Norman Hendrich

HADES Tutorial

version 0.43 — July 7, 1998

Contact:

University of Hamburg
Dept. Computer Science
Norman Hendrich
Vogt-Koelln-Str. 30
D-22527 Hamburg
Germany

hendrich@informatik.uni-hamburg.de

version 0.43
July 7, 1998

Contents

1 Introduction

This section introduces the concepts behind HADES, the "Hamburg Design System", a portable Java and Java-Beans based visual simulation environment. Subsection 1.2 explains the need and goals for HADES. A short overview of related work, covering existing simulation frameworks, digital simulation, and visualization, is presented in subsection 1.3. Subsection 1.4 contains a summary of the functionality of HADES, together with a list of the features planned for the near future. For a detailed discussion of HADES see [Hendrich 97].

visual simulation

1.1 How to read this Tutorial

For a quick start with HADES, read section 2, "Hades in a Nutshell". It presents the user interface and the most basic editor commands and simulation commands. Section 5 on page 53 lists some frequently asked questions.

Section 3 builds upon section 2 and has information for advanced users already familiar with digital simulation and the basic HADES functionality. It describes how to create hierarchical designs, lists the common shortcut *bindkeys*, and presents some advanced editor and waveform commands.

Section 4 explains how to write your own simulation models. Three examples are studied in detail. First, a simple AND gate is presented as a quasi "minimal" example. Next, a D flipflop shows the interaction between components and simulator. The third model, a clock generator, is used to explain access to the simulator, graphics, mouse input, I/O, and configuration.

1.2 Concept

Our main intention is to provide a simple and portable design and simulation environment that may be used by students without long training, and that allows them to "play" with digital circuits. To this end, HADES—like DIGLOG [Gillespie & Lazzaro 96]—allows an interactive simulation mode, where the inputs to a circuit are clickable. So you can set the input values while the simulation is running, without the need to write an external stimulus file. Also, HADES includes a higher degree of animation capabilities than found in most other electronic design systems.

simple and portable

On the other hand, HADES should also allow more advanced design styles to avoid boring experienced designers too soon. With HADES, you can design your systems at home on a standard PC running the Java virtual machine. At least for electronic designs, you can always export your circuits to a state-of-the-art design system when necessary. See section 3.7 for the available export options.

Unlike some other systems, where simulation models have to be written in a crippled internal language, all HADES simulation models are directly written in Java. This

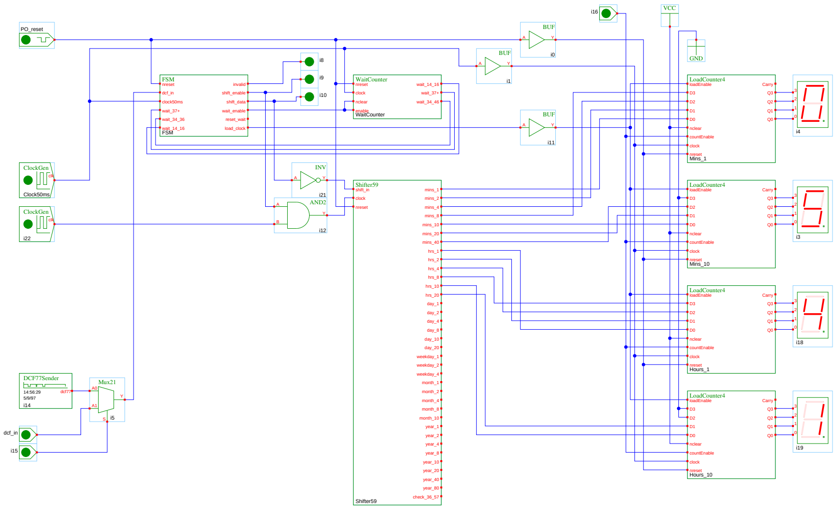


Figure 1: A HADES example design: The toplevel schematic of a DCF-77 radio controlled digital clock, showing basic gates, subdesigns, animated seven segment displays, and behavioral components including the DCF-77 sender.

gives the designer full access to a state-of-the-art object-oriented programming language with a rich class library, including full network access and portable graphics. Naturally, typical HADES simulation models written in Java tend to be a bit more verbose than models written in a special simulation language like VHDL [IEEE-93], which offers tailored syntax constructs as signal assignments (<=) or `wait` statements.

The unique combination of a powerful language, a visual editing environment with animation support, and full access to the simulation kernel allows for new design and debug possibilities. For example, writing complete and interactive system-level testbenches for digital system simulation is much easier than with other design systems. Also, due to the full object-oriented design of each of the key modules, including all simulation objects, signals, the simulation engine and the editor, it should be easy to use HADES for other application areas than just digital system design. While HADES cannot reach the simulation speed of special-purpose simulators, the additional flexibility of its object-oriented design should outweigh this at least for models with low event rate. At the moment, we plan to develop additional HADES simulation models for basic image processing operators. See section 4 about how to write your own HADES simulation models.

1.3 Related Work

ModSIM, KHOROS Some commercial general-purpose simulation environments are built upon special simulation languages. One example is the SIMGRAPHICS-II environment with its proprietary simulation language called MODSIM-III [CACI 97]. Most tools, however, use standard programming languages like Modula, C/C++ or still FORTRAN for their simulation models, and provide function or class libraries optimized for specific application areas. Perhaps the best known example is the KHOROS [Khoral 97] environment for the simulation and visualization of image processing tasks.

professional EDA tools

In electronic design automation, interactive schematics and layout editors have been around for years. However, most commercial design systems like Cadence's Design Framework [Cadence 97] or Synopsys VSS [Synopsys 97] are targeted towards professional design engineers. Therefore, they concentrate on functionality and performance, while ease of use is of little concern. Usually, support for interactive (instead of batch mode) simulation is limited in these frameworks.

DigLOG

On the other hand, several public-domain and shareware tools intended for beginners are available for the design and simulation of electronic circuits. A well known example are the DigLOG and AnaLOG simulators [Gillespie & Lazzaro 96] from the University of Berkeley. Both simulators use a simple visual editor and provide libraries for all standard components and most of the 74xx series. The simulation models are accurate and highly optimized, resulting in high simulation performance. Behavioural or complex models, however, are difficult to write, because DiGLOG models use a very simple internal language, and support for hierarchical designs is limited. Also, the graphical capabilities are limited, because DigLOG uses a proprietary user interface without access to the standard GUI components.

Digsim [DIGSIM] is a digital simulator written in `java` which supports interactive simulation of simple gate-level circuits. However, support for more complex or hierarchical designs is limited.

1.4 What is HADES?

The following list of "buzzwords" may help to clarify what HADES is, and what not. With the current version of HADES, you can:

- design and simulate digital electronic circuits,
- on gate level, RT level, and system level,
- visually compose and configure your designs,
- include behavioral and system level components,
- co-simulate hardware and software systems, where software may be either native Java code or software running on embedded processors,
- write your own models, and include them seamlessly with other components,
- export your designs to RTL and gate-level VHDL (Synopsys compatible),

- annotate your designs with all graphical objects available from the xfig or JavaFIG graphics editors [xfig 3.2]. This includes the option to export your schematics in high quality to color Postscript format.

available libraries Naturally, the application spectrum of any design system depends largely on the collection of available component libraries. In the current version 0.43of HADES, the following models are available:

- all basic gates up to four inputs,
- complex-gates with up to six inputs,
- all standard flipflops,
- a variety of I/O models, including animated components like LEDs or seven-segment displays,
- behavioral models for some 74xx TTL circuits,
- some RTL and high-level models,
- interfaces to our KV-diagram editor and FSM editor applets.

At the moment, all models use a simplified generic timing as might be typical for a 1.0μm CMOS library. If necessary, subclasses with detailed timing data could be generated to model a specific ASIC library accurately. See section 4 on how to write your own HADES simulation models.

The following extensions are planned for the near future:

- application specific HADES model libraries, e.g. image-processing operators, neural computing elements,
- a library of industry standard processor and ASIC models,
- limited support for mixed analog/digital design simulation,
- support for switch-level simulation.

2 HADES in a Nutshell

This section introduces the first steps necessary to run HADES. It explains the installation and the most basic editing and simulation commands. As a first example, a D-latch flipflop will be built and simulated in this section.

2.1 Installation

system requirements

HADES is written as a pure Java 1.1 program [Sun 97]. Therefore, HADES should run without modification on every computer system that offers a Java 1.1 compatible Java virtual machine. At the time of writing (Q3 1997) this includes WINDOWS'NT and WINDOWS'95, LINUX, most commercial variants of Unix, and several others, with a Macintosh version announced. While HADES should run on older hardware like a 486 or microSPARC, you probably won't like the performance. A minimum of 16 MBytes of memory is recommended, but 32 MBytes or more are certainly better. We have tested and used HADES as a standalone application on Pentium based PCs under WINDOWS'NT and WINDOWS'95, and on UltraSPARC workstations running SOLARIS.

running HADES as an applet

You may run HADES as an applet from within any Java 1.1 compatible WWW-browser, e.g. `hotjava` from Sun Microsystems or MSIE version 4.0 from Microsoft. Note that Netscape NAVIGATOR 4.0 does not support Java 1.1 (yet). However, the HADES applet is only useful if you have a high-speed network connection to the applet server, because it needs to download about 2 MByte of class-code each time you start it. Ask your system administrator to install the HADES classes, documentation, and examples to your intranet WWW-server. Otherwise, install the HADES software to your local disk.

Java 1.1 installation

The installation of HADES to your machine requires two steps. First, make sure that a Java 1.1 compatible virtual machine is available on your system. While current versions of some operating systems, e.g. LINUX and OS/2 already include support for Java 1.1, on other operating systems you will need to install the Java virtual machine as a separate program. A free version of the Java 1.1 runtime (JRE) for SOLARIS or WINDOWS'NT and WINDOWS'95 is available for download on the WWW-server of Sun Microsystems, `http://www.javasoft.com/nav/download/`. For other operating systems, check your system vendor or `http://www.javasoft.com/cgi-bin/java-ports.cgi` for availability of a Java 1.1 compatible virtual machine.

Also, make sure that your environment variables (e.g. `PATH`) are set to include the Java programs. Try to run the following command:

```
borneo>java -version
java version "1.1.3"    // or whatever version you have
```

If the above command fails, check your Java installation. Note that HADES won't run under Java version 1.0.2—it requires at least Java version 1.1beta. However, because of several security related bug-fixes, you should upgrade to version 1.1.3 or newer as soon as possible

HADES download Second, you have to install the HADES software to your system. Unless you receive the HADES software on CDROM or magnetic media, this requires to download an archive file from our WWW-server `tech-www.informatik.uni-hamburg.de`. Use your favorite WWW-browser to download the file `http://tech-www.informatik.uni-hamburg.de/applets/hades/archive/hades.jar`. This is an archive in JAR-format which includes all Java-class files, documentation, and examples necessary to run HADES.

HADES installation After downloading `hades.jar`, you have to unpack the archive using `jar` to some directory, e.g. `C:\java\classes` on WINDOWS'95, or `/opt/java/classes` on Unix. `jar` is a general-purpose archiving tool available in Java 1.1. For example, assuming that you use WINDOWS'95 and downloaded the file `hades.jar` to the directory `C:\temp`, and want to install to `C:\hades\classes\`, you would issue the following commands on the DOS command line:

```
C:>cd c:\hades\classes
C:>jar xvf c:\temp\hades.jar
installing ...    // messages from jar
```

CLASSPATH The final installation step is to update your `CLASSPATH` environment variable, to allow your Java 1.1 runtime to find the newly installed classes. Unfortunately, the details of the `CLASSPATH` usage and syntax differ between operating systems and even different command interpreters. For example, on WINDOWS'95 you would add the following line to your `c:\autoexec.bat` file:

```
set CLASSPATH=%CLASSPATH%;C:\hades\classes;
```

three directories While future versions of HADES will include an interactive and flexible design and library browser, the current version of HADES locates all components and designs via the standard Java 1.1 “class loader”, and it expects all designs to be in subdirectories of a directory with the fixed name “`hades`”. Therefore, to enable HADES to find your own designs, you have to create another subdirectory called `hades` somewhere on your computer,

```
C:>mkdir C:\users\hendrich\hades
```

and you add the *parent* directory of the `hades` subdirectory to the `CLASSPATH` variable set by the startup script. To summarize, you need to include three directories with your `CLASSPATH` to run HADES with your own designs:

- the location of the Java 1.1 system classes, e.g. `C:\jdk113\lib\classes.zip`,

- the location of the HADES classes and example designs, e.g. `C:\hades\classes`,
- the parent direcotory of a subdirectory called "`hades`" for your own designs, e.g. `C:`**`\users\hendrich`**`\hades`.

You might consider running HADES from a script, to simplify the setup of the environment variables and directory settings. The example script below assumes that you installed Java 1.1 to `C:\jdk113`, the HADES classes to `C:\hades\classes`, and that you created the directory `C:\users\hendrich\hades` for your own designs. Then, the batch file for WINDOWS'95 might look like this:

Windows setup

```
rem hades.bat - HADES batch file for Windows'95/NT
rem
set PATH=%PATH%;C:\jdk113\bin;                          programs,
set CLASSPATH=C:\jdk113\lib\classes.zip;.               system classes,
set CLASSPATH=%CLASSPATH%;C:\hades\classes              hades classes,
set CLASSPATH=%CLASSPATH%;C:\users\hendrich             your designs
java -mx32M hades.gui.Editor
```

Edit the script as necessary to reflect your installation.

The following code shows the corresponding standard startup script for Unix, with Java 1.1 installed to `/opt/java` and HADES installed to `/opt/classes`. Also, the script assumes that you create a subdesign called "`hades`" in your home directory. You will have to save all your own designs to that directory or nested subdirectories, for HADES to find them:

Unix setup

```
#!/bin/csh -f
# Run Script for HADES 0.35, 24.09.97
set path = ($path /opt/java/bin)
setenv CLASSPATH /opt/java/lib/classes.zip:/opt/classes:$HOME:.
/opt/java/bin/java -mx32M hades.gui.Editor &
```

Note for experienced Java users: In order to minimize download time and network traffic, the file `hades.jar` is a packed `jar`-archive. Therefore, you cannot include the archive directly in your `CLASSPATH`. If you plan to use the HADES classes from within a `jar`-archive to improve performance, you will have to unpack `hades.jar` first, before recreating the archive using the `-0` option to `jar`. Also, because the current version of HADES won't be able to access the example designs from within a `jar`-archive, you shouldn't include the example directory with your newly created archive.

repacking the JAR

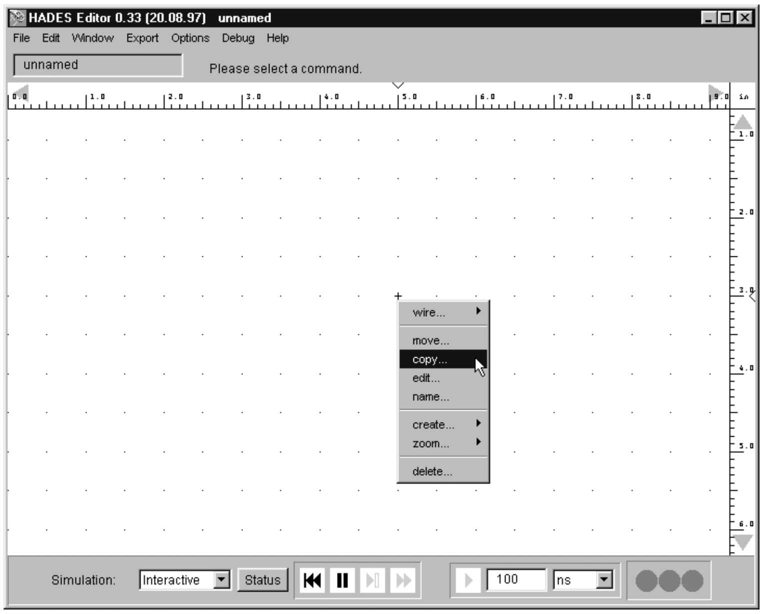


Figure 2: The HADES editor at start-up, with activated popup-menu. The main portion of the window is used by the so called 'object canvas' for editing. The panel above the object canvas displays the design name and status messages. The bottom panel is used to control (start/stop) the simulation.

2.2 Running HADES

After the installation steps described in the previous subsection, you are now ready to run HADES. From the command-line, either call the HADES startup script as described above,

```
C:>hades.bat
```

or start the Java virtual machine with the name of the HADES editor class:

```
borneo>java -mx32M hades.gui.Editor
```

If everything goes well, after some seconds the main editor window of HADES will appear on the screen. If instead you get an error message that class "x.y.z" was not found, see the preceding subsection on how to set the `CLASSPATH` variable.

the editor window explained

Initially, the HADES editor window should look similar to figure 2. The main part of the window is occupied by the 'object canvas' used for editing. The panel above the object canvas displays the design name (if any) and status messages. If HADES ever seems to hang or behaves unexpectedly, check the status message. Typically, you have initiated a command and HADES just needs (and waits for) more data to complete the command. You can interrupt and cancel each editing command by pressing the `ESC` key, if necessary.

simulation control

The panel below the object canvas contains several buttons for simulation control, see section 2.4 for the details. From left to right, the buttons have the following functions:

- `i` prints information about the simulator status, including the current simulation time and the number of executed and pending events,
- `<<` stops and resets ("rewinds") the simulator,
- `||` pauses the simulator,
- `>|` executes the next pending event ("single step"),
- `>>` starts or restarts the simulator, running "forever" until stopped or paused,
- `>` will run the simulation for the specified time interval

Usually, the simulation is running in interactive mode directly after startup, so that you can begin to edit and simulate right away. The "'traffic light"' at the right provides immediate visual feedback about the simulator status, whether running, paused, or stopped.

2.3 Editing Commands

Once HADES is running, most editing commands are available from either the main menu bar (on top of the editor window), a context-sensitive popup-menu on the editing canvas, or bind-keys (shortcut keys) for the most important functions.

The remainder of this section will explain the basic commands to create, edit, and connect components, and how to run a simulation. As an example, a simple D-latch flipflop will be constructed from four NAND-gates.

2.3.1 Using the Popup-Menu

While bind-keys are the preferred way to invoke editing commands for experienced users, the best way to start is to use the popup-menu. Moreover, in the current version of HADES, some commands are only available from the popup-menu.

the popup-menu

In an effort to provide the native look and feel for each platform, the way to activate a popup-menu is platform dependent in Java 1.1. For example, to activate the popup-menu on a Linux or Unix workstation running X-windows, you have to click and hold down the right mouse button until the menu appears. You can then move the mouse to the menu item you want to execute, and release the right mouse button. On a PC running WINDOWS'NT or WINDOWS'95, click and release the right mouse button. The popup menu will appear and you can select an item by clicking with left mouse button. Please consult your system provider for the rules on other platforms.

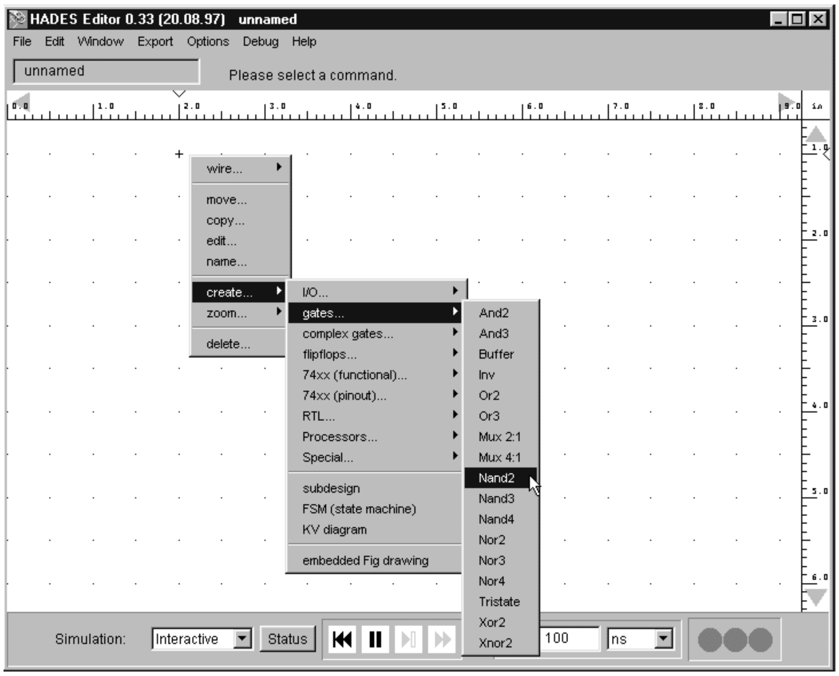


Figure 3: Selecting a Nand2 gate from the main popup-menu.

Now try to activate the popup-menu. It will appear at the current mouse position and should look similar to figure 2. It provides the following items:

wire invokes a submenu used to create and edit wires (signals), see section 2.3.5 below for details.
move move a component to a new position.
copy copy a component and select a position for the copy.
edit edit a component. This command will open a context-sensitive dialog window that allows to edit all user-settable properties of the given component.
name change the name of a component.
create create a new component. This actually is the root menu item for a tree of submenus to create gates, flipflops, I/O components, subdesigns, and several other simulation objects.
zoom activates a submenu that allows to change the current zoom factor.
delete delete the component at the mouse position.

Note that some menu items are hierarchical. Once you move the mouse to the little arrow symbols in one of these menu items, another popup-menu with additional menu items will appear. This allows to order the menu items logically, and it saves screen space, because only a fraction of all menu items has to be displayed.

2.3.2 Creating Components

As usual, digital circuits in HADES are constructed from predefined objects, called *components* in the remainder of this tutorial. The complexity of components may vary in a wide range, from simple transistors to logic gates and flipflops to complex subdesigns, e.g. memory or processors.

creating components

In order to build a D-latch flipflop, we just need four NAND-gates. (If you don't remember how to build a D-latch flipflop from NAND-gates, take a look at figure 6 on page 16, which shows the completed design). Therefore, activate the popup-menu and select the create submenu. A new popup-menu should appear which in turn contains additional submenus called *I/O*, *gates*, *complexgates*, *flipflops*, etc. As you might expect, the menu item to create a two input NAND-gate is to be found in the *gates* submenu. Move the mouse to the *Nand2* menu item and select it, see figure 3. This instructs HADES to load both the simulation model (Java class code) and the symbol (graphical representation in HADES) of a NAND-gate with two inputs. Depending on the speed of your computer, this process might take about a second.

placing components

The popup menu will disappear, and in place of it the HADES editor will display a small rectangle at the mouse position, which moves together with the mouse. This rectangle indicates the bounding box of the NAND-gate symbol. Move the mouse to the desired position (top left corner), and click the left mouse button. The editor places the NAND-gate at the selected position and displays the full symbol of the NAND-gate.

moving components

If you have dropped a component in a wrong position, you can always move it to a new position. Just position the mouse on one of the component's corners and activate the popup-menu, then select move. The bounding box rectangle will appear at the mouse positon, and you can move the component. Click the mouse to drop the component at the current mouse position. Just try to move the NAND-gate a little bit to the right, and then back to its original position.

copying components

For the complete D-latch design, four NAND-gates are required. Use the popup-menu to create and place the missing three NAND-gates required for the D-latch. If you need a lot of components of the same type, you might prefer the *copy* command instead of the *create* command. Move the mouse to the object to copy, activate the popup-menu, and select *copy*. HADES will create a new instance of the selected component class, and display its bounding box rectangle at the mouse position. Move the mouse to the position you want to place the new object, and click the left mouse button. Because the mouse now is positioned over the new object, you can instantly repeat the process to create the next object, etc.

deleting components

undo and redo

As you might have guessed, you can also delete components from your designs. Position the mouse on one of the corners of the component to delete, activate the popup-menu, and select *delete*. HADES will delete the component without further notice, but in an emergency you should be able to undo the delete command by selecting *undo* from the *Edit* menu. While HADES will eventually include support for unlimited recursive *undo* and *redo*, at the moment this works reliably for some commands only. As a safety measure, and unlike the commands move or copy, the *delete* command will only work if you position the mouse exactly on the component's corner.

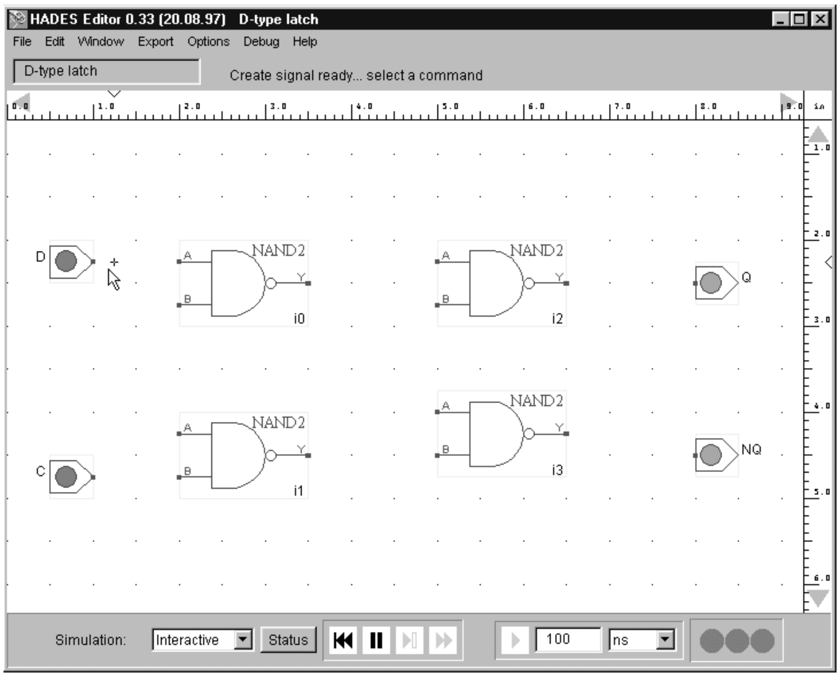


Figure 4: The D-latch design with all components before wiring

2.3.3 Adding I/O-Components

I/O pins Next, it is necessary to specify the inputs and outputs of your design. As most design systems, HADES uses special components to indicate the inputs and outputs of a design. These components can to be found in the *I/O* submenu of the *create* popup-menu. The most important are *Ipin* for an input pin and *Opin* for an output pin of a design. Ipin and Opin are also used to build up hierarchical designs and to specify input values to your design during simulation. The D-latch design requires two Ipin components for the D (data) and C (clock) inputs, and two Opin components for the Q (flipflop output) and NQ (inverted output) outputs. Use the popup-menu to create and place these components.

changing names Next, move the mouse to one corner of the data input Ipin symbol, activate the popup-menu, and select *name*. A dialog window appears that allows you to specify or change the name for the component. While HADES will create unique component names automatically, names like “I42” are not usually considered good style, and are certainly not acceptable for key components like I/O pins. Good practice demands that the inputs for flipflops are called *D*, *C* or *clk*, while the outputs are often called *Q* and *NQ*. Change the names of the Ipins and Opins correspondingly. Your design should now look similar to figure 4.

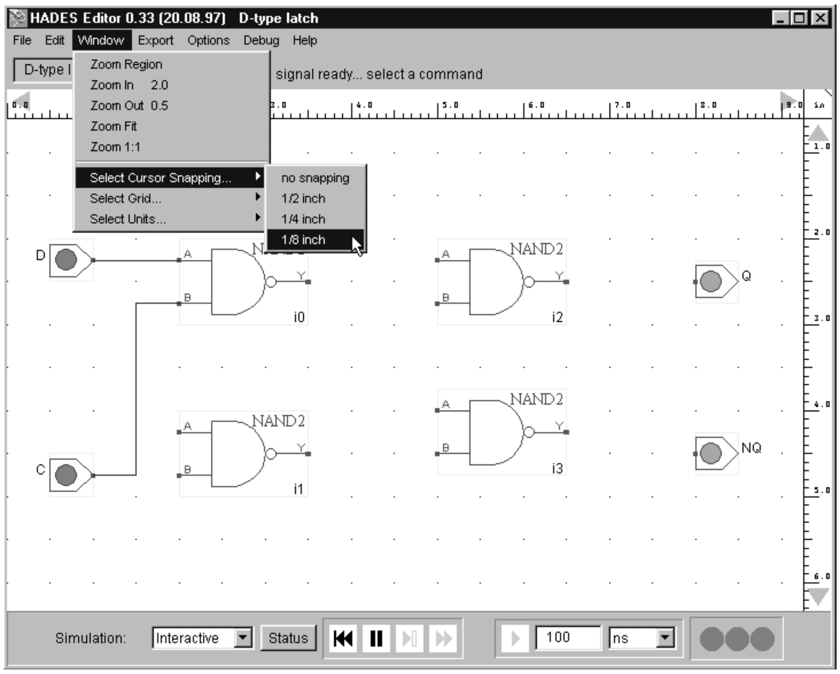


Figure 5: Selecting the cursor snap grid

2.3.4 Display Control

two cursors

You will have noticed that HADES actually displays two cursors: the standard mouse cursor that traces the mouse position exactly, and a little cross which indicates the nearest *grid-point* next to the mouse position. Because the little cross is snapped to the nearest grid point, it is much easier to position exactly than the system cursor. This helps a lot to place and align components, and to connect and align wires (see below).

cursor snapping

Unfortunately, it is not possible to hide the system cursor in Java 1.1, so you have to live with both cursors. Some operations, like activating the popup-menu, use the system mouse position, but most editing commands use the grid-snapped cursor. You can change both the grid display and the cursor snapping via the main editor menu. Select the *Window* menu and either the *Select Grid…* or the *Select Cursor Snapping…* submenus. The default is to snap the cursor to points 1/4th of an inch apart, which is useful for most of HADES components. However, some components like the four-input NAND gate actually use ports that lie on a 1/8th inch grid. To use such components you may have to change the snapping grid to 1/8th of an inch, compare figure 5.

zooming and panning

To change the current *viewport*, that is, the area visible on the screen of a large circuit schematic, use the corresponding menu items from either the *Window* menu or the *zoom* submenu from the popup menu. For the *zoom-region* command you are prompted to click on two opposite corners of the region to zoom into. The *zoom fit* command tries to fit the whole design into the current viewport and the *zoom 1:1* command restores the default zoom-factor, where visible coordinates on a 72dpi screen should match the real size. It also restores the origin of the viewport to the point $(0, 0)$. The rulers on the top and right sides of the object canvas mark the current coordinates (in inches). To move the viewport over a large schematic, either use the cursor arrow keys, or click on the gray triangles inside the rulers. This will move the visible area about 1/5th along the corresponding direction.

negative coordinates While HADES allows to place objects anywhere you choose, you should probably restrict objects to the positive quadrant. For example, the program `fig2dev` used to export HADES schematics to FIG format, Postscript, and other formats, silently clips objects with negative coordinates. If necessary, you can always move objects back "into place".

2.3.5 Connecting Wires

simple wires Once all or some components are placed in a design, it is time to connect them with wires, called *signals* in HADES. This is only possible at the points called *ports*, indicated by small red rectangles on the component's symbols. To connect a new wire to a component, move the mouse to the corresponding port position, and click the mouse. This creates a new wire and connects its first end to the component port. If you move the mouse, you will see it traces a rubber-band, to indicate the current position of the wire. Click the mouse on any intermediate points you need for the wire, then move to its destination port and click the mouse again. This attaches the wire to the port, and the signal is ready to use.

Try to create the wires between the *D* and *C* Ipin components and the corresponding gate inputs of the NAND-gates for the D-latch circuit. Because of the cursor snapping, it should be quite easy to move the mouse to the component ports and correctly place intermediate points. Also, note that HADES won't enforce Manhattan geometry for wires. For some circuits, like the D-latch flipflop, diagonal wires actually improve the readability of the circuit schematic.

2.3.6 Adding Wire Segments

complex wires Many wires, however, are not just a point-to-point connection, but require connection to many ports. For example, in the D-latch circuit the outputs of the SR-flipflop NAND-gates need to be connected to an output Opin and an input of the other NAND-gate each. To create these signals, you would first create a simple point-to-point connection as described above. Afterwards, additional segments are added to the signal. Move the mouse to one of the *vertex points*—either or the start, end, or intermediate points—of the wire, activate the popup-menu and select *wire...*, then *add segment*. HADES will create a new wire segment attached to the original wire. As above, specify any intermediate points needed, then click on the component port to connect the wire. Remember to first click on a point of the existing wire, then connect to the new port. This has to be done in the right order, as HADES won't let you connect a signal to a port already in use.

2.3.7 Moving Wire Points

moving wire points

Sometimes, you may want to move signal vertices after initial placement. This is possible with the *move point* command. Position the mouse over the wire vertex you want to move, activate the popup menu, select the *wire...* submenu, then the *move point* command. Move the mouse to the new position for the wire vertex and click the left mouse button. The editor will move all wire segments connected to that vertex from the old to the new position. Another way to invoke the *move point* command is the `o` bindkey. Just position the mouse on the vertex to move, press the `o` key, move the mouse to the new position, and click the left button. Please excuse that the rubber-banding throughout the *move point* command is implemented only partially yet.

Note that the HADES editor will try to keep all wire segments connected if you move a component, which is useful to show the connectivity of your circuits. However, the editor does not contain an "auto-router" to find new and visually appealing paths for the wires. Instead, the editor will only move the very last vertex of each wire to the new position of the component's port it is connected to. Therefore, after moving a component, you may want to move other (intermediate) vertices of a wire too, to update your circuit schematic.

2.3.8 Wire Editing Features

frequent mistakes

Sometimes, HADES may appear to hang while creating wires or adding wire segments. In this case, look at the status messages. The usual reason is that HADES waits for you to specify a valid starting or end point for the wire. Also, you might have clicked beside a port, which could initiate some undesired command like moving. In the latter case, press `ESC` to cancel the annoying command.

First, to catch a frequent and difficult to find error, HADES won't connect a new wire to a port already connected. Instead, you have to use the *add segment* command explicitly to do this.

Second, in the current version of HADES, it is not possible to connect new wire segments to the middle of an existing wire. The corresponding editor function will only recognize wire vertices. Therefore it might be a good idea to include some intermediate points even into a straight line wire, in order to be able to connect additional segments later. This is a known bug.

Third, so far, the HADES editor is not able to delete only a part of wire. If you have accidentally connected a wire to a wrong port, you will have to delete the entire wire. This is also a known bug. To delete a wire, position the mouse over one of it's vertices, activate the popup-menu, and select *wire...*, then *delete*. You can also delete a wire by

deleting wires

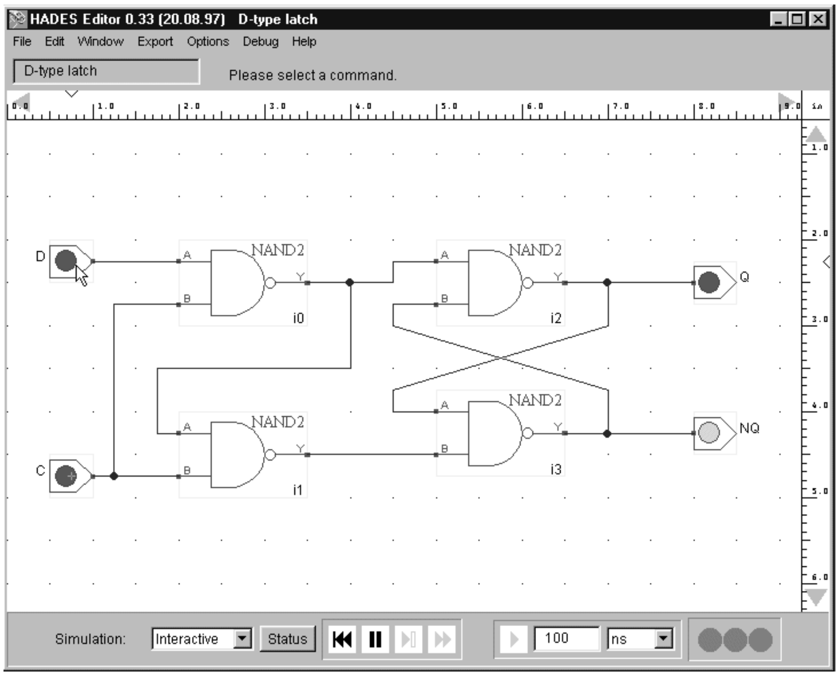


Figure 6: The completed D-latch design with all components and wires. During interactive simulation, you can change the input values to your circuit by clicking on the *Ipin* symbols.

positioning the mouse over one of it's vertices and pressing the `DEL` key.

bindkey w Now create all the wires necessary for the D-latch circuit. Because adding segments to wires is a very frequent operation, even in this small example, you might want to know about the `w` bindkey. Pressing the `w` key will initiate a *add segment* operation without the need to invoke the popup-menu. Once you are finished with all wires, your schematic should look similar to figure 6.

2.3.9 Loading and Saving Designs

saving designs This is a good time to save your design. Because you did not specify a name for the design until now, you should select *Save As...* from the *File* menu which will open the standard file selector dialog. Ultimately, our planned design and library browser will decouple the HADES design names form platform dependent file names. At the moment, however, HADES still stores design files directly in normal (text) files. Therefore, enter a suitable filename, e.g. `dlatch.hds`, and click the *OK* button. For a larger project with many subdesigns, you should probably consider to use subdirectories. By convention we use the extension `.hds` for HADES design files, which matches the German pronunciation of HADES. Moreover, we use filenames of the form `designname.sym` for the HADES symbol corresponding to a design `designname.hds`.

naming convention

Once you have saved your design, HADES should remember the filename, so that you can use the shorter *Save* command from the *File* menu afterwards. Because HADES doesn't support version management yet, it might bee a good idea to save different stages of your design to different files, e.g. `dlatch-1.hds`, `dlatch-wired.hds` etc. By the way, the default filename is `unnamed.hds` in the current working directory.

As you have saved your design now, you might as well try to read it back. Select the *New* menu item from the *File* menu, which instructs HADES to start a new design. Note that the *New* command erases any previous design. Now select the *Open* menu item from the *File* menu, which again will open a file selector dialog. Enter the file name for your D-latch design and click *OK*.

loading designs

Unfortunately, parsing design files is quite slow at the moment. Even loading a small circuit like the D-latch design may take a few seconds, and medium sized circuits like `DecoderFSM.hds` from the `examples/dcf77` directory will need about 20 seconds to parse and load even on a Pentium/166. This is one aspect where the flexibility of using standard text files in connection with dynamic class loading leads to a very noticeable performance hit. Therefore, we already consider to switch to a binary external representation for HADES designs. However, while annoying, loading times are often negligible in comparison with the time spent on editing and simulation. For example, it took me about three hours to edit and debug the `DecoderFSM` design.

be patient

2.3.10 Component Properties

Most components in HADES offer an interactive dialog to get and set some or all of their parameters. Position the mouse over a component, activate the popup menu, and select the *edit...* item. A *property sheet* dialog window will appear and list all user-settable properties of the selected component. The dialog window is not modal, so you can open many property dialogs (for different components) at once. Move the cursor to the textfields in the dialog window, and change the property values as desired. Click the *Apply* button to change the object's properties as specified, but to keep the dialog window open. Click *OK* to apply the values and to close the dialog, or click *Cancel* to just close the dialog window.

properties and configuration

One example of the property dialog for a *clock generator* component is shown in figure 7. The clock generator allows to change its name and all of its timing parameters. As another example, you can specify the gate delay of the basic and complex gates interactively. You might try to specify delays in the range of seconds in combination with *glow mode* (see below) to get a feeling of both discrete event simulation, and the behavior of digital circuits.

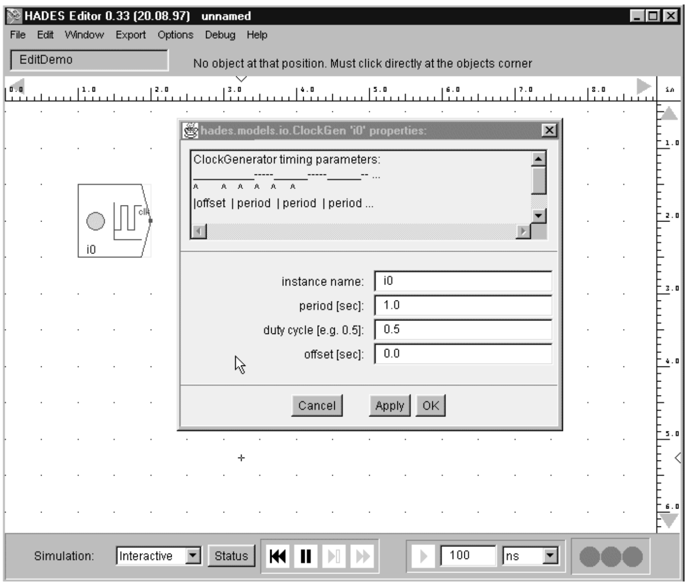


Figure 7: An example of the visual editing capabilities in HADES: A clock generator and its property sheet, allowing you to set and change its name and its timing parameters.

2.4 Simulation

With HADES, you can simulate a design interactively during all stages of the design process. The following subsections explain the logic model (IEEE StdLogic1164) used in the simulator, how to apply values to your circuits in interactive simulation, and the commands used to control the HADES simulation engine.

2.4.1 Digital Simulation and StdLogic1164

This subsection explains the logic model used in the HADES simulation engine. It is based on the 9-valued *StdLogic1164* logic model [IEEE-93]. Please skip the next few paragraphs if you are familiar with digital simulation or the StdLogic1164 logic model.

simulation models It should be obvious that any digital logic simulation is used to simplify the complex behavior of real electronic circuits, even beyond the approximation by differential equations for voltages and currents. The most simple approximation is a logic model with only the two values `0` and `1`. This model, however, is often inadequate, because it fails to model several typical scenarios in digital circuit design. For example, a simple `0/1` logic cannot model *short-circuit* situations, where multiple gates drive a signal with different values, or a *high-impedance* state, where a signal is not driven

value	meaning	color
U	undefined and never initialized	cyan
X	undefined during simulation	magenta
0	logic '0'	light gray
1	logic '1'	red
Z	not driven, high impedance	yellow
L	weak logic '0'	middle gray
H	weak logic '1'	dark red
W	weak undefined	yellow
D	undefined, but don't care	—

Figure 8: The logic values used in HADES, based on StdLogic1164 [IEEE-93], and their associated colors in *glow mode*.

by any output. Also, it is important to guarantee the initialization of a circuit, and the re-initialization after a short-circuit or high-impedance state. A simple solution that allows to model these situations in digital circuitry is to add corresponding values to the underlying logic model.

StdLogic1164

An important example of such logic models is the 9-valued logic model called *StdLogic1164* used in the VHDL simulation language [IEEE-93]. Figure 8 shows the logic values in StdLogic1164 and their meanings. At the start of a simulation, all signals and gates are initialized to the `U` state (undefined) in StdLogic1164. Because any logic gates will propagate `U` values if they receive `U` values, the circuit won't start to work, unless it is correctly initialized, e.g. by applying reset signals to all flipflops in the circuit.

X propagation

Another important example of the use of multiple valued logic systems is the reaction of flipflop models to a timing violation. In a pure `0/1` logic system, a flipflop cannot signal any timing problems to the rest of the circuit, and such errors would therefore pass unnoticed. In StdLogic1164, however, a flipflop whose hold- or setup-times are violated will simply generate a `X` value at its outputs, and propagate this value to all gates connected to its outputs. Because `X` values, like `U` values, will spread around your circuit, the errors are not only easily spotted, but rather hard to ignore. This behavior might seem annoying at first, but it really helps debugging circuits, especially under tight timing constraints.

2.4.2 Glow Mode

glow mode

As an aid to circuit debugging, HADES supports a color-encoding of the logic values in your circuit. This idea is borrowed from DIGLOG, and it is also called *glow mode*. In *glow mode*, all signals are drawn in a color corresponding to their current logical value, compare figure 8. You can toggle between *glow mode* and normal mode (where

NOT a	
a	y
U	U
X	X
0	**1**
1	**0**
Z	X
W	X
L	1
H	0
-	X

a AND b									
a / b	U	X	**0**	**1**	Z	W	L	H	-
U	U	U	0	U	U	U	0	U	U
X	U	X	0	X	X	X	0	X	X
0	0	0	**0**	**0**	0	0	0	0	0
1	U	X	**0**	**1**	X	X	0	1	X
Z	U	X	0	X	X	X	0	X	X
W	U	X	0	X	X	X	0	X	X
L	0	0	0	0	0	0	0	0	0
H	U	X	0	1	X	X	0	1	X
-	U	X	0	X	X	X	0	X	X

Figure 9: The NOT and AND functions of the StdLogic1164 logic system. Note that StdLogic1164 includes the standard boolean operations with arguments `0` and `1` as a subset (shown in boldface). Also note that many input combinations result in `U` or `X` outputs.

all wires are drawn in a blue color) at any time by selecting/de-selecting *glow mode* from the *Options* menu, or by pressing the `g` bind-key. The main drawback of using *glow mode* throughout a simulation is the significantly slower simulation performance, due to the high rate of signal repaints.

2.4.3 Interactive Simulation and Switches

Most digital simulators run the simulation in batch-mode. After designing your circuit you have to write a file with the *input stimuli*, that is a description of the values to apply to the input pins at specified times. This file is fed into the simulator, which in turn generates a trace of all values you requested. Finally, you would use other tools to analyze the simulation trace.

interactive simulation

While HADES supports batch-mode simulation, the preferred way to explore and debug your circuits, is to "play" with the circuit in *interactive simulation* — at least in the early design phases. In this mode, also borrowed from DIGLOG, all *Ipin* components and several other components from the HADES *I/O* library work like small switches. Figure 10 shows a collection of the *interactive* input and output models. You use the *Ipin*, *pulse switch*, *power on reset*, *clock generator*, and *hex switch* components to generate input values for your circuit. The *Opin*, *LED*, and *HexDisplay* components allow to visualize the output values generated by your circuit.

Ipin = switch

If you click on an *Ipin* with the mouse, it will toggle its state and propagate the corresponding value to the circuit. The current state of the *Ipin* is displayed in the same colors as signal values in *glow mode* (see above). This means that an undefined switch (never clicked on since the start of the simulation) will be shown in magenta, an inactive switch ('`0`' level) in gray, and an active switch ('`1`' level) in red.

0 - 1 - 0

Note, that all *Ipin*s are initialized to the *undefined* (`U`) state at the start of the simulation.

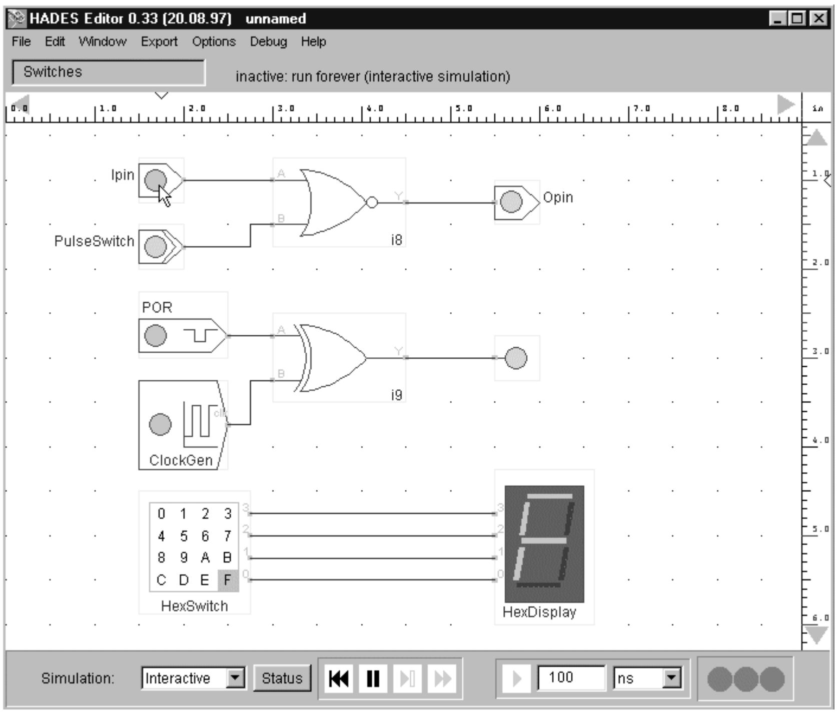


Figure 10: A collection of interactive I/O components: Ipin (switch), pulse switch, power on reset, clock generator, hex switch, Opin (LED), LED, and hex display.

On the first click, an *Ipin* will change its state to `0`, on the second click to `1`. Afterwards, it will toggle between `1` and `0`. Because the *Ipin*s start in `U` state, you will usually have to click on all *Ipin*s at least once to initialize your circuit with useful values. If you hold down the `SHIFT`-key while clicking on an *Ipin*, it will use another sequence of states, namely `U` to `0` to `1` to `Z` to `X` to `0`, etc. This allows you to specify `Z` or `X` inputs to your circuits if necessary. *0 - 1 - Z - X - 0*

Some other models from the *I/O library* also work as switches, or can be controlled by clicking with the mouse. Each time you click on a *pulse switch* component, it will send out a short `1` pulse, before returning to the `0` state. The duration of the `1` pulse can be specified in the *pulse switch* properties from the *edit...* popup menu. *pulse switch*

In normal operation, the *power on reset* component sends out one `U -> 0 -> 1` pulse at the start of the simulation. After this pulse is finished, you can set the output value of the *power on reset* just like an ordinary switch. Again, the current output value of a *power on reset* instance is indicated by a gray or red circle. *power on reset*

Clicking on a *ClockGen* component, however, will not directly set its output value. Instead the clock generator's internal state is toggled between started (indicated by *clock generator* a red circle) or stopped (indicated by a gray circle). Note that a started *ClockGen* generates simulation events even when no other components are connected to its output. Therefore, it may be useful to stop all clock generators while you are editing and restart them for simulation only.

Opin = LED Similarly, the *Opin* instances in your design work like little LEDs, displaying the value of their input signal. If you want to visualize some signal value in your circuit without having an explicit output connection, you should use *LED* components instead of *Opin*s. Note that you can select the color of LED components.

hex switch, hex display The *hex switch* and *hex display* components are useful to control and display hexadecimal (sedecimal) values. Click on a value in the *hex switch* to select the corresponding four-bit output values. The *hex switch* generates a four-bit `U` signal until you first select one of the `0` to `F` values.

2.4.4 Starting and Stopping the Simulation

In the current version of HADES the simulator is started automatically after initial startup and each time a new design is loaded into the editor. This allows you to "play" with your circuits right away from the start. However, in circuits with a high-event rate, editing may become slow while the simulation is running. In these situations you should manually pause or stop the simulator while editing, and restart it for testing and debugging your circuits. To control the simulator, use the buttons in the simulation control panel at the bottom of the editor window. From left to right, the buttons have the following functions:

control buttons

- `i` prints information about the simulator status, including the current simulation time and the number of executed and pending events,
- `<<` stops and resets ("rewinds") the simulator,
- `||` pauses the simulator,
- `>|` executes the next pending event ("single step"),
- `>>` starts or restarts the simulator, running "forever" until stopped or paused,
- `>` will run the simulation for the specified time interval

Note that the buttons provide visual feedback of their status—inactive buttons are "grayed out" while enabled buttons are drawn in black and white. For example, in figure 6 the simulator is running, so that only the `||` and the `<<` buttons are enabled.

2.4.5 Single-Step Simulation

single-step mode For debugging, HADES also allows to *single-step* a paused simulation through all pending events. Each click on the `>|` button will process the next single event from the event list. As usual, the button will be disabled and "grayed out" if no further events remain. The single-step mode is sometimes quite useful for demonstration purposes of simple circuits—especially in combination with *glow mode*, to indicate the signal changes after each event.

However, single-stepping through a deeply nested hierarchical circuit is not very useful at the moment, because most events occur on signals not visible in the top level circuit schematic. We plan to improve the debugging functionality soon.

2.5 Waveforms

waveforms = signals vs. time

Because of the complex timing involved, the presentation of the signals *waveforms* (signals vs. time) is essential for the understanding and debugging of most digital circuits. While HADES needs to keep track of the most recent value of each signal to propagate this value to all components connected to a signal, normally it will not trace and store any previous values. Therefore, if you are interested in waveforms for some signal, you have to add a *probe* to that signal.

2.5.1 Adding Probes

adding a probe

To add a probe to a signal visible on the top level schematic of your design hierarchy, just move the mouse to a vertex of that signal, activate the popup menu, select *wire...*, then select *add probe*. The editor will mark the selected signal with a small *probe symbol* that also displays the signal's name. If necessary, the editor will create and display the waveform display window *waves*. The probed signal will then be added to the end of the signals visible in the waveform display window. Internally, a data structure is created and used to store all subsequent events on that signal.

hidden signals

Because not all signals are visible on the top level schematic of a hierarchical design, the HADES editor also allows to add probes to signals by name, where signal names are constructed similar to Unix filenames, starting from the top level design. For example, the name `/clock/counter1/DFF1.Q` might specify the output signal of a D-flipflop in the subdesign `counter1` of the top level design `clock`. As a convenience, the editor also allows you to add probes to all top level I/O signals, to all top level signals, or to all signals in the design hierarchy from the main *Edit* menu.

memory problems

Depending on the number of signals traced and the number of events on these signals, HADES may run out of memory very soon, because it needs to store the full history of the signal values. Per default, the Java virtual machine is limited to a heap size of 16 MBytes of memory. If necessary, you can increase the heap size with the `-mx` option to the Java interpreter, e.g. `java -mx32M hades.gui.Editor`. Another solution is to limit both the number of signals traced and the simulation time interval during which the traces are active.

2.5.2 Waveform Display

planned... and available waves

While we plan to include an advanced waveform viewer with many high-level func-

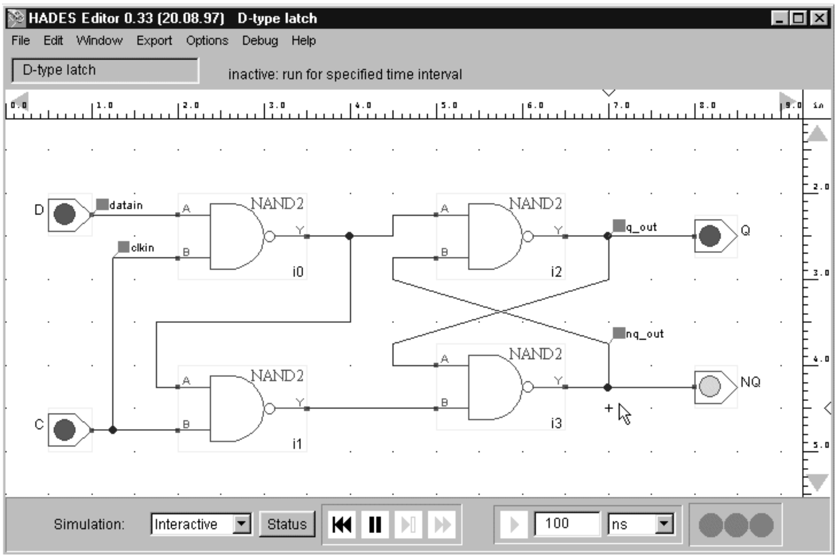


Figure 11: The completed D-latch design with probes at the input and output signals

tions for searching and data-analysis in future versions of HADES, the current version supports only a minimal functionality. Figure 12 shows example waveforms for the D-latch circuit from figure 11. The HADES *Waves* display allows you to show all waveform data or to zoom into a specified time window. Also, it is possible to measure times, time differences, and to identify the corresponding signal data values. *Waves* uses a crosshair-cursor to simplify time comparisons between different waveforms.

zoom and redraw Note that the waveform display is not updated automatically during a simulation. You might have to change the zoom factor or request an explicit *redraw* to see the current waveform data. From the *Display* menu, select *Zoom Fit* to display the whole time window of the simulation, from start time to the current end time. Select *Zoom In* or *Zoom Out* to half or double the time window. Press the `l` or `r` keys to pan to larger or smaller times.

regions of interest To zoom into a specific region of the waveforms, move the mouse to the center of that region. Then press the `z` or `Z` keys to zoom out or in around the time selected by the mouse position. Each key click increases or decreases the visible time window by a factor of two, which allows to rapidly zoom into an interesting part of the waveforms. During interactive simulation of gate-level circuits, however, you might still have to press the `Z` key some thirty times, before the effects of gate-delays become visible in the waveforms. This is due to the very different time scales—gate-delays are in the nanoseconds range, while input switching occurs on the order of seconds.

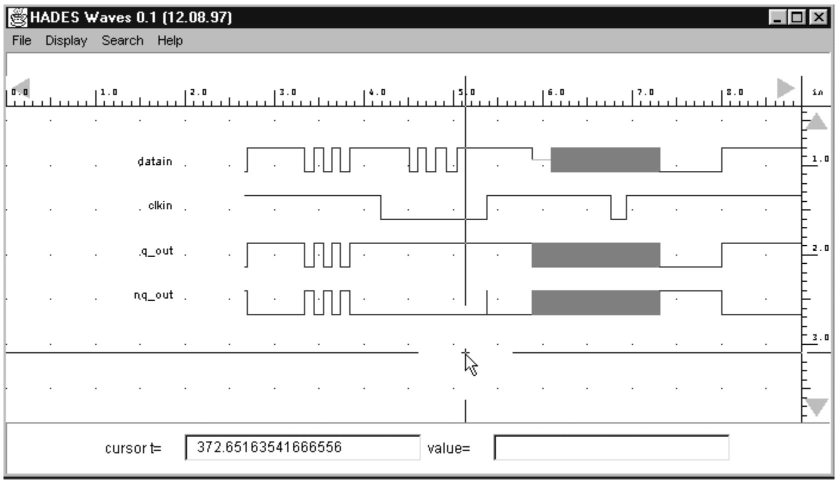


Figure 12: Example waveforms for the D-latch

2.6 Hints and Tips

2.6.1 Unexpected Timing Violations

flipflop timing violations explained

During interactive simulation, all events generated internally by your design are scheduled together with the user-events generated by clicking on input components like *Ipins*. On any user-event, the current simulation time is used as its time stamp. This behavior may result in unexpected faults in your simulation if the user-event feeds the input of a flipflop, which might detect a setup- or hold-time violation with respect to its clock signal and will generate and propagate a `X` value on its output.

metastability

If such a timing violation occurs, you may have to re-initialize the flipflops concerned or even the whole circuit, and to restart part of your simulation. Unfortunately, there is no simple solution to this problem except to use batch-mode simulation, where all input changes occur on predefined times with respect to the clock signals. Note that real circuits also suffer from these timing problems, because the timing of input signals can never be guaranteed. However, real flipflops won't signal the timing problem with a defined `X` state, but may enter an undefined *metastable* state with unpredictable duration and output value. The common way to signal these problematic situations in a digital simulation is to enter the `X` state.

2.6.2 Circuit (Re-) Initialization

At the start of a simulation, you will have to initialize all input values of your circuit, and possibly also some internal signals. As described in subsection 2.4.3, you can set and change the values of input components and therefore the signals connected to them directly by clicking. To specify the value of some internal signal in your circuit, position the mouse over one of its vertices, activate the popup menu, select the *wire…* submenu and then *set (zero)* or *set (one)* to set the signal value to `1` or `0` respectively.

setting signal values

Another possibility to initialize a circuit is to use the *initialize all signals…* menu from the main *Edit* menu. These commands will initialize all signals in your circuit at once, with either the common values `0`, `1`, and `U` for all signals, or with randomly generated `0/1` values. Especially the latter command is a good way to model the initial state of a real circuit, where initial values are largely dependent on small variations in device geometry and physics.

3 Commands

This section introduces some of the advanced editing commands in HADES. Most importantly, section 3.1 explains how to build hierarchical designs in HADES.

3.1 Hierarchical Designs

subdesigns

For circuits with more than a few dozen gates, most designers will use a *hierarchical* design, where the complete circuit is partitioned into smaller blocks. These blocks, called *subdesigns*, and are built from either the basic components like gates or nested subdesigns in turn. Obviously, the whole design then forms a tree with the top-level design at the root, an arbitrary set of subdesigns as internal nodes, and basic components as the tree's leaves. Each subdesign is a complete HADES design in itself, but only it's *symbol* is used in the circuit schematic of the parent design.

three advantages

The main advantages of a hierarchical design style are to reduce design complexity and to increase design reuse. First, each of the subdesigns is designed as a separate circuit, with a much lower number of components and subdesigns than the complete design. Second, each subdesign can be simulated for itself, catching most errors very soon in the design cycle and in small circuits. Third, subdesigns can often be reused both in one top-level design or among several separate top-level designs. Naturally, the most prominent examples of reusable components are the basic gates, complex gates, and flipflops. Other well-known examples are special registers, functional units like counters, encoders, decoders, and interface components.

top-down
bottom-up

Frequently, two different approaches to build the design hierarchy are distinguished. The *top-down* approach starts to partition the specification of the complete circuit into smaller units, which in turn are further partitioned, until each subdesign is small enough to be realized from basic components. The *bottom-up* approach starts to build small subdesigns from basic components, and integrates these subdesigns to larger subdesigns, and finally to the complete circuit. Usually, both design styles are mixed, because several iterations through the design phases are required. Also, refinements to the original specification further tend to hide the differences. HADES supports *bottom-up* design directly, but also the *top-down* design methodology.

a four-bit counter

As an example, assume that you need a synchronous four-bit counter with `reset` and `enable` inputs and `carry` output. One solution is to design this circuit directly from gates and D-flipflops. The more modular solution is to build the four-bit counter from four one-bit counters. Each one-bit counter needs both an `enable` input and a `carry` output, so that you can carry-chain the one-bit counters to form the four-bit counter — and counters of any other length as well.

the one-bit counter

One possible realization of this one-bit counter circuit is shown in figure 13. Besides the `enable` input it uses a `clock` and a `nreset` input, and it generates the output `dout` and the carry bit `cout`. As usual, *Ipin* components are used for all inputs, and the outputs are realized with *Opin* components. Try to design this circuit for yourself, save it as `counter1.hds`, and finally simulate it.

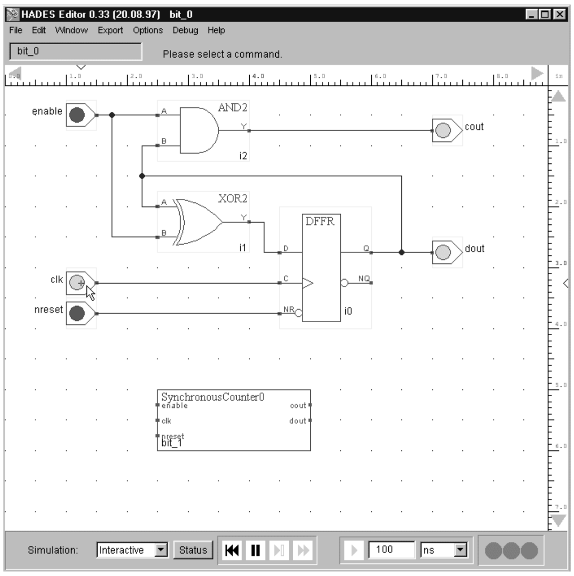


Figure 13: A simple synchronous one-bit counter with enable and asynchronous reset. At the bottom, the symbol generated for this circuit.

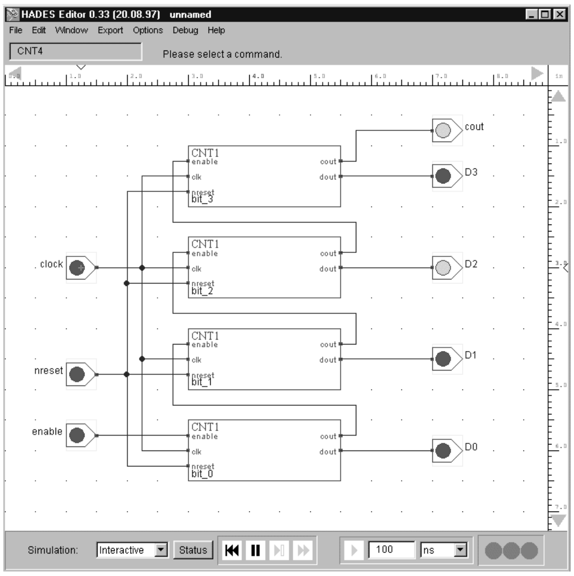


Figure 14: The hierarchical design for the four-bit counter, built from four one-bit counters.

symbol creation

In order to reuse the one-bit counter as a subdesign in another circuit, you have to create a *symbol* for it, which is used to reference the subdesign from the top-level circuit's schematic. To create the symbol, just select *create symbol…* from the *Edit* menu. This command searches the current design schematic for all *Ipin* inputs and all *Opin* outputs. It then generates a simple HADES symbol file that contains a suitable bounding box, the designs name, all inputs on the left side and all outputs on the right side of the bounding box. The order of the inputs and outputs in the symbol is directly determined from the position of the *Ipin*s and *Opin*s in your schematic. Note that other *I/O* components like clock generators, a power-on-reset, or LEDs will not be included in the symbol!

naming convention

After symbol generation, you are prompted for the name of the symbol file. In the current (beta) version of HADES, the editor expects a symbol file for the design `designname.hds` to have the name `designname.sym`. Please respect this naming convention and enter the corresponding name into the file dialog window. Be careful not to overwrite your existing design file!

If necessary, you can edit the symbol file with a text editor, to update or change the symbol representation. To allow a direct graphical editing of the symbol, we plan to include a symbol editor with later versions of HADES.

creating subdesigns

You are now ready to design the four-bit counter circuit. Select the *Edit* menu, then *New* to erase any previous schematic and to start a new design. To create the first subdesign, move the mouse to an appropriate position, activate the popup menu, select *create…* and then *subdesign*. Shortly, the file selector dialog will appear. Change to the correct subdirectory, if necessary, then enter the name of the subdesign (`counter1.hds`) and select `OK`. The HADES editor will now load both the subdesign and its symbol, and it will place the symbol at the selected position. As any other components in a circuit schematic, the subdesign is labeled with a unique instance name. Naturally, the subdesigns name can be changed via the *name…* item from the popup menu. To create the remaining three one-bit counter subdesigns, you can either repeat the above steps, or simply use the *copy* command three times. Position the subdesigns with the *move* command to find a visually appealing layout.

Now create the *Ipin* and *Opin* components for the four-bit counter circuit, namely the inputs `enable`, `clock`, and `nreset`, and the outputs `d0`, `d1`, `d2`, `d3`, and `cout`. Connect the output of the `clock` and `nreset` Ipins to the `clock` and `nreset` inputs of all four one-bit counters. Then connect the outputs and the carry-chain. Your circuit will probably look similar to figure 14. Simulate the four-bit counter to verify that the hierarchical design works correctly.

hierarchical or flat?

Because of the additional steps like symbol creation involved, using the hierarchical design may not be much faster than the *flat* design in this small example. The advantages of hierarchical designs become much more noticeable for larger projects. Just imagine to design a 128 bit counter… With a hierarchical design, you would probably reuse the four-bit counter to build a 16 bit or 32 bit counter, and combine these to the 128 bit counter. This should only take you a few minutes, if at all.

Also, you may have noticed that the HADES editor gets much slower when the number of visible objects is large, because of the large number of individual graphical objects (labels, boxes, lines, etc.). With a hierarchical design, only the few top-level symbols and objects are visible, which dramatically improves editing speed. In short, use design hierarchy as much as possible.

3.2 Editor Commands

3.2.1 Bindkeys

To improve the usability of the schematics editor, HADES provides several standard *bindkeys* as shortcuts for the most important drawing functions. Typically, you will keep one hand on the mouse while pressing the bindkeys with the other hand, which allows for very fast editing. Unfortunately, the standard Motif or Windows accelerator key mapping is not optimal in this case, because these accelerators combine the `ALT` and `CNTL` keys with other keys. Therefore, HADES does not use the `ALT+X` combinations, but reacts to simple keys alone. Figure 15 shows the table of the available predefined bindkeys. In the current version of HADES, it is not possible to change this mapping or to define new bindkeys, sorry.

3.3 Simulation Commands

3.4 Waveform Commands

3.5 Design and Library Browser

3.6 Scripting

3.7 Import and Export Commands

3.7.1 FIG format

3.7.2 Postscript

3.7.3 VHDL

Key	Function
0	set a signal to ’0’ level
1	set a signal to ’1’ level
A	’autoconnect’ the object at the mouse position
c	copy a component
e	edit a component’s properties
f	zoom fit
F	zoom full (100%)
g	toggle glow mode
m	move a component
n	change a component name
N	change a signal name
o	move a signal vertex
p	probe the selected signal
P	remove a signal probe
r	force a redraw
v	insert a new vertex into a signal
w	create a wire segment attached to a signal vertex
W	create a wire (possibly unconnected) at the mouse position
x	delete a wire segment at the mouse position
y,z	zoom out, centered (71%)
Y,Z	zoom in, centered (141%)
DEL	delete a component/wire
ESC	interrupt and cancel any command

Figure 15: Bindkeys in HADES 0.43, sorted alphabetically

4 Writing Components

This section explains how to write your own simulation models and how to integrate them into HADES. Unlike the other sections of this tutorial, this section assumes that you have a solid background as a Java-programmer. After a short explanation of the simulation concepts, three examples are presented and discussed in detail. First, an AND2 gate is used to illustrate the basic methods of a HADES component with very simple functionality. Next, the simulation model of an edge-triggered D-flipflop is presented. Finally, the model of a clock generator shows how to use both `WakeupEvents` for periodical activities and a user-specified configure dialog.

writing your own models

three examples

4.1 Basic Concepts

All simulation models in HADES are realized as subclasses from the generic base class `hades.simulator.SimObject`. This class fulfills a double role. On one hand, it defines the basic interface required for all simulation models. On the other hand, it provides a default implementation of these interface methods, which may be reused by all subclasses. You might argue that a separation between an abstract class to define the interface and a concrete class to provide the default implementation would be a cleaner solution from the object-oriented point of view. In fact, we are currently discussing this.

SimObject

Naturally, you have to override some or all methods of `SimObject` for your own simulation models. The following list shows the relevant methods of `SimObject` with a short description, grouped into methods with similar roles. Because the classes and interfaces of HADES might have changed since this text was written, you should check the details with your copy of the `javadoc`-generated class documentation:

```
public class      SimObject
       extends    Object
       implements Simulatable, Cloneable, Serializable
{
  SimObject()               // the default constructor
  copy()                    // empty implementation

  initialize(String)        // read parameters from a String
  write(PrintWriter)        // write parameters as a String

  // basic accessor and setter methods
  getDebug()                // get or set the debug status,
  setDebug(boolean)         // to dis/enable status messages
  message(String)           // print a message
  setConsole(Console)       // where to send the messages
  toString()                // the usual info method
```

```
  getFullName()           // the hierarchical name of this object
  getName()               // the simple name
  setName(String)         // the simple name

  getParent()             // the toplevel (parent) object, if any
  setParent(SimObject)
  getEditor()             // the editor we're in
  setEditor(Editor)

  // simulation methods
  getSimulator()          // the simulator for this SimObject
  setSimulator(SimKernel)
  getPorts()              // array of all Port objects
  getPort(String)         // get a name Port object

  elaborate(Object)       // initialize the SimObject
  evaluate(Object)        // handle events

  // symbol and graphics stuff
  getSymbolResourceName() // where to find the symbol
  getSymbol()             // the symbol object, if any
  setSymbol(Symbol)       // set a new symbol object
  isVisible()
  setVisible(boolean)     // use true for top-level objects

  needsDynamicSymbol()
  constructDynamicSymbol()
  mousePressed(MouseEvent)
  configure()
}
```

Fortunately, the default implementation of most methods is sufficient for standard simulation objects. For simple models with a static graphical representation, you will only override the constructor, the `getSymbolResourceName()`, and both the `elaborate()` and `evaluate()` methods. The next few paragraphs will explain the role of each group of these methods in more detail.

4.1.1 Constructor and I/O

constructors Because of the special role of constructors in the Java language, and to simplify the instantiation of new objects from design files, HADES uses the following convention. All simulation objects must provide a default constructor (without arguments) which is used to create all instances. While you may provide additional constructors, these are not used by HADES.

Besides the initialization of any class-internal instance variables, the main role of the constructor is to create and initialize the simulation model's external `Ports`. To this end, the array `ports[]` and its members are created. Some classes also define aliases, to allow for more readable code for the simulation model.

component I/O

For specialization, a simulation model may provide a method `initialize()`, which will be called directly after object creation with a string argument. The model can then parse this string to initialize its own variables. Together with its dual function `write()` each simulation model is able to create and maintain its own external representation. The base class `SimObject` provides an empty default implementation for both methods. An example of a typical `initialize()` and `write()` method pair is presented on page 46 below.

4.1.2 Accessors

Most simulation models need not change the basic accessor and setter methods provided by `SimObject`. One important exception are simulation models with a *dynamic symbol*, used to reflect the current state of the object throughout the simulation. These might have to override the `setSymbol()` method. See section 4.1.4 below and the example in section 4.4 for details.

4.1.3 Simulation

elaborate()
evaluate()

Basically, HADES uses only three methods to control the interaction between simulation models and the simulator. At the start of the simulation, the simulator first calls the `elaborate()` method of each component. During the simulation, the simulator will call the `evaluate()` method of the component for each event on the component's input signals. Usually, the simulation engine will provide a `SimEvent` object corresponding to the simulation event as the argument to either `elaborate()` or `evaluate()`. However, both methods expect an `Object` only, which allows to call them directly with arbitrary arguments. A separate method `wakeup()` is provided to handle the special wakeup events used for periodic activities, e.g. from a clock generator component. Together, these three methods are sufficient for discrete event based simulation, because each component can create and schedule further simulation events as a reaction to the `elaborate()` and `evaluate()` calls.

wakeup()

Because most of the object initialization is done by the constructor and possibly the `initialize()` method, the typical `elaborate()` method is very simple. The `evaluate()` method, however, contains the full behavioral model of your simulation object.

4.1.4 Graphics

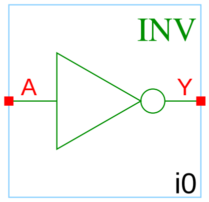
symbols

Besides the methods needed for the simulation functionality, all simulation objects in HADES also have a reference to a `Symbol` object, which defines their graphical representation in the editor. A `Symbol` itself is invisible, but it is used as a container object for an arbitrary number of graphical subobjects. Most components will only need a static object that does not change during simulation. For these components you would normally create a *symbol file* with a textual representation of the graphical objects. Then, the `getSymbolResourceName()` should return the Java 1.1 resource name of the symbol file, like this:

```
public class xx.yy.zz.ImageViewer
       extends SimObject {
  ...
  String getSymbolResourceName() {
    return "/xx/yy/zz/ImageViewer.sym";
  }
}
```

basic symbol objects

At the moment, HADES provides just a few graphics primitives for use in symbols, namely `Polyline`, `Rectangle`, `Circle`, `Arc`, and several styles of `Labels`. The `BboxRectangle` is special, because it is used to define the *bounding box* of the symbol in the editor. The following code shows a typical symbol file for a basic inverter, from `hades/models/gates/Inv.sym`. All coordinates are given in inch/2400 and relative to the top left corner:




```
hades.symbols.BboxRectangle 0 0 2400 2400
hades.symbols.ClassLabel 2350 450 3 INV
hades.symbols.InstanceLabel 2350 2350 3 i0
hades.symbols.PortLabel 150 1130 A
hades.symbols.PortSymbol 0 1200 A
hades.symbols.PortLabel 2100 1130 Y
hades.symbols.PortSymbol 2400 1200 Y
hades.symbols.Polyline 2 0 1200 600 1200
hades.symbols.Polyline 2 1950 1200 2400 1200
hades.symbols.Circle 1800 1200 150 150
hades.symbols.Polyline 4 1650 1200 600 600 600 1800 1650 1200
```

The file shows the use of the `BboxRectangle`, `ClassLabel` and `InstanceLabel` objects. Also, note the `PortSymbol` objects, used to define both the positions and the names of the object's ports. The `PortLabel` objects are just annotation without any other function. Many coordinates in the above example are multiples of 600, so that the corresponding points lie exactly on the default snap grid of 1/4th inch.

other symbol objects

See the class documentation of the `hades.symbol` package for the list of all symbol objects and their parameters. For example, the `ImageObject` and `FigWrapper` object allow to include GIF- and JPEG-images or complete drawings in FIG-format [xfig 3.2] in HADES symbols. Because the planned symbol editor is not ready yet, you have to create the symbol files with a text editor. Compared with the complexity of writing the simulation functionality, however, this is a relatively easy task.

4.1.5 Configuration

configure()

In principle, each simulation object in HADES is responsible for its own configuration. To this end, it can create, initialize, and use its own user interface, namely its own top-level windows. If the user selects the *edit...* command for a simulation object, the editor will call the object's `configure()` method without any parameters. The user interface provided by a simulation object should allow to get and set the relevant parameters for that simulation object. Note that the simulation is not stopped during component configuration.

PropertySheet

While HADES does not restrict the visual editing functionality needed by simulation objects, most components only need a simple user interface. The class `PropertySheet` provides a default user interface which will suffice for most simulation models. To use this class, you would construct a `PropertySheet` object with a list of the variables to modify and a short description for each variable in the `configure()` method of the simulation model. An example for this is shown in section 4.4 on page 49.

4.2 A Simple Example: The AND2 Gate

An AND gate with trivial timing

This section presents the full Java source code for a first implementation of a two-input AND gate. In order to keep the example short, the AND gate uses a trivial timing model with only one parameter `t_delay` for the gate delay that specifies the output delay (in seconds) of the output versus any input changes.

typical imports

As for any Java class, the source code for the AND2 gate model begins with a package statement and some import statements. The import statements in the following code example are typical, because all models need to access classes from the `hades.simulator` and `hades.signals` packages, and digital gates based on the StdLogic1164 logic model need to access the class `hades.models.StdLogic1164`:

```
/* And2.java -- class hades.models.gates.And2 */

package hades.models.gates;

import hades.simulator.*;
import hades.signals.*;
import hades.models.StdLogic1164;

/**
 * And2 - a simple class that models a 2-input AND-gate.
 *        Signals are expected to be StdLogic1164 objects.
 */
public class And2
       extends SimObject {
  ...
}
```

Note that all simulation models in HADES are direct or indirect subclasses of `SimObject`. Therefore, all models implicitly implement the interfaces `Simulatable` and `Serializable`.

the constructor The first task of the constructor of any simulation model in HADES is to create the variables for the *Port* objects used for connection to other models. This is done by allocating the array `ports` inherited from `SimObject` with the necessary number of `Port` objects. Then the individual `Ports` are initialized. Note that the names of the ports have to match the names specified in the symbol file, because the HADES editor searches the names of the `Ports` in the symbol when trying to connect signals.

port aliases Technically, only the `ports` array is needed. However, because it is much more readable to write `port_A` instead of `ports[0]` in the rest of the code, the AND2 class also declares three instance variables as aliases to the `ports` members:

```
...
public class AND2 ... {

  private Port          port_A, port_B, port_Y;
  double                t_delay;

  public And2() {
    super();
    ports = new Port[3];
    ports[0] = new Port(this, "A", Port.IN, null );
    ports[1] = new Port(this, "B", Port.IN, null );
    ports[2] = new Port(this, "Y", Port.OUT, null );

    port_A = ports[0];  // aliases
    port_B = ports[1];
    port_Y = ports[2];

    t_delay = 10.0E-9;  // 10ns, twice inverter
  }

  public String getSymbolResourceName() {
    return "And2.sym";
    // or  "/hades/models/gates/And2.sym"
  }

  public String toString() {
    return "And2: " + getFullName();
  }

  public void elaborate( Object arg ) { ... }
  public void evaluate( Object arg ) { ... }
}
```

toString()

Together with the class header, this code snippet is actually the complete source code for the class AND2. Besides the constructor, it contains a simple implementation of the usual `toString()` method, where `getFullName()` returns the full instance name of the current object, e.g. `/toplevel/adder1/and_134`.

getSymbolResourceName()

The method `getSymbolResourceName()` is used to tell the editor how to find the graphical symbol description for a simulation model. It returns a Java 1.1 *resource name* String. In the example, the editor will try to load the symbol from a resource called `hades.models.gates.And2.sym` via its ClassLoader. The corresponding file name is platform dependent, e.g. `/hades/models/gates/And2.sym` on Unix systems.

elaborate()

It now remains to write the implementation for the `elaborate()` and `evaluate()` methods. Because an AND2 gate needs no special initialization beyond its constructor, the `elaborate()` method is almost empty:

```
/**
 *  elaborate(): This method is called by the simulation
 *  engine to initialize the simulation object.
 */
public void elaborate( Object arg ) {
  simulator = parent.getSimulator();   // super.simulator
}
```

As explained above, the `elaborate()` method is called by the simulator at the start of the simulation. Therefore, this is the right place to get and store a reference to the simulator in the variable `simulator` inherited from `SimObject`. Afterwards, the rest of the code need not bother to check whether the `simulator` reference is valid.

evaluate()

As with all HADES simulation models, the `evaluate()` method specifies the individual behavior of the AND gate. The example code uses the following algorithm. First, to improve simulation performance, it checks whether a signal is connected to the gate's output, `port_Y`. If not, it just returns, else it saves a reference to the signal in the variable `signal_Y`. Then it checks whether signals are connected to the gate inputs `port_A` and `port_B`. If so, it tries to convert the generic `Signal` references to the specific `SignalStdLogic1164` values required by the AND gate and accesses the logic values of the input signals. If no signals are connected to an input, the value `U` is used instead:

```
...
Signal  signal_A; StdLogic1164  value_A;
...
if ((signal_A = port_A.getSignal()) != null)
      value_A = (StdLogic1164) signal_A.getValue();
else  value_A =  StdLogic1164.get_U();
...
```

calculating the output value

Next, the new output value is calculated by calling the `StdLogic1164.and()` method with the values of the `a` and `b` inputs. Finally, the new output value is packed into a new `SimEvent` object and scheduled at the corresponding simulation time by calling the `scheduleEvent()` method of the `simulator`. Because the `evaluate()` method is called whenever an input value changes, we just add the propagation delay `t_delay` of this gate to the current simulation time. The constructor for a `SimEvent` expects four parameters, like this:

scheduling events

```
SimEvent( Signal     target,    // signal connected to port Y
          double     eventTime, // in seconds
          Object     value,     // e.g. a StdLogic1164 object
          Port       source );  // port Y of this AND gate
```

The following code shows the complete `evaluate()` method for the AND gate:

```
/**
 * evaluate(): called by the simulation engine for all events
 * on the input signals of this object. It has to update its
 * internal state and to schedule all required output events.
 */
public void evaluate( Object arg ) {
  Signal        signal_A, signal_B, signal_Y;
  StdLogic1164  value_A,  value_B,  next_Y;

  if ((signal_Y = port_Y.getSignal()) == null) return;

  if ((signal_A = port_A.getSignal()) != null)
        value_A = (StdLogic1164) signal_A.getValue();
  else  value_A =  StdLogic1164.get_U();

  if ((signal_B = port_B.getSignal()) != null)
        value_B = (StdLogic1164) signal_B.getValue();
  else  value_B =  StdLogic1164.get_U();

  next_Y       = StdLogic1164.and( value_A, value_B );
  double  time = simulator.getSimTime() + t_delay;

  simulator.scheduleEvent(
     new SimEvent( signal_Y, time, next_Y, port_Y));
}
```

The method might look complicated at first glance. However, note that most of the code is quite regular and can directly be reused in other simulation models. For example, to write a two-input NOR-gate, you would need to change just one line of code from the above example, namely the call of the `StdLogic1164.and()` method:

```
    ...                            // calculate NOR gate output
    next_Y = StdLogic1164.not(
                         StdLogic1164.or( value_A, value_B ));
    ...
```

Naturally, for the NOR-gate you would also change the `toString()` and the `getSymbolResourceName()` methods. The generalization to other simple gates should now be obvious.

4.3 A D-Flipflop

As a slightly more complex example, this section describes the `evaluate()` method of an edge-triggered D-flipflop with asynchronous reset input. Also, it presents some utility methods (shown in boldface) of the classes `StdLogic1164` and `Signal`. The following example code for the `evaluate()` method was directly taken from the class `hades.models.flipflops.Dffr`. However, to keep example short and readable, some of the verbose debug messages were shortened:

```
/**
 * internal utility method for output event scheduling
 * on outputs port_Q and port_NQ
 */
private void schedule( StdLogic1164 next_Q, double time ) {
  if (port_Q.getSignal() != null) {
    simulator.scheduleEvent(
      new SimEvent( port_Q.getSignal(), time, next_Q, port_Q));
  }
  if (port_NQ.getSignal() != null) {
    simulator.scheduleEvent(
      new SimEvent( port_NQ.getSignal(), time,
                    StdLogic1164.not(next_Q), port_Q ));
  }
}
```

schedule output events

```
/*
 * evaluate() method for a D-flipflop with inputs
 * port_D (data), port_C (clock), and port_NR (reset)
 */
public void evaluate( Object arg ) {
  Signal        signal_D, signal_C, signal_NR;
  StdLogic1164  value_D,  value_C,  value_NR,  next_Q;

  if ((signal_D = port_D.getSignal()) == null) return;
  else  value_D = (StdLogic1164) signal_D.getValue();
```

collect inputs

```
    if ((signal_C = port_C.getSignal()) == null) return;
    else  value_C = (StdLogic1164) signal_C.getValue();

    if ((signal_NR = port_NR.getSignal()) == null) return;
    else  value_NR = (StdLogic1164) signal_NR.getValue();
```

flipflop function

```
    if (value_NR.isLow_0L()) {       // 0 or L: reset flipflop
       next_Q = value_0;
    }
    else if (value_NR.isHigh_1H()) { // 1 or H: flipflop active
      //  rising edge of clock?
      if (signal_C.hasEvent() && value_C.is_1()) {
        if (signal_D.isStable( t_setup )) {
          next_Q = value_D;
        }
        else {
          message( "-W- Setup-time violation on " + toString()
            + " clk at " + simulator.getSimTime()
            + " d   at " + signal_D.getLastChangeTime());
          next_Q = value_X;
        }
      }
      else { // flipflop keeps its value: no output events
        return;
      }
    }
    else { // reset input undefined
      next_Q = value_X;
    }
```

schedule outputs

```
    // we arrive here only, when clock or reset had an event:
    schedule( next_Q, simulator.getSimTime()+t_pass );
  }
```

Note that the above method just returns when either one of the input signals is not connected. This will work sensibly, if the `elaborate()` method schedules `U` values on both the `Q` and `NQ` outputs at time `t=0`:

```
  public void elaborate( Object arg ) {
    simulator = parent.getSimulator();
    schedule( new StdLogic1164() /*U*/, 0.0 );
  }
```

4.4 A Complex Example: The Clock Generator

wakeup()
configure()
graphics

This section presents a more complex example, a clock generator called `ClockGen`. It explains `WakeupEvents` for periodic activities independent of input changes and the `configure()` method necessary for interactive visual editing of model parameters. Also the symbol of the clock generator is "clickable" in the HADES editor, to start or stop the clock generator, and it provides visual feedback about its state.

The clock generator model allows to select arbitrary values for the clock `period` (in seconds) and the fraction of each cycle during which the output is high, called `dutycycle`. Also, the initial `offset` delay until the clock starts can be specified. These values are stored in `double` variables in the following code. The variable `running` stores the current state of the `ClockGen`, and `circleOnOff` is a reference to the circle object used to display the state in the `ClockGens` symbol. Additionally, the variable `portY` is used as an alias to the output port of the `ClockGen`, and `value_U`, `value_0`, and `value_1` will hold the `StdLogic1164` values corresponding to their name:

```
/* ClockGen.java --  class hades.models.io.ClockGen */

package hades.models.io;             // our package

import hades.simulator.*;            // standard imports
import hades.models.*;
import hades.signals.*;

import hades.symbols.*;              // import for graphics
import hades.utils.*;
import javafig.objects.GE_attribs;
import java.awt.*;


/**
 * ClockGen - a SimObject that models a clock generator
 *            with arbitrary frequency, duty cycle and phase.
 *            Default is a symmetric 1 Hz clock signal.
 *            Signals are expected to be StdLogic1164 objects.
 *
 <pre>
                    +-----+     +-----+     +-----+
   +----------------+    +------+    +------+    +------+ ....
   |                |          |          |          |
   |<              >|<        >|<  >|
      offset           period    dutycycle
 </pre>
 */
```

a DOC-comment explaining the timing parameters

```
public class ClockGen
       extends SimObject
       implements Wakeable {

  private Port           port_Y;
  private StdLogic1164   output_U, output_0, output_1;

  /* the timing parameters of this clock generator */
  private double period    = 1.0;   // 1sec period, i.e. 1Hz
  private double dutycycle = 0.5;   // 50% dutycycle, symmetric
  private double offset    = 0.0;   // no initial delay

  private boolean  running = true;
  private Circle   circleOnOff;     // on/off symbol

  ... // constructor and methods go here
}
```

Note that the class `ClockGen` implements the `Wakeable` interface to signal that it understands and handles `WakeupEvents`.

constructor The main function of the constructor, again, is to allocate and initialize the `ports` array of this simulation model. The `ClockGen` has only one output port, and `port_Y` is used as a more readable alias for `ports[0]`. Then the constant output values required over and over by the clock generator are initialized:

```
  /**
   * ClockGen(): construct a default clock generator
   */
  public ClockGen() {
    super();

    ports = new Port[1];
    ports[0] = new Port(this, "clk", Port.OUT, null );
    port_Y = ports[0];

    output_U = new StdLogic1164( StdLogic1164._U );
    output_0 = new StdLogic1164( StdLogic1164._0 );
    output_1 = new StdLogic1164( StdLogic1164._1 );
  }

  public String getSymbolResourceName() {
    return "ClockGen.sym";
  }
```

```
public String toString() {
  return "ClockGen: " + getFullName() + "[timing: "
         + period + "," + dutycycle  + "," + offset + "]";
}
```

For the description of the `getSymbolResourceName()` and `toString()` methods see section 4.2.

elaborate()

The `elaborate()` method is used again to get and store a reference to the `simulator`. However, the `ClockGen` models needs to do more during elaboration than the simple AND gate. To model its initialization (offset) phase with the corresponding `U` output value, a first event with `U` output value is scheduled at the current simulation time `time`. Then a second event with `1` output value is scheduled at time `(time+offset)` to model the start of the duty-cycle in the first clock period. Finally, a `WakeupEvent` at `(time+offset)` is scheduled. This instructs the simulation engine to call the `wakeup()` method of the `ClockGen` at the specified time. The third argument of the `WakeupEvent` constructor is passed as an argument to the `wakeup()` call by the simulator, which allows to distinguish between different wakeup sources. In this example, an internal method `schedule()` is used to make to code more readable:

```
/**
 *  elaborate(): This method is called by the simulation
 *  engine to initialize the simulation object.
 */
public void elaborate( Object arg ) {
  simulator = parent.getSimulator();
  double time = simulator.getSimTime();

  schedule( output_U, time );                 // U now
  schedule( output_1, time+offset );          // U -> 1
  simulator.scheduleWakeup( this, time+offset, this );
}


private void schedule( StdLogic1164 value, double time ) {
  Signal signal = port_Y.getSignal();
  if (signal == null)  return;

  simulator.scheduleEvent(
            new SimEvent( signal, time, value, port_Y));
}
```

wakeup()

The `wakeup()` method is constructed similar to `elaborate()`. It first schedules all output events for one clock cycle, namely the change from `1` to `0` at the current simulation time plus `(dutycycle*period)`, and the change to `1` at the start of the next clock cycle, exactly one `period` later than the current simulation time. Afterwards it schedules a new `WakeupEvent` at the start of the next clock cycle:

```
/**
 * wakeup(): Called by the simulator as a reaction to our own
 * scheduleWakeup()-calls. One period of the clock has
 * elapsed, therefore, schedule the next cycle.
 */
public void wakeup( Object arg ) {
  if (!running) return;
  double time = simulator.getSimTime();
  schedule( output_0, time+dutycycle*period ); // 1 -> 0
  schedule( output_1, time+period );           // 0 -> 1
  simulator.scheduleWakeup( this, time+period, this );
}
```

evaluate() Because the `ClockGen` model has no input, the simulator should not normally call its `evaluate()` method. If `evaluate()` is called in spite of this, we just print an error message:

```
/**
 * evaluate(): called by the simulation engine for all events
 * on the input signals of this object. It has to update its
 * internal state and to schedule all required output events.
 */
public void evaluate( Object arg ) {
  message( "-E- Don't call evaluate() on a ClockGen!" );
}
```

Note that the methods presented above are all that is required for the full simulation functionality of the clock-generator model, including the initialization, its periodical wakeup activities, and the output event scheduling.

I/O The next two methods are required to save and restore `ClockGen` objects with their current parameters to and from a HADES design. The design pattern for both methods is to use one single `String` as the argument or return value. The `write()` method is called on saving a design, and it should return a `String` with a textual representation of all required parameters. The `initialize()` method is called by the editor, which also supplies the `String` written on the previous saving. Both methods have a default implementation in `SimObject`. Therefore, you only need to override these methods if the model needs to store non-default parameters to a design. Note that you should also provide and check the `versionId` variable used by all HADES simulation models:

```
public void initialize( String s ) {
  StringTokenizer st = new StringTokenizer( s );
  try {
   versionId = Integer.parseInt(st.nextToken() );
   period    = Double.valueOf(st.nextToken()).doubleValue();
```

```
      dutycycle = Double.valueOf(st.nextToken()).doubleValue();
      offset    = Double.valueOf(st.nextToken()).doubleValue();
    }
    catch( Exception e ) {
      message( "-E- ClockGen.initialize(): " + e + " " + s );
    }
  }


  public void write( java.io.PrintWriter ps ) {
    ps.print( " " + versionId + " " + period + " "
              + dutycycle + " " + offset );
  }
```

graphics The next part of this section explains how to integrate simulation models with the graphical capabilities of the HADES editor. Because the `ClockGen` model needs to modify its static symbol, it first overrides the `setSymbol()` method inherited from `SimObject`:

```
  public void setSymbol( Symbol s ) {
    symbol = s;
    symbol.setInstanceLabel( name );
    initDisplay();
  }
```

visibility Typically, most simulation objects in a hierarchical design are inside of subdesigns and don't need their symbols, because they are not visible on the top-level. To improve startup time and memory usage, the `setSymbol()` method is called by the editor only for those simulation models in visible designs. While the `setSymbol()` inherited from `SimObject` works fine for static symbols, the clock generator model needs to add a dynamic object to its symbol. Like most other I/O components in HADES, the clock generator uses a small circle called `circleOnOff` to indicate its state. If "running" the circle is filled with red, and if "stopped" the circle is filled with gray, compare figure 10 on page 21. The following method `initDisplay()` is used to create a `Circle` object, to specify its size and position, and to add it to the existing (static) objects of the symbol:

```
  private void initDisplay() {
    circleOnOff = new Circle();
    circleOnOff.initialize( "600 1200 300 300" ); // x y rx ry
    getSymbol().addMember( circleOnOff );

    showState();
  }
```

painter

The `showState()` method is used to display the `circleOnOff` with the current running status of the `ClockGen`. To this end, it checks the value of the `running` variable and sets the fill color of the `circleOnOff` correspondingly. Finally, it checks whether the `painter` object for the circle is set, and then calls the `paint()` method, to request a repaint of the circle:

manipulate attributes

```
private void showState() {
  if (getSymbol() == null) return;

  GE_attribs attr = circleOnOff.get_attribs();
  attr.fillStyle = attr.solidFill;
  if (running)  attr.fillColor = output_1.getColor();
  else          attr.fillColor = output_0.getColor();
  circleOnOff.set_attribs( attr );

  if (circleOnOff.painter != null)
      circleOnOff.painter.paint( circleOnOff );
}
```

GE_attribs

Each graphical object in HADES contains a reference to a `GE_attribs` object which controls its attributes like line and fill color, line and fill style, text font, and layer (depth). See the class documentation for class `javafig.objects.GE_attribs` for a complete description of all attributes and their possible values. The above example shows the typical use of the `GE_attribs` class. Given an graphical object like `circleOnOff`, you first request its current attribute object with the `get_attribs()` method. Next, the desired attributes are specified for this `GE_attribs` object. Finally, you call the `set_attribs()` method for the changes to take effect. Also, note the use of the `getColor()` method to get the default color for a given `hades.models.StdLogic1164` object.

mousePressed()

inside vs. border

The methods `setSymbol()`, `initDisplay()`, and `showState()` allow to visualize the current state of the clock generator. It remains to implement the `mousePressed()` method to allow the user to start and stop the clock generator interactively. The `mousePressed()` method of a `SimObject` is called by the HADES editor every time the mouse is clicked *inside* the symbol of that `SimObject`. Note that mouse clicks on the *border* of a symbol are usually not delivered to the simulation model but intercepted by the HADES editing commands, e.g. *move* and *copy*.

deleting events

The algorithm to implement `mousePressed()` might seem trivial for this example. On each mouse click the `running` variable in negated. If it results in the value `true`, the `elaborate()` method is called to (re-) initialize the clock generator. However, once the clock generator is running, it always has scheduled two `SimEvents` and one future `WakeupEvent` with the simulator. To stop it, it is therefore not sufficient just to reset the `running` variable. Instead, it is necessary to remove all events scheduled by this clock generator from the simulator event list by calling `deleteAllEventsFromSource()`:

```
/**
 * start or stop this ClockGen'erator
 */
public void mousePressed( java.awt.event.MouseEvent me ) {
  SimKernel simulator = parent.getSimulator();

  if (running) { // stop this ClockGen
     if (debug) message( "...stopping " + toString() );
     running = false;
     if (simulator != null)
        simulator.deleteAllEventsFromSource( this );
  }
  else {        // start/restart
     if (debug) message( "...restarting " + toString() );
     running = true;
     elaborate( null );
  }
  // update symbol with current state
  showState();
}
```

SimObjects as Beans

The rest of this section explains how to interface simulation models with the predefined `PropertySheet` class used to access and set model parameters and properties. In principle, all simulation models in HADES are written to be Java-Beans [Sun 96] compliant. However, so far most simulation models use only the two basic requirements of the Beans specification. First, all simulation models provide the standard design pattern for all user-settable properties, that is, the `getX()` and `setX()` methods for each variable `x`. Second, the interaction between components and the simulation engine is based on the Java 1.1 event model, because all event classes in HADES are subclasses of `SimEvent` and therefore `java.util.EventObject`.

Obviously, the three variables `period`, `dutycycle` and `offset` in the clock generator model should be user-settable properties. Therefore the basic getter and setter methods are required for these three variables, while the base class `SimObject` provides the suitable methods `getName()` and `setName()` to edit the instance name:

```
...  // Java Beans compatible getter/setter methods:

public double getPeriod()    { return period; }
public double getOffset()    { return offset; }
public double getDutycycle() { return dutycycle; }

public void setPeriod( double d )    { period = d; }
public void setOffset( double d )    { offset = d; }

public void setDutycycle( double d ) {
```

```
    dutycycle = d;
    if ((dutycycle < 0.0) || (dutycycle > 1.0)) {
       // might print a warning message here
       dutycycle = 0.5;
    }
  }
```

setter(String) Additionally, the clock generator provides three special setter methods with `String` arguments. This allows to use the standard HADES `PropertySheet` class which expects these setter methods:

```
  // special setter methods required by hades.gui.PropertySheet
  public void setPeriod( String s )   { period = parse(s); }
  public void setOffset( String s )   { offset = parse(s); }
  public void setDutycycle( String s) { setDutycycle(parse(s)); }

  public double parse( String s ) {
    double d = 0.0;
    try { d = Double.valueOf(s).doubleValue(); }
    catch( Exception e ) { d = 0.5 }
    return d;
  }
```

configure() Finally, the `configure()` method itself has to be written. This method is called by the editor in reaction to an edit command for the component, e.g. by selecting *edit…* from the popup menu. The default implementation in `SimObject` allows to change the instance name only.

While you can always write your own classes to get and set the components properties, it is much simpler to use the predefined class `hades.gui.PropertySheet`. In order to avoid a multitude of property editors for one single object, the constructor of `PropertySheet` is private, but it provides a `getPropertySheet()` method instead. This method expects two parameters, a reference to the object to edit and an array of `String` with labels and the names of the properties to edit. Each label/variablename pair is used by the class `PropertySheet` to construct one additional line in its window. Typically, the `PropertySheet` for the `ClockGen` component should look like the example shown in figure 7 on page 18.

Once the `PropertySheet` is constructed, you can also specify a help text with instructions about the class' parameters. Finally, you `show()` the `PropertySheet`:

```
  public void configure() {
    String[]  fields = { "instance name:", "name",
                         "period [sec]:", "period",
                         "duty cycle [e.g. 0.5]:", "dutycycle",
                         "offset [sec]:", "offset" }
```

```
  hades.gui.PropertySheet propertySheet;

  propertySheet =
    hades.gui.PropertySheet.getPropertySheet( this, fields );
  propertySheet.setHelpText(
           "ClockGenerator timing parameters:\n"
          + "_________-----_____-----_____-- ...\n"
          + "^        ^    ^    ^    ^    ^          \n"
          + "|offset  | period  | period  | period ... \n"
          + "         | d.c|    | d.c.|   | d.c.| ... \n"
          + "all times are given in seconds " );
  propertySheet.show();
}
```

When the user clicks on either the *Apply* or *OK* buttons of the `PropertySheet`, the setter methods for the simulation objects will be called with the current contents of the `PropertySheets` text fields. Clicking either *Cancel* or *OK* will also close the `PropertySheet` window.

At the moment, the `PropertySheet` class support only the basic setter methods with `String` arguments. We plan to provide other important and standard dialog components like checkboxes, choice menus and buttons soon.

5 Frequently Asked Questions and Known Bugs

This section lists some of the frequently asked questions about HADES. It also describes some of the features and known bugs of HADES version 0.43.

Q: The editor won't load or save my designs

Check your `CLASSPATH` environment variable, possibly set by your HADES startup script. See section 2.1 on page 6 for detailed instructions.

While future versions of HADES will include an interactive and flexible design and library browser, the current version of HADES locates all components and designs via the standard Java 1.1 "class loader", and it expects all designs to be in subdirectories of a directory with the fixed name "`hades`".

Q: How do I cancel a command?

Just press the ESC ("escape") key to interrupt any ongoing editing command. However, you cannot interrupt the *load design* command which may take a while for larger circuits.

Q: The editor seems to hang

This is unlikely, unless you ran out of memory. Typically, you have initiated a command like *move* or *create wire*, and HADES waits for you to click on the object to move or where to connect the signal. Check the status message at the top of the editor window. Then supply the missing parameters or points, or cancel the command.

Q: I got an OutOfMemoryError

This can easily happen when working with large circuits or large simulation traces, because by default the current Java virtual machines start with a memory limit of 16 MBytes. To avoid this problem, start the Java interpreter with the `-mx` option to set a larger memory limit:

```
borneo>java -mx64M hades.gui.Editor &
```

If you start HADES from a script, edit the corresponding line in the startup script to include the `-mx` option.

Q: HADES won't connect wires to some ports

There are two possible reasons, one good and one bad. First, as noted in section 2.3.5 HADES won't let you connect a new wire to a port already used—this is only possible with the *wire add segment* command.

The second situation is a bug in the editor. The editor needs to find the bounding box of the object in order to decide whether to deliver mouse clicks to the simulation object or to initiate an editor command, like *move* or *create wire*. Sometimes, the bounding box calculation fails when the corresponding component has a very long instance name. While the editor will find the correct object, the port symbols seem to be inside of the object (instead of on its border). Workaround: use a shorter instance name that fits inside the standard objects border.

Q: the cursor won't snap to some objects

By default HADES starts with a cursor snapping of 1/4th inch, but some objects have ports on a 1/8th inch grid. See the explanation on page 13 on how to change or deactivate the cursor snap grid.

Q: HADES won't let me delete an object

For *delete*, you must click exactly on the objects corner. Make sure that the cursor snapping is set correctly. Also, see the answer to the previous question on problems with object bounding boxes. Workaround: try clicking on another corner of the object to delete.

Q: The bindkeys don't work at all

Sometimes, the editor won't react to the bindkeys described in section 3.2.1 on page 30. Due to "features" in the Java 1.1 *keyboard focus* policy for WINDOWS'95, the current version of HADES does not activate the keyboard focus automatically for the object canvas. Just click anywhere in the object canvas to set the keyboard focus. Afterwards, the editor will accept the bindkeys.

Q: I want to delete a part of the wire

Sorry, but this is not possible with HADES version 0.43. You can only delete the whole wire.

Q: I get timing violations from my flipflops

There are two possible reasons. First, your design might contain problematical circuits, e.g. gated Clocks. Second, input changes during interactive simulation might occur at times near clock changes, especially on slow computers or under high simulation load. See section 2.6.1 on page 25 for a detailed explanation.

Q: I clicked 'create symbol' for design 'xx.hds', but the editor cannot find it

At the moment, the *create symbol* command writes its output always to a file named `_unnamed.sym` in the current working directory. Rename this file to `xx.sym` from the DOS/Unix shell and move it to the same directory as the `xx.hds` design file. Also, read the explanation to the "editor won't load or save my designs" question above.

Q: The editor won't accept to rename a component/signal

To keep its data structures consistent, the editor will not accept duplicate names. Instead, it will automatically construct a new and unique name by appending a small integer to your name. Also, some special characters (like spaces) are not allowed in names, and will be changed to underscore characters.

Q: I have multiple outputs connected to one signal

So far, HADES does not support signals driven by multiple sources (busses). Instead, the last value assigned to the signal will be used for the simulation. This implies that HADES won't detect short circuit situations in your designs.

Q: operation X is slow

The current version of HADES still uses inefficient algorithms for some infrequent operations. For example, wire creation should be acceptably fast, while wire deletion is significantly slower. Also note that a typical circuit schematic may easily contain several thousand graphical objects (hundreds of component symbols with twenty or more graphical subobjects each). Please don't hesitate to provide us with feedback about which commands need improvement.

5.1 Design issues

I need a two-phase clock

To create a two-phase clock, create two `ClockGen` components with the same clock periods, the corresponding values for the duty cycle, and finally set the offset of one of the clocks to achieve the desired effect. For example, to create a non-overlapping 10 MHz clock with waveforms "`_1__`" and "`___1`", specify periods of 100 nsec., a dutycycle of 0.25 (25%) and offsets of 0 nsec. and 50 nsec., respectively.

Naturally, you could also create a simple state machine subdesign to create multiple clocks from a single clock input.

5.2 Known bugs in HADES 0.43

Bug: spurious objects displayed

During simulation, many threads—the simulation thread, the AWT event input handler thread, and AWT image drawing threads— initiate redraw operations on the object canvas. Due to bugs in the Java 1.1 thread scheduling, there is a race condition which may leave the HADES editor display (but not the internal data structures) in inconsistent state. Just select the *redraw* menu time from the main *Edit* menu to repaint all objects.

Under high simulation load, even the object canvas grid may temporarily be damaged. If this happens, pause the simulation, select *redraw*, then restart the simulation.

Bug: the 'run for' command may deadlock

Under rare circumstances, the current implementation of the simulation engine may deadlock after several invocations of the *run for* command.

References

[Cadence 97] *Design Framework II 4.41 User Manual*, Cadence Design Systems, Inc., San Jose, CA, USA, 1997

[DIGSIM] X.Y.Z. DIGSIM

[Gillespie & Lazzaro 96] D. Gillespie and J. Lazzaro, *DigLOG and AnaLOG, Caltech VLSI CAD Tools*, California Institute of Technology, 1996 `ftp://ftp.pcmp.caltech.edu/pub/chipmunk/`

[IEEE-87] *IEEE Standard VHDL Hardware Description Language Reference Manual*, IEEE Std 1076-1987, 1988

[IEEE-93] *IEEE Standard VHDL Hardware Description Language Reference Manual*, IEEE Std 11xx-1993, 1993

[Hendrich 97] N. Hendrich, *HADES — A Java-based visual simulation environment*, in preparation, Fachbereich Informatik, Universität Hamburg, 1997

[Khoral 97] *KHOROS 2.2*, Khoral Research Inc., Albuquerque, New Mexico, US, 1997

[CACI 97] CACI, *SIMGRAPHICS-II User's Manual*, CACI Products Company, La Jolla, CA, 1997

[Sun 95] *The Java Whitepaper*, Sun Microsystems, Mountain View, CA., 1995

[Sun 96] *Java Beans Specification, version 1.0*, Sun Microsystems, Mountain View, CA., 1996 `http://java.sun.com/beans`

[Sun 97] *Sun 100% pure Java initiative*, Sun Microsystems, Mountain View, CA., 1997 `http://java.sun.com/purejava/`

[Synopsys 97] *Synopsys VSS 3.5a user manual*, Synopsys Inc., Mountain View, CA, 1997

[xfig 3.2] Supoj Sutanthavibul, Paul King, Brian Smith, *XFIG — Facility for Interactive Generation of figures under X11*, Lawrence Berkeley Laboratory, CA, 1996