementations of those methods may use attributes of the interface's base type.

4.4.2 Input and output operations

Operations on interfaces, at least the most typical ones, are almost the same as operations on mailboxes. In particular, it is possible to "deposit an item" into an interface with the following operation:

`deposit` *interface item*

where *interface* is an interface handle and *item* is a string. With the above operation, the deposited string is simply written to the external agent associated with the interface, i.e., TCP/IP port or device.

If the interface type defines a method called `output`, this method is called with the deposited string passed as the argument, before it is actually written to the port or device. As the string is passed by variable, it can be modified by the `output` method.

Note that `deposit` never blocks. Internal buffering is used to store the written data before it can be physically accepted by the port or device.

The operation is illegal on an interface that has been associated with a server socket or on a device interface not declared as "writable" (section 4.4.1).

The operation returns 0 if it has succeeded (this may be a bit confusing in confrontation with the mailbox variant—section 4.3.2), and 1 if it has failed. The only reason for interface's `deposit` to fail is to hit an i/o error. This will happen, e.g., if the interface has been `shutdown` (section 4.4.3).

The following operation can be used to extract (read) a piece of string from an interface:

`extract` *interface where pattern*

where *interface* is an interface handle, *where* is a variable to store the result, and *pattern* is an optional selector of the extracted string.

The operation is illegal on an interface that has been associated with a server socket or on a device interface not declared as "readable" (section 4.4.1).

The operation never blocks waiting for data to arrive from the port or device. The input buffer of the interface is filled asynchronously, up to the declared limit (section 4.4.1) as soon as new data becomes available. If `extract` fails to extract the string from the current contents of the buffer, it returns 0 and *where* is not set. Otherwise, `extract` returns 1 and *where* is set to the extracted string.

If *pattern* is an empty string (or is not specified at all), the entire contents of the interface's buffer are extracted and the buffer is emptied. If the buffer is empty, the operation fails and returns 0.

If *pattern* is an integer number, it specifies the number of bytes to be extracted from the buffer. If the buffer contains that many bytes, its initial portion of the specified length is removed and stored in *where*, and `extract` returns 1. Otherwise, i.e., if the buffer is shorter than the specified number, it is left intact and `extract` returns 0.

If *pattern* is neither empty nor it looks like an integer number, it is treated as a regular expression to be matched against the contents of the buffer. If there is no match, the operation fails and returns 0. Otherwise, the first matching portion of the buffer is stored in *where*. The entire initial fragment of the buffer (starting from the first byte) up to the last byte of the first match is removed from the buffer. Note that if the matching string does not fall on the beginning of the buffer, the initial (umatched) portion is discarded and ignored.

If the interface type defines a method called `input`, the method is used to preprocess the data acquired from the external agent associated with the interface, before those data are stored in the interface's buffer (section 4.3.1). Every chunk of data acquired asynchronously from the port or device is passed through the `input` method before being stored in the buffer. It can be preprocessed in an arbitrary way, in particular it can grow or shrink. If it grows, then the limit on the number of bytes that can be stored in the buffer at a time (section 4.4.1) may be exceeded, because whatever is returned by `input` is appended at the end of the buffer regardless of its length.

Note that the value of *pattern* is interpreted in the context of the preprocessed data rather than the original data arriving from the interface's external agent.

Example

Below we list the `input` method defined for an interface representing the serial controller for X10 modules.

```
method X10Monitor input { b } {
  upvar $b buf
  binary scan $buf H[expr 2 * [string length $buf]] buf
}
```

As we can easily see, the method converts binary strings arriving from the controller to their hexadecimal character representations. Every arriving message is expanded by the factor of two.

It isn't difficult to guess that this interface also declares an `output` method that works in the opposite direction. As a matter of fact, it is a bit more complicated because it also calculates a checksum required by the controller:

```
method X10Monitor output { buf } {
  upvar $buf b
  useown CheckSum
  set CheckSum 0
  set N [string length $b]
  for { set i 0 } { $i < $N } { incr i 2 } {
    incr CheckSum 0x[string range $b $i [expr $i + 1]]
  }
  set CheckSum [expr $CheckSum & 0xff]
  set b [binary format H[string length $b] $b]
}
```

4.4.3 Special operations

An interface can be destroyed by executing `delete` on the handle. This operation not only deallocates the interface object, but it also closes the socket or device associated with the interface. It is possible to close the interface's agent without destroying the interface with the following operation:

`shutdown` *interface*

where *interface* is the interface handle. After this operation, the interface becomes unusable and it should be destroyed. One reason why sometimes it may make sense to separate the two operations, i.e., closing and destroying, is to make sure that multiple processes perceiving the status of the same interface do not attempt to reference an invalid handle.

The following function:

`closed` *interface*

returns 1 if the interface's agent has been closed, and 0 otherwise.

Note that an interface's port or device becomes closed not only if we perform `shutdown` on it, but also in consequence of hitting the end of file (e.g., on a socket that has been closed by the peer) or an i/o error. Therefore, by checking the interface status, e.g., before attempting an i/o operation, we make sure that we are not beating a dead horse. Although `deposit` on a shut down interface will indicate a failure (section 4.4.2), `extract` will only tell us that it has failed to extract the string.

The usual i/o operations are not applicable to server interfaces, although those interfaces can be destroyed, shut down, or checked for being `closed`. The sole purpose of a server interface is to signal incoming connections. With the following operation, it is possible to receive a new connection from a server interface:

`client` *interface ctype bufsize*

If there is no pending connection on the server interface pointed to by the first argument, the operation returns an empty string. Otherwise, it creates a new interface of class type *ctype* and returns its handle. The last (optional) argument (which defaults to 256) gives the buffer size for the new interface. The implicit *how* parameter of the new interface (section 4.4.1) is of course `internet+client`. It represents our end of a peer-to-peer socket.

4.4.4 Interface events

The non-blocking i/o provided by interfaces needs a collection of events to indicate when it becomes possible. All those events are listed below.

`Newitem`
: This event occurs when at least one new byte appears in the interface's input buffer and is available for extraction. `Newitem` never occurs immediately, e.g., if the buffer is nonempty when the wait request is issued. The buffer contents must change for the event to occur.

`Receive`
: This event occurs when the input buffer is or becomes nonempty. The entire buffer contents are returned in `Message` when the event is presented to the waiting process, and the buffer is cleared. `Receive` is very similar to its mailbox counterpart (section 4.3.3).

`Close`
: This event occurs when the interface is closed and the buffer is empty. It will occur immediately, i.e., at the moment when the wait request is issued, if these conditions are met.

`Client`
: This event occurs on a server interface when there is a pending connection request. A process receiving this event may execute `client` (section 4.4.3) to accept this connection.

An integer number

This event (the so-called *count event*) occurs when the length of the string in the input buffer is (or becomes) greater than or equal to the specified number. Note the difference with respect to the mailbox count event (which only occurs on the exact count). In particular, count 1 for an interface means "nonempty," which is the reason why there is no `Nonempty` event for interfaces.

Anything else

Any other string appearing as the event identifier (except "`Attention`" (section 4.5) is interpreted as a regular expression. This event (the so-called *pattern event*) is triggered if the contents of the input buffer match the expression.

The contents of `Message` (section 3.3.2) when a process is restarted by an interface event depend on the event. For `Newitem` and a count event, `Message` returns an integer number indicating the number of bytes in the buffer. For `Receive`, `Message` returns the entire contents of the buffer (at the moment when the event was triggered). For a regular expression, `Message` returns the portion of the buffer that has been matched. For `Close`, `Message` returns an empty string.

When an interface is closed and there is a process waiting for `Newitem` on the interface, the process is restarted with `Message` returning the number of bytes currently in the buffer. The `Receive` event is also triggered in such circumstances, with `Message` containing an empty string. Thus, a process awaiting `Receive` must be prepared to receive nothing, which normally indicates the end-of-file condition on the device or socket, but may also signal an error. The actual `Close` event is only triggered if the input buffer happens to be empty. For as long as the buffer contains some bytes, those bytes can be extracted from the interface in the standard way, even though the interface's agent has been closed and will send no more data. A regular expression event (but no count event) on a closed interface is also triggered (but only if the buffer is empty) with `Message` containing an empty string.

Example

Assume that we would like to implement an "addition" server that will receive integer numbers from clients and calculate their sums. A client may submit a list of integer numbers separated with blanks, with a period at the end of the list. The server will compute the sum of all those numbers and return it to the client, closing the interface after that operation.

We start from the process responsible for accepting connections. This is how it may look:

```
class MPort
interface MPort
```

```
class AServer { port } {
  useown Port
  set Port [create MPort internet+server $port]
}
process AServer {
  useown Port
  set nc [client $Port MPort]
  if { $nc != "" } { create Adder $nc }
  wait $Port Client
}
```

The above sequence illustrates the standard way of receiving connections on a server interface. The `AServer` process creates a server interface (of type `MPort`) representing a master socket listening on the specified port. The process's code method attempts to acquire a connection by executing `client` (section 4.4.3). If the operation succeeds, `nc` points to a new interface (its type is also `MPort`) which represents our end of the client's socket. Then `AServer` creates a new process of type `Adder` passing it the new interface. That process will be responsible for handling the new request.

Before putting itself to sleep (i.e., returning from the code method), the process issues a wait request to the server interface—to be awakened when there is a new connection request. Note that if there already is a pending connection request, the event will be triggered immediately and the code method of `AServer` will be re-executed.

Below we list our implementation of `Adder`.

```
class Adder { cl } {
  useown Client Sum
  set Client $cl
  set Sum 0
}
process Adder {
  useown Client Sum
  if [extract $Client num {^[1-9][0-9]*[ .]+}] {
    regexp {([0-9]*)(.*)} $num ign num sep
    incr Sum $num
    if { [string first "." $sep] >= 0 } {
      deposit $Client ${Sum}\n
      delete $Client
      sleep
    }
  }
  if [closed $Client] {
    delete $Client
  } else {
```

```
            wait $Client {[1-9].*[ .]}
        }
    }
```

Also this time we can get away with a single-state code method. Upon creation, the process initializes the sum and then attempts to extract a number from the interface's input buffer. To be complete, this number must terminate with a space or period (in fact, we admit multiple spaces/periods at the end of a number). The `Sum` is incremented by the new number and if the delimiter contains a period (indicating the end of sequence), the `Sum` is sent to the client and the interface is closed. The process terminates because it goes to sleep without specifying a single wait request. The newline character at the end of output makes it look better, e.g., when our simple server is tested using `telnet`.

If the extraction fails, or if more numbers are expected, the process issues a wait request to the interface to be awakened as soon as a new number appears in the input buffer. Note that before issuing the wait request the process checks if the interface is not closed. This would indicate an abnormal condition (i.e., misbehaving or dead client) and the process would just destroy the interface and terminate itself without completing its service.

To turn the above code into a complete program, we have to insert

```
package require sicled 1.0
sicle
```

in front of it, and

```
create AServer 5055
kernel
```

at the end. Of course, `5055` is just a sample port number, which can be replaced with another legitimate (and unused) port. It is easy to test the server via `telnet` by connecting to port 5055.

4.5 The attention event

One event is common for all activity interpreters and is triggered on all of them in the same way. This event is `Attention` and it occurs when somebody executes

`attention` *ai message*

on the *AI*. The first argument is an *AI* handle and the second is the optional string to be returned in `Message` (section 3.3.2) when the event is presented to a process. The role of Attention is to indicate special (programmable) conditions that may occur on activity interpreters, which are not covered by the standard semantics of the *AI*s.

5 Sensors

SICLE offers a simple collection of tools for defining logical sensors and actuators, interfacing them to physical sensors and actuators, and writing programs operating on them. Among those tools is a built-in server process that allows remote clients to connect to a SICLE program and perform sensor operations. Using the server, it is also possible to upload so-called *actions* into SICLE programs. Actions can be viewed as dynamically loaded processes identifiable from the outside and triggered by some conditions.

The sensor tools offered by SICLE are intentionally extensible. Their present collection includes X10 modules (accessed via the CM11A serial controller), SDS sensors and actuators (accessed via an SDS interface daemon provided as a separate program), and simulated sensors and actuators.

To make the sensor tools available to SICLE processes, the `sicle` operation (section 3.1) must be called with a parameter including the string `sensors` as its part. This will force the program to read the so-called sensor map (the contents of a special file) and launch the sensors server process listening for client connections on a dedicated port.

5.1 Sensor map

The default name of the sensor map file is `smap.txt`. The file is sought in the directory in which the SICLE program was called, if the argument of `sicle` (section 3.1) contains the string `sensors`, i.e., the sensor component of SICLE is enabled. This default name can be changed with the `-s` call parameter of the SICLE program, e.g.,

```
mypgm -s mysensormap.map
```

or

```
tclsh mypgm -s mysensormap.map
```

Call arguments of a SICLE program are discussed in section 6.1.

A sensor map consists of textual entries describing the mapping of logical sensors used by the program. Typically, each entry occupies one line.

Example

Below we list the contents of a sample sensor map file for a program driving three X10 modules.

```
Tag: TV     Class: X10  Value: U  Params: A1,nodim,/dev/ttyS3:4800,n,8,1
Tag: Lamp1  Class: X10  Value: U  Params: A2,dim,/dev/ttyS3:4800,n,8,1
Tag: Lamp2  Class: X10  Value: U  Params: A3,dim,/dev/ttyS3:4800,n,8,1
```

A complete entry for one sensor/actuator consists of four fields named `Tag`, `Class`, `Value`, and `Params`. These fields must occur in the listed order and they must be preceded by the headers (i.e., their names terminated by colons).

The `Tag` field specifies the symbolic name via which the sensor will be visible to the program. This can be any string that does not include blanks, tabs, or newlines. Needless to say that sensor names should be unique within a program (map file).

The `Class` field identifies the sensor type. The following types are legal in the present version of SICLE: `X10` (an X10 module), `S` (a simulated sensor/actuator), and `SDS` (an SDS sensor/actuator).

The `Value` field tells whether and how the sensor/actuator values are to be mapped, i.e., transformed into other values. The whole purpose of the sensor map file is to make all sensors (possibly with diverse, vendor-specific characteristics) appear in a uniform way to the SICLE program. This may involve remapping the values assumed by the physical sensors/actuators, so that their logical values (used by the program) look uniform.

The letter `U` occupying the `Value` field in the above example means that the sensor's values are not remapped. In fact, any string that does not contain "`=`" has the same meaning. A remapping is only effected if the `Value` field has the following format:

l_1=r_1,l_2=r_2,...

where each l_i identifies a logical value (as perceived by the program) and each r_i identifies an actual (real) value (as perceived by the sensor/actuator). If l_i is symbolic (i.e., it doesn't look like a nonnegative integer number), it defines a single discrete mapping of a symbolic value (used by the program) into a numerical (or possibly symbolic) value understood by the sensor. If both l_i and r_i are numeric, there must be exactly one other pair l_i, r_i with this property. The two pairs specify a linear mapping between logical and physical numeric values, carried on according to the following formulas:

$$R = (l_i-L)*(r_j-r_i)/(l_j-l_i) + r_j$$
$$L = (r_i-R)*(l_j-l_i)/(r_j-r_i) + l_j$$

where L is the logical value and R is the "real" value.

The `Params` field is specific to the sensor type and is intended to give the physical coordinates of the sensor within its domain. For an `S`-type (simulated) sensor, which has no physical counterpart, the `Params` field is ignored (although it must be specified).

For an X10 module (visible via the X11A serial controller), the `Params` field specifies all the parameters needed to locate and address the module within the home network. The field consists of the following subfields separated by commas (in the order of occurrence):

1. The X10 address of the module. This address consists of a letter `A-P` identifying the "house" and a number `1-16` identifying the unit (module) within the house.

2. `dim` or `nodim`, depending on whether the module can be dimmed or not.

3. The serial device representing the X11A controller driving the module. This field is in the format accepted by the interface constructor (section 4.4.1). In particular, it may include the specification of the serial parameters for the device (for X11A, they should be "`4800,n,8,1`").

For an SDS sensor/actuator, the `Params` field consists of the following six fields separated by commas (in the order of occurrence):

1. the name of the host on which the SDS daemon is running
2. the TCP/IP port on which the daemon is listening
3. `S` or `A`, depending on whether the unit is a sensor or an actuator
4. the SDS board number
5. the SDS network number
6. the SDS unit number within the network

The last three fields can be viewed as the physical coordinates (SDS network address) of the sensor/actuator.

Example

Below we list the contents of a simple map file describing one SDS sensor and one actuator.

```
Tag:                        Switch
Class:                      SDS
Value:                      ON=1,OFF=0
Params:                     sheerness.cs.ualberta.ca,2286,A,0,1,2
Tag:                        Bulb
Class:                      SDS
Value:                      ON=0,OFF=1
Params:                     sheerness.cs.ualberta.ca,2286,S,0,1,1
```

Note the mapping of the binary values taken by the two units into symbolic names.

5.2 Working with sensors

From now on, by "sensor" we will mean "sensor or actuator." If we need to distinguish between the two, we will say it explicitly.

Sensors are represented by SICLE objects. Each sensor class (simulated, X10, SDS) has its own sensor type (`SSensor`, `X10Sensor`, and `SDSSensor`, respectively). There is no need to know these types—all sensor operations can be carried out in a standard way independent of the sensor class. For the record, `SSensor` and `X10Sensor` are implemented as mailboxes (section 4.3) while `SDSSensor` is an interface type (section 4.4).

5.2.1 Identifying sensors

A sensor can be referenced by its object handle or by its symbolic name assigned in the sensor map file. The following function returns a sensor object handle, given the sensor's symbolic name:

`sensor` *name*

When called for the first time with a given (legitimate) sensor name, the operation creates the sensor object (based on the description in the sensor map file) and returns its handle. With subsequent calls, `sensor` just returns the handle of the existing named sensor.

The following argument-less function:

```
sensors
```

returns the list of all sensor names known to the program (based on the sensor map file).

5.2.2 Operations on sensors

The primary attribute of a sensor is its value, which may be a number or an arbitrary string, possibly mapped from a physical value (section 5.1). This value is available directly as attribute `Value`, e.g., given a sensor handle, the following operation:

`getattr` *handle* `Value`

will return the sensor's value.

Although for the presently implemented sensor types the above method of reading the sensor's value is going to work, it is recommended not to use the `Value` attribute directly. Certainly, it is not recommended to set `Value` using `setattr` (section 2.5.3). Instead, the following two methods (section 2.5.1) of the sensor types should be used for these purposes:

`getValue`
`setValue` *newvalue*

Note that usually, when the value of a sensor (or actuator) is set, some specific action must be carried out, e.g., the new value must be propagated to the physical counterpart of the actuator, processes waiting for the new value of the sensor should be awakened, etc.

By always using the above two methods to reference sensor values, we make sure that all those operations are performed as needed.

Example

Consider the X10 sensors described by the sample map from section 5.1. The following sequence of operations:

```
invoke [sensor TV] setValue on
invoke [sensor Lamp1] setValue off
invoke [sensor Lamp2] setValue 50
```

switches on the TV, turns off one lamp, and sets the other one at 50% of its maximum brightness.

It is also possible to perform operations on sensors using their names rather than object handles. The following function:

`sense` *how value s1 ... sn*

checks the values of a group of sensors identified by their names *s1 ... sn*. If *how* is `any`, the function returns 1 if *any* of those sensors has the specified value. The only other legitimate value of *how* is `all`. The function returns 1 if *all* the sensors have the specified value.

Another operation that uses sensor names rather than handles is

`setall` *value s1 ... sn*

which sets all the listed sensors to the specified value.

Example

Consider the following command involving a collection of X10 modules:

```
if [sense any on Motion1 Motion2] {
  setall on Alarm1 Alarm2 Alarm3
}
```

If any of the two modules (probably motion detectors) is `on`, the three alarm switches are all set `on`.

5.2.3 Sensor events

Sensors may be implemented internally as objects of different types, but all these types are activity interpreters. In fact, there is only one event of interest for a sensor. This event is `Attention` (section 4.5), which is triggered whenever the value of a sensor is changed.

Example

Below we list a simple process monitoring a set of motion detectors and switching on and off the lights.

```
class MMonitor { slist alist } {
  useown SList AList TenSeconds
  global Second
  set SList $slist
  set AList $alist
  set TenSeconds [expr $Second * 10]
}
```

The first constructor argument is a list if sensors to be monitored. If any of those sensors goes off, we will switch on all the modules from the list represented by the second argument. Then every 10 seconds we will switch those modules on and off until at least one of the sensors remains on. These operations are accomplished by the following code method:

```
process MMonitor {
  useown State SList AList TenSeconds
  global Timer
  state Start {
    if [eval sense any on $SList] {
      eval setall on $AList
      wait $Timer $TenSeconds Quiet
    } else {
      foreach s $SList { wait $s Attention GoOff }
    }
  }
  state Quiet {
    eval setall off $AList
    wait $Timer $TenSeconds Start
  }
}
```

The use of `eval` is necessary because the sensor lists for `sense` and `setall` must be explicit, i.e., every sensor must be specified as a separate argument.

Note that the `Attention` event on a sensor (or any other *AI* for that matter) is only triggered at the moment when somebody executes `attention` on the *AI* (section 4.5). The attention condition is not queued if nobody is waiting for the event when the operation is executed. Consequently, to perceive a change of a sensor's value, a process has to make sure that the sensor is never left "unattended."

Example

Consider the following process code method:

```
process SensorTrace {
  useown State Sensor
  switch $State {
    Start {
      wait $Sensor Attention NewValue
    }
    NewValue {
      puts stdout "New value: [getValue $Sensor]"
      proceed Start
    }
  }
}
```

which may seem to exemplify a natural way of tracing the value of a sensor. Note, however, that some value change (attention) events may be lost because there are moments when the sensor is not monitored for this event. These moments occur when the process transits from state `Start` to `NewValue` (things can happen while that transitions is being scheduled) and from `NewValue` back to `Start` (`proceed` is not a direct "goto"—section 3.3.3—but involves scheduling). Note that the following version:

```
process SensorTrace {
  useown State Sensor
  switch $State {
    Start {
      monitor $Sensor Attention NewValue
    }
    NewValue {
      puts stdout "New value: [getValue $Sensor]"
    }
  }
}
```

is much better. Technically the sensor's value may still change while the process is being awakened (after the event is triggered and before the process is scheduled to run in state

NewValue). However, there is no universal remedy for this, because there is always a certain time threshold such that a condition arising for less than that time is not perceptible. To reduce the duration of this threshold in the above code, the monitor operation can be issued at high priority (section 3.3.2).

The following single-state version of the process also does a good job:

```
process SensorTrace {
  useown Sensor OldValue
  set nv [getValue Sensor]
  if { $nv != OldValue } {
    puts stdout "New value: $nv"
    set OldValue $nv
  }
  wait $Sensor Attention
}
```

To reduce the length of the sensitivity threshold in the above code, the wait operation can be rewritten as

```
wait $Sensor Attention "" high
```

One important property of the last two versions of SensorTrace, in contrast to the first version, is that the printed value of the sensor is always the actual current value. Suppose that the sensor has two binary values, e.g., on and off, and consider the first version of the process. Imagine that the sensor turns on, which event triggers a transition from Start to NewValue, but during the transition from NewValue to Start, the sensor turns off. This change will be completely lost and the last recorded value of the sensor will be on. This cannot happen with the last two versions. Although some value changes can be lost, the recorded value is actually the current value of the sensor.

5.2.4 Legitimate values of standard sensors and actuators

Sensor values can be mapped into arbitrary numbers and strings (section 5.1), but to do this mapping we have to know something about the legitimate values assumed by the physical sensors.

Simulated sensors are not interfaced to any physical objects: their values can only be set and perceived by the program. Therefore, the value of a simulated sensor/actuator can be an arbitrary string and it is never mapped.

For an SDS sensor/actuator, the situation is also quite simple. The unmapped value of such a sensor/actuator is a signed integer number between -32768 and 32767, whose interpretation is specific to the given sensor model.

For an X10 module, the situation is a bit more complicated, owing to the following two facts:

- Some X10 modules are binary (on/off) while some others are *dimmable*.
- It is not practically possible to keep track of the actual “value” of a dimmed module.

The physical interpretation of “setting the value of an X10 module” is that of “sending a command to the module.” For example, one can dim or brighten a module by a given number of units (and a dimmable module will respond to this command) but there is no direct interpretation of the resultant value of the module actuator. The actual response of various modules to such commands is somewhat uneven and random.

The home network of X10 modules is visible to SICLE via a controller (X11A) that creates some perception of the status of the various modules. X10 sensors and actuators, as perceived by a SICLE program, are implemented by examining and changing this perception. There is no clear distinction between a sensor and an actuator: every X10 module has a potential to behave both ways. For example, a motion detector is technically a sensor, i.e., its value is changed by some physical actions, but this value can also be set by the program. This operation will actually set the value of the controller’s perception of the motion detector (the motion detector itself has no means to be affected), but, for all practical purposes, the net effect is the same as if the status of the motion detector has actually changed.

For a binary X10 module (declared as non-dimmable in the sensor map file—section 5.1), its perceived (unmapped) value is one of the strings `on`, `off`. These are the values that will be returned by `getValue` (section 5.2.2) for such a module.

The value of a dimmable module (as returned by `getValue`) can be an integer number between 0 and 100 (representing the percentage of brightness), or a string `on`/`of` indicating that the module is completely on or off.

The argument of `setValue` may take one of the following forms:

`on`
: The module is turned on. The value stored in the module is `on`.

`off`
: The module is turned off. The value stored in the module is `off`.

possibly signed integer number between -100 and 100
: If the module is not dimmable, zero is treated as `off`, anything else is treated as `on`. For a dimmable module, a negative number is treated as a request to dim the module by the specified percentage while a positive number is treated as a request to brighten the module by the specified percentage. The resultant value of the module depends on the previous value and it is a number between 0 and 100.

`on` *directly followed by a number between -100 and 100*
: For a non-dimmable module, this is equivalent to `on`. For a dimmable module, this is a request to set the module to the indicated absolute brightness

percentage. If the specified brightness is negative, it is subtracted from 100. The module is first turned off, then (fully) on, and finally it is dimmed by 100 minus the specified brightness. The resultant value of the module is equal to the specified brightness.

Incremental dimming and brightening changes the value of the module by incrementing it or decrementing by the specified count, but never below 0 or above 100. Usually, this value gives only a very approximate idea of the actual state of the module and should be used with a large grain of salt. Note that the value of a module that has been brightened/dimmed is never `on` or `off`, even if it was `on` when the module was brightened or `off` when it was dimmed. A module in this state can be turned `on` or `off`, and then its value will become `on` or `off`, respectively.

Example

Below we list a few sample sensor commands addressed to X10 modules.

```
invoke $tv setValue on
setall on50 Lamp1 Lamp2 Lamp3
invoke $lamp setValue -20
```

Note that `Lamp1`, `Lamp2`, and `Lamp3` are symbolic names of modules whereas `$tv` and `$lamp` are sensor handles. Naturally, all we said about the legitimate format of a `setValue` argument also applies to the first argument of `setall` (section 5.2.2).

5.3 The action loader

When the sensor mechanism of SICLE is switched on (section 3.1), the program automatically configures a server, visible as a TCP/IP port, that can be used to perform remote operations on sensors, check their status, or even upload into the SICLE program new processes (actions) dynamically. This is why the server is called the "action loader," although in most cases it is used to pass simple commands rather than load actions. The commands accepted by the server look like lines of texts which can be comfortably entered by hand, e.g., using `telnet`. The standard port number of the server is 3766. It can only be changed by editing a constant in the package header.

5.3.1 Authentication and session format

The action loader can handle multiple session at the same time. Immediately after receiving a connection request, the loader sends the following line of text to the client:

```
SICLE Action loader ver listening:
```

where *ver* stands for the version number of the package. This line indicates that a connection has been established: the loader is ready to accepts commands.

The first line entered by the client is ignored. Together with the beginning of the next (second) line, the action loader uses it to determine the client's convention for the *end-of-line* sequence. The second line arriving from the client should be the authentication line with the following contents:

`auth` *username password*

where the arguments should identify a user authorized to ask the loader for service. These arguments will be matched against the contents of the file named `users` that should be present in the directory in which the SICLE program was called. It consists of lines of texts, each line containing a pair of tokens: a username and a DES-encrypted password with "salt"—in the convention of encrypting passwords in most versions of UNIX. For example, the `users` file for the SICLE program controlling my home lights and appliances consists of a single line that looks like this:

```
pawel Jh3GyUKVNS4ps
```

If the `users` file defines more than one user, all these users have the same rights, except for accessing actions (section 5.3.3). If the `users` file is absent or the specified user name and password do not match one of the lines in `users`, the authentication fails and the loader closes the connection.

One way to create an entry in the `users` file is to copy a user name and the associated encrypted password from a UNIX `passwd` file. Some other commonly available tools, e.g., the `htpasswd` program that comes with web server's, can also be used for this purpose. SICLE offers the following two functions that can be used for implementing DES-based authentication:

`crypt` *plaintext*
`pmatch` *encrypted plaintext*

The first function takes a plaintext string and returns its DES-encrypted version with salt, to be used as a stored password. Upto eight initial characters of the *plaintext* string take part in this operation. The second function returns 1 if the specified *plaintext* string matches the *encrypted* version, and 0 otherwise.

Example

To encrypt a password, e.g., for an entry in the `users` file, you can type the following sequence of commands:

```
tclsh
package require siclef 1.0
crypt trykowka
```

The produced string is a “salted” encrypted version of the specified password which can be used directly in a `users` file entry.

In response to `auth`, if the authentication was successful, the action loader outputs the following line:

```
990 Authentication OK
```

and becomes ready to accept other commands. Otherwise, it diagnoses the problem, e.g.,

```
001 Authentication failed
```

and terminates the connection.

Note that the action loader will close a session in progress after **10 minutes of inactivity** (no commands) on the client’s part.

All replies from the loader start with a sequence of three digits identifying the reply, followed by a text explaining what has happened. This way, the replies can be easily processed by a program and, at the same time, are legible to a human user communicating directly with the loader. A similar approach is used, e.g., in SMTP.

If the first of the three digits of a message code is `9`, it indicates a successful execution of the last command. Sometimes a message will carry additional information, i.e., a reply to the last command, following the message code. In all cases, the message code determines the format of that reply.

If the first of the three digits of a message code is not `9`, it indicates a special message which may be an error message or a status report (code `077`—section 5.3.2). Different error messages have different message codes, which we will list with the commands.

5.3.2 Simple commands

Below we present the basic commands accepted by the action loader (the ones that do not deal with actions). Loading and monitoring actions is the subject of section 5.3.3. Each command takes a single line and terminates with the end-of-line sequence appropriate for the connecting client.

`turn` *sensor value*

This command sets the value of the indicated sensor/actuator. Of course, the sensor must be specified by its name—as assigned in the map file (section 5.1). Possible replies:

```
997 OK
009 No such sensor
010 Sensor tag and value required
011 Failed
```

The third message indicates a command format error, the last one is returned when `setValue` for the sensor is aborted for whatever reason.

`show` *sensor*

This command returns the current value of the named sensor. Possible replies:

```
996 Value:  value
008 Sensor tag required
009 No such sensor
```

`note` *how s1 ... sn*

This command sets up or cancels *notifiers* for the indicated sensors. If no sensors are specified, i.e., the *s1 ... sn* part does not occur, the command refers to all sensors defined in the map file (section 5.1).

If *how* is `add`, it means that the client wants to be informed whenever the value of any of the indicated (or all) sensors changes. If this happens, the loader will send the following status message to the client:

`077` *sensor value*

where *sensor* is the symbolic name of the sensor whose value has changed, and *value* is the new value.

Note that following a `note` command, the client must be prepared to receive asynchronous messages from the action loader that will arrive spontaneously rather than as responses to explicit commands.

If the *how* argument is `cancel`, the command removes the notifiers for the specified (all) sensors. Following this command, no more update messages regarding the indicated sensors will be arriving from the loader.
Possible replies:

```
997 OK
012 Notifier action required
013 Invalid notifier operation
```

Both error messages indicate formal errors in the command, e.g., the first argument being something different from `add` or `cancel`. A non-existent sensor appearing on the argument list is silently ignored without raising an error condition.

`idle`

This command takes no arguments and it does nothing (it triggers no reply). Its sole purpose is to indicate to the loader that the client is still alive. Note that if the client doesn't issue a command for 10 minutes, the loader will close the connection.

```
exit
```

This command terminates the session and closes the connection. There is no reply.

Example

Below we list a sample telnet session with the action loader. User typed-in commands are in teletype font while the system replies are in italics.

```
telnet sheerness.cs.ualberta.ca 3766
SICLE Action loader 1.0 listening:
Hi there!
auth pawel dyrdymala
990 Authentication OK
turn tv on
009 No such sensor
turn TV on
997 OK
show TV
996 Value: on
note add Motion1 Motion2
997 OK
077 Motion1 off
077 Motion2 off
idle
idle
077 Motion1 on
077 Motion1 off
077 Motion2 on
idle
077 Motion2 off
turn TV off
997 OK
077 Motion1 on
note cancel
997 OK (2)
exit
Connection closed by foreign host.
```

Note that the first line accepted by the loader (`Hi there!`) is ignored. Following the `note` command, the loader immediately responds with the current status of the indicated sensors and then only reports the changes. The `OK` reply after `cancel` gives in parentheses the number of canceled notifiers.

5.3.3 Actions

Action loading is best performed from a program, but in principle it can be done manually, e.g., in a telnet session. An action is essentially a SICLE process (or rather a process code method) together with a *trigger condition.* The trigger condition specifies when the action should become active.

Let us have a look at how an action is loaded. This operation is started by the client with the following argument-less command:

```
load
```

In response, the loader sends the following line (the only possible reply):

```
999 Proceed
```

which indicates its readiness to accept the action.

Following this reply, the loader will be accepting the subsequent lines arriving from the client as components of the loaded action, until it encounters a line containing `.` as the only character. This character will not be stored as part of the action, but it will terminate the load operation and revert the loader to the command mode.

The action body should look like the interior of a hypothetical process code method, i.e., something that could be sensibly put between

```
process ... {
```

and the matching closing brace. The action may assume that the `State` attribute of the encapsulating process is already visible and that its first value will be `Start` (as for a regular process). No other attributes are provided by default, but of course the action can create them and reference (e.g., with `useown`) as needed.

The action body may (but doesn't have to) be preceded by a *trigger*, i.e., a condition that will start up the action. If a trigger is present, it should be separated from the action body by `+++` (three consecutive pluses). A trigger-less action is started immediately after being loaded.

Example

Suppose that we would like to load an action that on every Saturday at 9:00 p.m. will launch a watchdog monitoring two motion detectors and raising an alarm when any of them goes off. The watchdog will deactivate itself in three hours, i.e., at midnight. This is a complete `telnet` session that does the job (system responses are in italics).

```
telnet sheerness.cs.ualberta.ca 3766
SICLE Action loader 1.0 listening:
Hi there!
auth pawel dyrdymala
990 Authentication OK
load
999 Proceed
[cdate match "Sat .*21:..:"]
+++
useown Until M1 M2 Alarm
global Hour FiveMinutes Timer
switch $State {
    Start {
        set Until [expr $Hour * 3]
        set M1 [sensor Motion1]
        set M2 [sensor Motion2]
        set Alarm [sensor AlarmSwitch]
        monitor $Timer $Until Done
        proceed Watchdog
    }
    Watchdog {
        if [sense any on Motion1 Motion2] {
            proceed GoOff
        } else {
            wait $M1 Attention Watchdog
            wait $M2 Attention Watchdog
        }
    }
    GoOff {
        invoke $Alarm setValue on
        wait $Timer $FiveMinutes SwitchOff
    }
    SwitchOff {
        invoke $Alarm setValue off
        proceed Watchdog
    }
    Done {
        invoke $Alarm setValue off
        hibernate
    }
}
.
```

```
991 ec8
exit
Connection closed by foreign host.
```

After an action has been successfully loaded, the action loader returns to the client code `991` (as in the above example) followed by a piece of text representing the action's handle. Using this handle, it is possible to monitor the action status or kill the action when it is no longer needed. An action loaded by one user (section 5.3.1) can only be monitored and killed by the same user.

If the loaded action appears syntactically incorrect, the loader will send the following line to the client:

```
005 Action failed: message
```

and ignore the action. The text after the colon will explain the problem.

The trigger condition for the action from the above example is described by a call to `cdate`. The general format of this call is

```
cdate how date
```

where *how* determines how the following *date* argument is interpreted. If *how* is `match`, the *date* argument is a regular expression pattern to be matched against the date as returned by the following expression:

```
[clock format [clock seconds]]
```

For example, this is what the above expression returned at the time when I was writing this line: `Wed Oct 14 13:11:10 MDT 1998`. If the current date matches the specified regular expression, `cdate` returns 1, otherwise it returns 0.

The remaining legal values of *how* are: `earlier`, `later`, `notearlier`, and `notlater`. In all these cases, the *date* argument should be a valid textual representation of date/time as recognized by `clock scan`. The function returns 1 or 0 depending on whether the current date is earlier, later, not earlier, not later than the specified date.

Trigger conditions for actions are checked every 10 seconds. Note that in principle such a condition can be any Boolean expression.

An action can terminate (as a regular process), or it can `hibernate` (as the above action). A terminated action is removed from the set of actions and deallocated. A hibernated action becomes dormant awaiting another occurrence of the trigger condition. An action that has terminated, although it is deallocated and removed, leaves its trace in the loader until the terminated status of the action can be presented to the user.

An action that hits an error during its execution is automatically (and gracefully) terminated without crashing the loader. Note, however, that in the present prototype version of SICLE, actions are not separated from their environment in an absolutely foolproof way. Therefore, it is possible for a malicious action to damage the state of the loader in an unrepairable way, although this seems to require destructive intentions on the user part.

Besides `load`, two other commands of the action loader deal with actions. One of them is `stat`. If if used with no argument, it polls the loader for the status of all actions loaded by the authenticated user. In reply, the loader sends a line looking like this:

```
995 (h1: status count) ... (hn: status count) end
```

where the first element of each parenthesized sequence is an action handle, and the remaining two element indicate the current action status (1–active, 0–hibernated), and the number of times the action has changed its status from hibernated to active (on the trigger condition). For a terminated action, *status* and *count* are replaced with a message indicating how the action terminated. If this message is `done`, it means that everything went fine, otherwise the message gives a description of the error condition.

The optional argument of `stat` can identify a single action (by its handle, as returned by the loader when the action was submitted). In that case, the loader returns one of the following replies:

```
007 No such action
008 Not owner
994 status count
992 Action terminated: message
```

In the last case, *message* is either `done` or it describes the error condition that terminated the action.

Example

As soon as the status of a terminated action has been presented to the user, this status information is removed from the loader. This is illustrated by the following two inquiries:

```
stat ec8
992 Action terminated: done
stat ec8
007 No such action
```

Actions can be forcefully terminated by the `kill` command which handles one action at a time. Its mandatory argument identifies the action to be killed. In response, the loader will send one of the following lines:

```
006 Action handle required
007 No such action
008 Not owner
992 Action terminated
993 Already terminated: message
```

The last response is produced when the action is already terminated (with its completion status pending) when the kill command is issued. The pending completion status of the action is erased, as after `stat`. Note that if an action is terminated with `kill`, its completion status is not remembered, i.e., a subsequent `stat` call will fail to locate the action.

6 Additional features

In this section we list some of the more obscure (and perhaps less useful) features of SICLE which have not been mentioned in the preceding sections.

6.1 Program call arguments

A SICLE program accepts two standard program call arguments, which are interpreted by the `sicle` function (section 3.1). One of them is `-s` followed by a file name (section 5.1) specifying a non-standard name of the sensor map file. This argument is only relevant if `sicle` has been called with an argument containing `sensors` as a substring (section 3.1).

The other argument is `-d` optionally followed by a file name, `-` or `--`. This argument identifies the input file to be used by the program (section 6.2). If `-d` never occurs in the argument list, no standard input file is opened, although, of course, the program may still read whatever data it pleases using its private means.

If `-d` is followed by a double `-`, or if it isn't followed by anything, the input file is named `data.txt`, which can be viewed as the standard name of the input file. If `-d` is followed by a single `-`, the input file is equivalenced with the standard input.

The call arguments interpreted by `sicle` are terminated by a double `-` (or by the end of the argument list). If the program wants to use some private call arguments, they should be specified (and sought) after the double `-`.

6.2 Input file

The input file (section 6.1) is intended to provide the SICLE program with some data, other than the sensor map (section 5.1). The input file (if one has been specified in the program call line—section 6.1) is opened by `sicle` and closed by `kernel` (section 3.1); thus, it can only be used in the setup phase, before the initial processes created by the program are started. The file cannot be read directly, but only with the following SICLE functions:

`skipinput` *pattern*

The argument is a regular expression pattern. The function skips the input file until the first line containing a string that matches the specified expression. Then, the file is positioned at the first character following the matched string at the matched string is returned by the function. If the function hits the end of file, it returns an empty string.

`readnum` *how*

The function skips the input file until the first occurrence of a number and returns the number. The file is positioned at the first character following the number. Argument *how* tells the function what kind of a number we are looking for. It can be `int` (the default), `hex`, or `flt`. The syntax of hexadecimal and floating point numbers is that accepted by Tcl expressions. If the function hits the end of file, it returns an empty string.

`readstring` *pattern1 pattern2*

This function first calls `skipinput` with *pattern1* and then extracts from the located string the portion specified by *pattern2* (which is another regular expression). If the function hits the end of file, or if the string located by `skipinput` doesn't match *pattern2*, an empty string is returned.

6.3 Logging and debugging

Yet another standard file available to a SICLE program is the log file. The standard name of this file is `siclelog`. It contains information written by the following operation:

`log` *string*

The specified string is written to the log file and followed by the end of line character. In front of the string, SICLE prepends the current time.

The log file is opened by the first call to `log`. If the program doesn't use the standard log file, it will not be created.

The default parameters of the log file can be changed by calling the following function:

`setlog` *fname lines versions*

The first argument specifies the name of the log file. If this name is different than the previous name, and the log file has been used (written to) already, the current file is closed and a new file is openened. If *fname* is an empty string, logging is disabled, i.e., the `log` operation will be void until a new non-empty name is defined by another `setlog` operation.

The remaining two arguments are optional and default to empty strings. If they are nonempty, they must be both nonnegative integer numbers. An empty value retains the previous setting of the argument.

If the log file is to be automatically rotated, *lines* specifies the number of lines that must be written to the file before its new version is started. The default setting of this parameter is 0, which stands for “no limit,” i.e., by default log rotation is disabled.

The last argument gives the number of versions of rotated files. The minimum value is zero (this is also the default), which indicates two versions: the current version and one old (previous) version. Old version are named as the current version (*fname*) with suffixes `.1`, `.2`, and so on.

Note that when the server is terminated and then started again, it will continue writing to the last (current) log file with the line count initialized to zero. This way, the file may significantly exceed the line limit before it is rotated.

The debug version of the package (`sicled`—section 1.4) verifies the arguments and environment of all SICLE functions and methods as they are being called. It also keeps track of 40 most recent calls. If the program gets aborted, file `debug` (created in the dirctory in which the program was called) will contain the trace of last 40 function calls with their object handles and arguments. Note that only SICLE functions are traced this way.

6.4 Miscellaneous functions

In several sections of this manual we mentioned the function `defined` whose purpose is to test various conditions, primarily dealing with SICLE objects and types. Below we describe the full syntax of this function.

`defined` *how where what*

The second argument can be one of the following:

`class`
: The function returns 1 if *where* is a valid class type, and 0 otherwise. The third argument is ignored.

`valid`
: The function returns 1 if *where* is a valid object handle pointing to an existing object, and 0 otherwise. The third argument is ignored.

`method`
: The function returns 1 if *what* is a valid method of the class type identified by *where*, and 0 otherwise.

`qualify`
: The function returns 1 if *where* is a class type that has been qualified (section 2.2) as *what*, and 0 otherwise. If *what* is not specified (or equal to an empty string), this option behaves exactly as `class`.

`is`
: The function returns 1 if *where* is a valid object handle pointing to an existing object whose type has been qualified (section 2.2) as *what*. If *what* is not specified (or equal to an empty string), this option behaves exactly as `valid`.

`constructor`
: The function returns 1 if *where* is a valid object type with at least one defined constructor, and 0 otherwise. The third argument is ignored.

`destructor`
: The function returns 1 if *where* is a valid object type with at least one defined destructor, and 0 otherwise. The third argument is ignored.

`attribute`
: The function returns 1 if *where* is a valid object handle and *what* is an attribute associated with that object (it can be an array name).

`belongs`
: The function returns 1 if *where* is a valid object handle and *what* is the class type of that object.

`callable`
: The function returns 1 if *where* is a valid object handle and *what* is a defined method for the object's type.

`sensor`
: The function returns 1 if *where* is a valid sensor name defined in the map file (section 5.1).

The following function returns the time of day in a compressed, purely numeric form:

`telltime` *how*

If *how* is `full` (the default), the result consists of exactly six digits `hhmmss`. If *how* is `minute`, the `ss` part is absent, i.e., we do not care about seconds. Finally, if *how* is `hour`, the results has only two digits representing the hour in the 24-hour notation.

SICLE offers the following three functions for generating random numbers:

`rndinit` *seed*

The function initializes the random number generator based on the specified integer seed.

`random`

The function returns a floating point pseudo-random number between 0 and 1.

`irandom` *range*

The argument is a positive integer. The function returns an integer pseudo-random number between 0 and *range*-1 inclusively.