User's Manual for the Sisyphus Simulation Environment[1]

Joel Grodstein
UUCS-86-012
October 12, 1986

Department of Computer Science
University of Utah
Salt Lake City, Utah 84112

[1]This work was supported in part by DARPA contract #DAAK-11-84-K-0017.

1. Introduction and disclaimer

This report describes how to create and simulate a design with Sisyphus. Inasmuch as Sisyphus is written in Symbolics-Lisp, some familiarity with both Lisp and with Symbolics computers is presumed. In addition, the concepts presented here presume an acquaintance with [3]. First, a disclaimer — this report is neither a user's manual nor a reference manual, but rather a combination of the two. As such, it is neither as detailed as a reference manual, nor as conversational as a tutorial. Further, this report describes yet another hardware description language and, in this author's humble opinion, there are already far too many HDLs and hence very little justification for writing more. To quote from the eminent numerician C.W. Gear's Forsythe award lecture [1],

> "Too many people are fond of designing new languages, languages which have wonderful structures for handling the 'n + 1/2 loop' problem. What we have is not the 'n + 1/2 loop' problem, but the 'n + 1/2 language' problem. We already have n languages, and there is always another half-assed proposal being made."

If Sisyphus were being rewritten, this report should merely describe a superset of EDIF [2]. However, at the inception of the current work, EDIF had not crystallized in sufficient form to be considered a standard (and, perhaps, still has not). In any case, the fact that EDIF does not currently support behavioral description would force Sisyphus to use a very extensive superset.

This document is arranged in several sections, which correspond roughly to the various phases of design and simulation. They are:

- Creation of the Sisyphus system.
- Description of the circuit under test to Sisyphus by the user. This is done by creating a set of Lisp flavors whose methods, when executed by Sisyphus, create a circuit data structure.
- Running a simulation via commands. These commands are simply a set of Lisp functions which call upon the Sisyphus simulation kernel.
- Graphical interaction with Sisyphus. This interaction enables the user to call, via menus, the functions which control a simulation. It also draws a picture of the circuit under test: all simulation commands which refer to individual circuit elements are executed by pointing at this picture.
- Using the standard library of components and flavors provided by Sisyphus.

Finally, two advanced topics will be covered:

- Using Sisyphus in nonstandard ways. This describes how to call the Sisyphus kernel without going through the standard graphics interface.
- Creating and defining new methodologies. This section is mainly for those researchers who are using Sisyphus as a testbed for new simulation strategies.

2. Creating Sisyphus

Sisyphus is a Symbolics Lisp system, and can be created by

```
(make-system 'Sisyphus)
```

All of the files for Sisyphus are stored in the same directory (see a system administrator to find out where). The system definition is in the file "system.lisp." The Sisyphus kernel is in the package **SIsyphus**, which is nicknamed **SIs**. A useful library of code is found in the package **transistors** (see Section 8). The package specifications for both of these packages can be found in the file "packages.lisp."

3. Specifying the circuit under test

Sisyphus components are instances of Symbolics Lisp flavors. Each component type (e.g., 7400 nand gates) has its own flavor, henceforth called a **component flavor**. Any component flavor must include one of two mixins defined by Sisyphus. These mixins provide default methods and let the Sisyphus kernel know about instances of the component. They are:

1. **hardware-component** is a standard component. It can have a behavioral view and/or a structural view. Viewed structurally, it consists of a collection of interconnected subcomponents. Viewed behaviorally, it is atomic: that is, the subcomponents are invisible.
2. **translator-component** is for components which have no physical existence (see [3, chapter 3]). Translator components are not hierarchical: they do not have subcomponents. Thus, components built on translator-component cannot be simulated structurally.

A component flavor will have many predefined methods by virtue of the mixins hardware-component or translator-component. However, the code for structural and behavioral description of a component flavor must be provided by the user. This is done via several additional methods, described in the following sections. For the most part, they do fairly obvious things.

3.1. The :instantiate-ports method

Code to instantiate ports of a component belongs in the **:instantiate-ports** method. It simply instantiates any of three port flavors: **in-port, out-port**, or **io-port**. The function specification is as follows:

make-instance '*port-flavor* &key *name type drawing-side level* *Function*

Port-flavor is the type of port to be instantiated. It must be one of **in-port, out-port**, or **io-port.**

Name is the name of the port. It should be a string, though it can be any hashable and

printable object.

Type is the type of values accepted by this port. For example, some ports transmit integers: other ports transmit voltage values. *Type* is, simply, any object of the desired type. For more on typing, see [3, chapter 2].

Drawing-side has meaning only in connection with the graphics window. It can take the value of **:top**, **:bottom**, **:left**, or **:right**. Most components are drawn as rectangular boxes; *drawing-side* allows the user to specify on which side of the rectangle the icon for this port is drawn.

Level has meaning only for ports of translator components. It can take the values of **:low** or **:high**, depending on the port's function.

Once a port is created, there are two ways to refer to it:

1. As always, make-instance returns the port instance created, which can be stored as an instance variable of the component that created the port.
2. Another way is by name (i.e., the string given with the **:name** keyword). Sisyphus keeps track, in a hash table, of all ports created by a component; any use of a string in a context where a port is expected causes a hash table lookup to be done. In a typical design, there may be many ports with the same name. Thus, the port name must be relative to some component. When executing within one of the standard Sisyphus methods (e.g., **:behave**), this context is set up correctly. When one of the above functions is called out of context, however, port names are relative to the top level component. This is the value of the global variable **sisyphus:*top-level***. Ports of components which are hierarchical sons of the top level component can be specified via dot notation (e.g., "adder.bit-0.x-in"). Alternatively, the value of ***top-level*** can be temporarily rebound to another component.

As stated, Sisyphus provides three flavors of ports: in-port, out-port, and io-port. As a general rule, it is a good idea that a port be an instance of the correct flavor(!). In particular, the following actions are related to port flavor:

1. Trying to post a value to an input port will cause an error.
2. Input ports do not drive the wire that they are on.
3. When a wire's value changes, the event handler does not awaken output ports on that wire, but only input and io ports.

Io ports have the union of the properties of input and output ports. If a port which needs to have information flowing through it in a bidirectional fashion is not declared to be an io-port, then it will be missing some of the essential properties it needs (as per the above enumeration).

If a port is declared to be an io-port even though it is really only an output port, inefficiency will result. In particular, event handling causes components to be awoke needlessly and the simulation algorithm thrashes.

If a port is declared to be an io-port even though it is really only an input port, then it will drive a wire that it really should not. This may cause conflict on the wire and hence result in its value becoming unknown.

Sisyphus supports the notion of typed ports. That is, a port may be specified to accept only values of a given type, specified by its instance variable *type*. This is initable and settable: it may be modified at any time. The value of *type* should be any arbitrary object of the type desired for the port. In order for a Lisp machine type to be recognized as a type by Sisyphus, it must have certain methods. These can be supplied by the mixin **basic-type**, supplied by Sisyphus. For ease in defining new data types, the following macro is provided:

define-type *type-name (instance-vars. . .)* &key *base-types skip options* *Macro*

This defines a new flavor called *type-name*, with the instance variables *instance-vars*, built on the flavor 'basic-type. It also defines a constant named ***type-name*-type***, which is an instance of the flavor *type-name* suitable for giving to a port.

Skip lists methods which you wish to provide yourself instead of using the defaults. This might be used, for example, to provide special conflict resolution in the case of multiple drivers on a bus.

Base-types allows you to build this flavor on something other than 'basic-type. This is an advanced feature.

Options are any keyword options to give to the defflavor function.

For examples of these advanced features, see the code in "types.lisp". It provides the following standard type definitions:

```
(define-type boolean-transistor-signal (boolean-value))
(define-type thevenin-transistor-signal (V-thev R-thev))
(define-type norton-transistor-signal (I-norton G-norton))
```

All of these types are compatible with each other. Thus, boolean and thevenin can be interconnected on the same wire. Each of these types provides the following services:

- A constant value suitable for giving to a port as the **type** instance variable. The name of the constant is the type name followed by "type". E.g., the constant for boolean transistor signals is called ***boolean-transistor-signal-type***.
- A method **:boolean-value**, which translates the object's value into a 0, 1, or :unknown.
- Methods **:I-norton** and **G-norton**, which return the current and conductance of the value's Norton equivalent circuit.
- Methods **:V-thev** and **R-thev**, which return the voltage and resistance of the value's Thevenin equivalent circuit.

Resolution of any combination of these values occurs by first converting them to Norton

equivalent circuits. Then the currents and conductances are added, as circuit theory dictates for parallel combination. Finally, a new Norton object is created with the net value.

3.2. The :behave method

A component's **:behave** method defines that component's behavioral view. Code included here would typically read the component's input ports, calculate results based on the component's functionality, and post these results to the component's output ports. Code in this method can make use of several Lisp functions provided by Sisyphus:

value-of *port-spec* *Function*

Port-spec specifies a port of the component as per section 3.1. This function returns the current value on the wire connected to *port-spec*. It is typically used to read input ports.

post *port-spec value* &optional *(delay 0)* *Function*

This is used to send values to output ports. *Port-spec* is the port to send output to.

Value is the value to send to the port. It must be a typed object of the correct type.

The posting will take effect after *delay* units of time. Time is measured in integral multiples of the variable ***time-multiplier*** (originally set to .01 nS). Delay may be specified explicitly, or as a unit delay (as per the Conlan timing model). In the latter case, the keyword **:unit-step** is used as the value of *delay*.

3.3. The :instantiate-sons method

Together with the :interconnect method (quo vadis), the **:instantiate-sons** method defines the structural view of a component. Code in this method instantiates subcomponents (note that translator components cannot have an :instantiate-sons method). The instantiation is done by the standard Lisp function **make-instance**. In addition to any keywords required by the particular component flavor being instantiated, there are several keywords accepted by the mixin flavors hardware-component and translator-component.

make-instance *component-flavor-name* &key *name graphics-position mirror-x mirror-y mirror-xy want-structural* *Function*

Component-flavor-name is the name of the component flavor to be instantiated.

Name gives the subcomponent a name for later reference.

Graphics-position specifies the location of the subcomponent in the graphics window. It should be a list of the form (x-location, y-location). The coordinate system ranges from 0 to 1, with the origin in the upper left hand corner.

Want-structural, if true, specifies that the component should be simulated structurally. If not supplied, then it defaults to structural simulation if possible.

The remaining keywords refer to the subcomponent's visual representation in the graphics window. They specify its orientation (in order to make the schematic less cluttered).

3.4. The :interconnect method

The structural view of a component is completed by the **:interconnect** method, which interconnects ports via wires. This code can make use of several functions provided by Sisyphus:

connect &rest *port-spec ...* *Function*

This connects ports with a wire. When connecting to a port of a subcomponent, it can be specified using the string "subcomponent-name.port-name."

set-value *port-spec value* *Function*

This allows you to preset input ports with values (for instance, connections to V_{dd} or ground might be set up in this way).

3.5. The :capacitances method

The **:capacitances** method helps Sisyphus determine the capacitances associated with a component's ports. It is called with one argument, a port. The component should return a list of any capacitances associated with the port. If this method is not defined by the user, a default method is provided by Sisyphus. This default method estimates capacitances as per [3, chapter 3]. Sisyphus defines two useful flavors of capacitances:

stray-capacitor

A stray floating capacitance. It has two initable instance variables, **port-1** and **port-2**. These should be set to the two ports of which it is a stray capacitance.

stray-grounded-capacitor

Stray grounded capacitance: just like **capacitor**, but has only one port. The other port is assumed to be connected to an AC ground. This would typically be used for capacitances to bulk.

3.6. The :graphics-lines method

The **:graphics-lines** method is called when a component is drawn on the graphics window. It should return a list of points which are interconnected to form the component's outline.

3.7. The :parameterize method

It often happens (e.g., with PLAs) that different instances of a component are similar, but not identical. This is handled by using parameterized component flavors. The flavor has instance variables which, when given different values, allow different instances of the flavor to be slightly different. The specification of values for these variables is done in the **:parameterize** method. If a :parameterize method is defined, Sisyphus calls it automatically whenever a component is instantiated. For ease in programming, the following macro is provided:

define-parameterize-method *component-flavor-name* &rest *instance-vars...* *Macro*

This automatically generates a :parameterize method for the flavor *component-flavor-name*. This method fixes values for the given *instance-vars*. If, for any particular instantiation of the component flavor, the variables are given values by init keywords to **make-instance**, then the :parameterize method does nothing.

If, however, there are any variables left unbound, then the user will be prompted for their values.

4. Control of a simulation through the functional interface

Sisyphus provides a set of functions to prepare and execute a simulation. They are:

set-value *port-spec value* *Function*

This function forces the value of the wire containing *port-spec* to *value*. Events are created as appropriate to process the change.

:set-border *want-structural* *Method of hardware-component*

When a component receives this message, it is changed to structural/behavioral simulation if *want-structural* is **t** or **nil** respectively. This message is not handled by translator components, since they can only be simulated behaviorally.

change-methodology *component methodology* *Function*

Responsibility for simulating *component* is given to an instance of *methodology*. *Methodology* should be a symbol, not an instance.

big-bang *Function*

When called, it resets global simulation time to zero and erases any stored state.

spin &optional *until-time* *Function*

Starts a bout of simulation. Simulation continues until *until-time*, which should be a time-stamp object (see section 11.2). If *until-time* is not given, then simulation continues indefinitely.

DC-settle *Function*

Performs DC operating point analysis. This works by binding the global variable ***DC-settling*** to **t** and then invoking **spin**. For methodologies which do not distinguish between DC and transient analysis, **DC-settle** is the same as **spin**.

5. The Sisyphus graphics window

For convenience of visualization, Sisyphus provides a graphical tool for viewing and simulating a circuit. It operates via menus and submenus. As is typical in Lisp machine applications, the top level menu itself is obtained by clicking once on the rightmost mouse button. It contains menu tags for the following choices:

- Top level file and component control
- Entering Zmacs to edit the methods for a component.
- Setting various variable values to customize the behavior of Sisyphus.
- Running a simulation and viewing the results.

Each of the above functions corresponds to one menu tag on the top level menu. They are discussed in more detail immediately below.

5.1. Editing a component

The menu tag "Session control," when picked, pops up a submenu with two options. One option, "Edit new component," allows a user to edit a new component. Sisyphus prompts for the name of a component (i.e., a flavor) to edit, and creates an instance of it. The resulting circuit is displayed in the Sisyphus graphics window.

The other submenu option, "Hardcopy," creates a PostScript file with a picture of whatever is currently displayed in the graphics window.

5.2. Editing methods

The second tag in the main menu is "Editing methods." Each of the submenu tags in this menu serve to enter Zmacs and edit a particular method. The possibilities are "Edit :behave method," "Edit :instantiate-ports method," "Edit :interconnect method," and "Edit :instantiate-sons method." The methods are relative to the component flavor of the component currently being viewed.

5.3. Setting variable values

This menu tag, when selected, allows the user to custom-tailor the behavior of Sisyphus. Its options are many: the user is invited to experiment.

5.4. Running a simulation and viewing the results

The main menu tag "Simulation control" allows the user to invoke the Sisyphus simulator kernel. This menu tag pops up a submenu with three options:

Big bang
: Resets the global simulation time to zero and clears all pending events. This is done through the previously described **big-bang** function.

Simulate
: Starts the timing wheel spinning: that is, starts a simulation. Sisyphus prompts for how long (in simulated time) the simulation should last.

DC Settle
: This has meaning only for analog simulation methodologies. In this case, it computes the DC solution to circuit network equations. For other methodologies, it is the same as the "Simulate" submenu tag.

5.5. Mouse Sensitive Items in the Sisyphus Graphics Window

Almost all of the items in a Sisyphus graphics window are mouse sensitive, allowing easy access to many of the most common Sisyphus commands. Mouse sensitive ports, when selected, allow several options:

Set \ The special variable \ is set to the selected port. This is very much like the similar facility available in the Symbolics Inspector window. It is useful when you wish to refer to the port later.

Set port value
: The user is prompted for a value, and the port is set to that value. This works via the aforementioned **set-value** function.

Watch
: This ports is watched by graphical plotting software. The graphical plot can be viewed by typing <SELECT>-G.

Mouse sensitive components, when selected, provide several options:

Set \ Sets the special variable \ to the selected component.

Push and Edit
: changes the frame viewed in the graphics window. The viewing level is pushed hierarchically deeper.

Desire

A component may desire to be simulated either behaviorally or structurally. This menu tag allows the user to set whether a component desires behavioral or structural simulation. It works by sending the component a **:set-want-structural** message.

Change methodology

This changes the methodology which is responsible for simulating this component. A submenu lists all methodologies available.

6. Driving a simulation

There are no special facilities in Sisyphus for driving a simulation. However, the standard Sisyphus facilities can be used for driving a simulation in several ways:

- By hand. That is, the user can use the "Set Port Value" command to set nodes individually to the values desired. Then, use the simulation control menu to start the simulation.
- By commands to the Sisyphus simulator kernel. These commands are only the standard commands that could be entered by hand. However, instead of giving the commands one at a time to the graphics window, they are given as function calls by Lisp code.
- By driver components. One or more components in a design can be set up to drive the simulation. These are standard Sisyphus components: their distinction as simulation drivers is for the user's benefit only, and is transparent to Sisyphus.

  The :behave method of these components can use arbitrary Lisp code to post values to the inputs of other components and hence drive the simulation.

7. Viewing the results of a simulation

Sisyphus does not provide elaborate facilities for viewing the results of a simulation. However, it does provide hooks so that such facilities can be easily written. This is compatible with the notion of Sisyphus as an *extensible* design environment.

Whenever the value of a wire changes, the change is implemented via the :set-value method of that wire. Normally, this method merely updates the values drawn in the graphics window. However, the user is free to define :before or :after daemons to do more complex things. For example, one might want to limit the messages to user-defined "watched" nodes.

8. Transistors package

Sisyphus supplies a library of useful components in the package **transistors**, which is loaded automatically with Sisyphus. It is not "special" code, but merely a set of useful routines defined just as any user could define them. The source code is in the files "transistors.lisp" and "transistors2.lisp." First, Sisyphus provides several flavors of transistors and gates for the user:

- nenhancement-transistor
- penhancement-transistor

- depletion-transistor
- one-way-nenhancement-transistor
- pull-down
- pull-down2
- precharge
- pull-up
- inverter
- xor
- nor-2
- nand-2
- boolean-to-thev

The functions of these components should be discernible from their names. To find the names of their ports, look at the **:instantiate-ports** methods in the code itself. For a more detailed description of the components, see the code. The library may be expanded in the future: again, see the code. In addition, two flavors of capacitors are provided. These are *not* the stray capacitances that are mentioned in section 3.5, but rather explicitly designed capacitors. They are:

explicit-capacitor A non-stray capacitance. This might be used for bootstrap capacitances. It is an actual component based on **hardware-component**. Its structural decomposition is a depletion transistor with **port-1** being the gate, and **port-2** being the source and drain tied together.

routing-capacitor Similar to **explicit-capacitor**, but with only one port **port-1**. The other is tied to ground. This is typically used for routing capacitances.

Transistors have no **:behave** methods, per se. Thus, they cannot be simulated in the default methodology. However, several other methodologies (e.g., Mossim) change transistors to a flavor which does have a **:behave** method. The higher-level gates tend to be unidirectional, directly simulatable and hence do have **:behave** methods.

It is a good idea to make use of these components instead of defining equivalents, since:

- All of the standard Sisyphus methods are already defined, saving a user the trouble.
- For transistors, most of the existing methodologies recognize transistor types by the flavor names given above. Thus, if a user defines his own type of transistors, existing methodologies will not recognize them as such.
- Some methodologies might treat compound components specially. For example, quasi-analog simulators can take advantage of a NOR gate's structure to simulate it efficiently with macromodeling.

9. Nonstandard uses of Sisyphus

9.1. Avoiding the graphics window

One might wish to use Sisyphus for simulation alone, and disregard its capabilities for schematic capture and graphical result-viewing. In this case, the Sisyphus graphics window would most likely be nonexistent. This presents no problem at all: in fact, the graphics window is completely separate from the Sisyphus kernel. None of the Sisyphus kernel references the graphics window.

Sisyphus without a graphics window can be created by

```
(make-system 'Sisyphus-no-graphics)
```

Of course, having chosen not to instantiate a graphics window, one cannot use the graphical interface to Sisyphus. Normally, components are instantiated via the graphics window. For those who wish to go it themselves, the following information is necessary. Most Sisyphus component methods are called automatically. Instantiating a component automatically generates the correct calls to instantiate its sons and ports, interconnect them, instantiate grandchildren, and so on. Thus, **make-instance** need be called by the user only infrequently.

Since Sisyphus can handle only one top level component at a time, setting a new top level component erases the old one. In order for Sisyphus to perform the necessary bookkeeping, a new top level component must be signaled with the following function:

edit-new-component &optional *flavor* (*methodology* 'normal-wheel) &rest *init-args* *Function*

This function creates an instance of *flavor* and makes it the new Sisyphus top level component. *Init-args* are merely passed along to **make-instance**. If *flavor* is not supplied, then Sisyphus prompts for *flavor*. The most notable side-effect of this is that the global variable ***top-level*** is set to the instance of the new top level component.

Methodology is the methodology used to simulate the new component: i.e., it is encapsulated in a supercomponent of flavor *methodology* [3].

Finally, one last thing must be done when bypassing the graphics window. In order to make unknowns work correctly [3, chapter 2], Sisyphus needs a special condition handler bound. Normally, this is done by the top level loop in the graphics window. Otherwise, it can be bound with

```
(condition-bind ((condition 'unknown-handler))
   (all of your code goes here))
```

9.2. Modifying circuit structure

Sisyphus is a conversational system. When the structure of a component is modified, it need not be recreated from scratch. There are two ways to change circuit structure: via methods and via low level function calls.

9.2.1. Sisyphus methods

Circuit structure is defined in the standard Sisyphus methods. For example, the wires for a component are defined by the **:interconnect** method. The execution of these methods produces a data structure. The easiest way to change circuit structure is by changing the appropriate methods and then calling them. These methods can be called any time — bookkeeping is automatic (see section 9.1).

Currently, there is one difficulty: physical wires are not regenerated after methods are called. To remake wires, the following function should be used:

redo-physical-wires *component* *Function*

Remakes the physical wires for *component* and all of its hierarchical descendents. It is very efficient.

9.2.2. Low-level function calls

Calling a method as just described is sledgehammer-like overkill. To change one wire, it requires deletion and rebuilding of whole blocks of component structure, which may be slow. There is a way to update circuit data structure piece by piece: however,

- it is not easy to use,
- the code for the methods must still be updated to reflect the incremental changes, as above, and
- it is not efficient. That it, making changes *en masse* allows more efficient algorithms to be used than does the piecemeal approach. Of course, an inefficient algorithm applied to a small amount of data may still be fast.

The following functions are useful for small structural changes:

connect *port-1 port-2* *Function*

This is the standard **connect** function described in section 3.4. It may be called outside of the context of a :interconnect method; however, in this case, the global variable ***current-interconnector*** must be bound to the component whose :interconnect method would have called connect.

make-physical-wire *ports* *Function*

When **connect** is called out of context, this must be called afterwards to recreate physical wires. It remakes the physical wire for the list of *ports* and any other ports connected to *ports* through local wires.

:split *ports* *Method of local-wire*

Disconnects the given *ports* from the local wire that they are currently on, creates a new local wire and places them on this new wire. Then, calls **make-physical-wire** as appropriate.

make-instance *component-flavor* &key *name* *Function*

The standard instantiation function. When a subcomponent is being created (as opposed to a new top-level component), make-instance can be called directly instead of through **edit-new-top-component**. When doing this, the global variable ***current-instantiator*** must be bound to the desired parent of the new component. All required methods are called automatically for creation of local wires, subcomponents, and ports. Physical wires must still be created by **redo-physical-wires**.

:delete *Method of basic-component*

When this message is sent to a component, it deletes itself, its ports, its local wires, and likewise for all of its hierarchical descendents. The physical wire structure is updated automatically.

10. Methodologies

A methodology is responsible for responding to messages from the Sisyphus kernel. These messages allow the methodology to use a tailor-made user interface. The messages are described below.

:init Called by the Lisp flavor system when a supercomponent (or, for that matter, any Lisp object) is instantiated. Sisyphus has no requirements on what, if anything, the designer does in this method.

:big-bang

When this message is received, all stored state should be destroyed, any pending events canceled, and any concept of simulation time reset to zero.

:add-son *son*

This message is sent for all of the son components and subcomponents in the supercomponent. This is done in one fell swoop, before anything else. A methodology may take advantage of this to prepare the component for simulation (e.g., by changing its flavor or by using its spare slot). Sisyphus provides a spare instance variable called **methodology-spare** in each component, for use by its supercomponent.

:delete-son *son*

The opposite of :add-son. This could be called when a component is deleted by the user. Equally, it could be called when some parcel of the current supercomponent is being given to another methodology.

:add-port *port*

These messages are send in one fell swoop, just after the :add-son messages are sent. A methodology may take advantage of this message to prepare the port for simulation (e.g., by using its spare slot). Sisyphus provides an instance variable called **methodology-spare** in each port, for use by its supercomponent.

:add-wire *physical-wire*

This message is sent for each *physical-wire* in the supercomponent, before any ports are added to the physical-wire. Physical wires have a hash table associated with them, and hence respond to hash table messages. This may be useful for associating data with the physical-wire: for instance, a given simulation methodology will typically keep information regarding a node above and beyond the simple connectivity information kept in physical wires. Another, more efficient way to keep track of information associated with a node is to keep a pointer to that information in the spare slot of ports on a wire. Incidentally, the reason that physical wires do not have a spare slot is that a single physical wire may span multiple supercomponents.

:delete-wire *physical-wire*

The opposite of :add-wire.

:add-awakable-port *port physical-wire*

Add an input or io port to a *physical-wire*. The *physical-wire* will have been already added via :add-wire. Calls to :add-wire and :add-awakable-port may be intermixed, but a wire will always be added before any of its ports.

:add-driver-port *port physical-wire*

Similar to :add-awakable-port, but for an output or io port.

:delete-awakable-port *port physical-wire*

The opposite of :add-awakable-port.

:delete-driver-port *port physical-wire*

The opposite of :add-driver-port.

:add-grounded-capacitor *capacitor physical-wire*

Adds the given *capacitor* to the *physical-wire*. A grounded capacitor has only one port, its instance variable **port-1**. This message is only sent for parasitic capacitors, i.e., those returned by a :capacitances method. Explicit capacitors, i.e., those generated in an :instantiate-sons method, are added just as any other components are, by the :add-son and :add-port family of messages. If a methodology does not handle the :add-grounded-capacitors message, then Sisyphus does not send any capacitance information to supercomponents of that methodology, significantly speeding up supercomponent creation time.

:add-floating-capacitor *capacitor physical-wire-1 physical-wire-2*

Adds the given parasitic floating *capacitor* between *physical-wire-1* and *physical-wire-2*. The capacitor has two ports, **port-1** and **port-2**. Sisyphus guarantees that all of a capacitor's ports are added, by :add-awakable-port and :add-driver-port before, the capacitor is added.

:delete-capacitors *physical-wire port*

The opposite of the preceding message. Delete, from the given *physical-wire*, all capacitors whose **port-1** or **port-2** are the given *port*.

:change-behavioral-status *component behavioral? input-wires output-wires*

The given *component* has been changed (typically at the user's request) from behavioral to structural simulation or vice versa. *Input-wires* are the physical wires

connected to *component*'s input or io ports. *Output-wires* are the physical wires connected to *component*'s output or io ports. All movement of ports on and off of physical wires will be accomplished via the :add-?-port family of messages — the :change-behavioral-status message will typically be used for minor bookkeeping only (e.g., event scheduling).

:draw-ports-value *port*

For graphics use only. If the supercomponent handles this message, then it is sent for each port when the port is drawn, and should return an object to be drawn as the port's value. If the methodology does not handle this message, then the port's value is provided by its :value-for-drawing method, which is provided by Sisyphus.

:force-value *physical-wire value* &optional *delay* &key *AC-ground*

Force a *physical-wire*'s value. This is called when the user sets a value — it is not called during simulation. Bus resolution is not invoked. If *delay* is given, then it is the rise or fall time for a piecewise linear ramp. If *AC-ground* is **t**, then the *physical-wire* should be considered an AC ground (i.e. either Vdd or ground).

:external-touch *physical-wire* &optional *old-value*

Called when a physical-wire is changed during a simulation. If the current value of the wire is significantly different from *old-value* (by whatever standards this methodology uses), then an event should probably be created with **awaken**. If *old-value* is not supplied, then the caller wishes the change to be considered significant regardless.

:behave Tells the supercomponent to simulate itself.

:figure-out-awakables *physical-wire inputs changed? port-posting*

If a methodology uses a timing wheel from Sisyphus' library (cf. Section 11), and the supercomponent sends the timing wheel a :behave message, the :figure-out-awakables message may be sent to the methodology in turn. When the timing wheel is sent a :behave message, it starts event processing. For each event, it calls the currently executing supercomponent for information as to who needs awakening. *Inputs* are all of the inputs on the physical-wire. *Changed* indicates whether processing this event has changed the wire's value. *Port-posting* is the port whose output created the events. This method will typically return some subset of *inputs.*

:trial-voltage *port*

Used primarily by relaxation methodologies. To avoid overhead, these algorithms will generally not awaken a wire until relaxation is complete. However, components must have access to the current relaxation value of a wire during their :behave methods. If this method is supplied, then the **value-of** function will call :trial-voltage instead of merely returning the wire's value. This method will typically return the provisional relaxation voltage, as opposed to the actual wire voltage.

:fast-voltages &rest *ports*

This method is provided for efficiency reasons. In an analog analysis, both the **:trial-voltage** and **value-of** functions are far too inefficient. The :fast-voltages message can be used with less overhead. *Ports* are the ports whose voltages are desired. This method should return a list of the desired voltages.

11. Default methodology

The default methodology used by Sisyphus is basically a straightforward event-driven algorithm. Since event-driven algorithms are ubiquitous in simulation, most of the services provided by this methodology may also be useful in writing new methodologies. The default methodology is based on a timing wheel and events.

11.1. Timing wheel

A timing wheel (also called an event queue) forms the heart of most event-driven simulation data structures. Sisyphus uses one (and provides it as a flavor called **timing-wheel**). Other methodologies (e.g., Mossim) use this flavor for simulating functional blocks, even though their basic algorithms are not event-oriented.

Event processing occurs when the timing wheel is sent a **:behave** message. Any instantiation of the default methodology forwards this message to its timing wheel. In response to this message, the timing wheel

1. Gathers up all of the events whose time stamp is the current time and sends them the **:update-port-values** message. For events which signal the assignment of a value to an output port, this tells the event to actually perform the assignment.
2. Sends them the **:resolve-and-figure-out-awakables** message. Here, each event must first, if applicable, set the new wire value based on bus resolution of all output ports on the wire. Then, it must decide which input ports on the wire need to have their components awoke. This list of input ports is returned to the timing wheel, which then sends **:behave** messages to the appropriate components.

The default methodology also forwards the **:external-touch** message to its timing wheel. In response to this message, the timing wheel creates and schedules an event to process the wire touched at the current time.

One other message that the timing wheel supports is called **:active?**. This message is a query predicate to the timing wheel about a port. Typically, the :behave method of a component sends it. The timing wheel responds in the affirmative if and only iff the given port is the port which caused its component to be awoke.

11.2. Events

Any timing wheel needs events. Sisyphus provides two flavors of events. They are both time-stamped, but are processed differently. One flavor is the standard **event**, used by methodologies in which components post values to output ports with a specified delay. When this flavor of event is processed it sets the specified output port to its new value, and notifies all

other components on the wire.

The other flavor, called a **touch-event**, has nothing to do with output ports. It is typically used by methodologies which do *not* see a wire's value as being determined by conflict resolution of all output ports on the wire, but rather as being determined by some other algorithm. Graph-based algorithms such as Mossim fall into this category. Also, wires which are clamped to a value (e.g., by the user forcing a node) create touch-events. When creating a touch-event, the user specifies only a wire to be 'touched,' a time delay, and the new value.

Both of these event flavors are time stamped. Sisyphus uses the Conlan notion of time, where global time is divided into macro-steps and micro-steps. In addition, each time-stamp includes a **nano-step**, which is a unique monotonically increasing integer. Nanosteps assure that time-stamps are completely ordered by the **<** function. In the event of two time-stamps having identical macrosteps and microsteps, the one which was created first (and hence has the lower nanotime) is deemed to be smaller.

Events respond to several messages:

:schedule:

This message is sent automatically by every event to itself at creation time. It causes the event to insert itself into whichever timing wheel is bound by the global variable ***timing-wheel***. The event's position in the timing wheel is determined by its time stamp.

:update-port-values:

Each of the two event flavors treat this message differently. A touch event is not concerned with values from ports: hence, it ignores the message entirely.

A normal event sets the new value of the port which created the event, as the event creator planned. If the new (old) value is **:high-Z**, then the port is removed from (inserted onto) its wire accordingly.

:resolve-and-figure-out-awakables:

Again, each event flavor treats this event differently. A normal event first gathers up all of the output ports on the wire, then invokes bus resolution to determine the new wire value. A touch event, on the other hand, simply sets the new wire value to whatever the event's creator specified.

In either case, the event notes whether the new wire value is different from the old value. It then contrives to have input ports on the wire awakened. First, it calls the **awaken** function [3, chapter 4], which takes care of ports belonging to all but the current methodology. Then, it finds all input ports on the wire which belong to the current methodology and asks this methodology which of them to awaken. This list of ports gets passed back to the timing wheel, which actually calls the **:behave** methods.

References

1. Gear, C.W. Numerical Software: Science or Alchemy. UIUCDCS-R-79-969, University of Illinois at Urbana-Champaign, June, 1979.

2. Crawford, J. "Edif: A Mechanism for the Exchange of Design Information". *IEEE Design and Test of Computers 2, No. 1* (February 1985).

3. Grodstein, J. Sisyphus — An Environment for Simulation. Master Th., University of Utah,1986.